Report project PHPC-2020

A High Performance Implementation of the Conjugate Gradient solver

Principal investigator (PI)	Pierre Vuillecard
Institution	EPFL-CSE
Adresse	Station 1, CH-1015 LAUSANNE
Date of submission	July 3, 2020

Abstract

The Conjugate Gradient (CG) is an algorithm that aims to find the solution of a systems of linear equations. This method is widely use to solve partial differential equations or optimization problems because it works well with large sparse matrix and requires small computation time. This work presents a parallel version of the conjugate gradient using Message Passing Interface (MPI) and also a version that uses cuda.

1 Scientific Background

The CG is typically used to find the solution of a systems of linear equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is symmetric and positive definite. $\mathbf{A}$ is an $N \times N$ matrix, $\mathbf{x}$ and $\mathbf{b}$ are $N \times 1$ vector. The main objective of this algorithm is to minimize the residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$. In each iteration, $\mathbf{x}$ is updated by moving in the search direction $\mathbf{p}$ by α. A pseudo code of the CG solver can be seen in algorithm 1. The algorithm complexity is in $\mathcal{O}(N^2)$ per iteration. The conjugate gradient ensures that the number of iteration does not exceed N and it depends on the conditioning number of the matrix $\mathbf{A}$. The more consuming part of each iteration is the matrix-vector multiplication that it's done in $\mathcal{O}(N^2)$. The second big operation matrix are the dot product done in $\mathcal{O}(N)$ and the $x = x + ay$ operation which is done in $\mathcal{O}(N)$.

2 Implementations

The application is implemented in C [1]. We have investigated four implementations of the same solver :

- A distributed memory version using the MPI library [3] and with dense matrix storage.
- A distributed memory version using the MPI library [3] and with sparse CSR matrix storage.
- A cuda version that works with GPU using dense matrix storage.
- A cuda version that works with GPU using sparse CSR matrix storage.

The code has been fully debugged using the `gdb` debugger. The `Valgrind` tool has been used to remove all the memory leaks.

Algorithm 1: Serial conjugate gradient

```
1  $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$
2  $\mathbf{p} = \mathbf{r}$
3  $r_{sqrd} = \mathbf{r}.\mathbf{r}$
4  $\mathbf{x} = \mathbf{x}_0$
5  for $iter=0$ to $maxiter$ do
6      $r_{sqrdold} = r_{sqrd}$
7      $\alpha = \frac{r_{sqrd}}{\mathbf{pAp}}$
8      $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$
9      $\mathbf{r} = \mathbf{r} - \alpha\mathbf{Ap}$
10     $r_{sqrd} = \mathbf{r}.\mathbf{r}$
11     if $\sqrt{r_{sqrd}} < tol$ then
12         break
13     end
14     $\mathbf{p} = \mathbf{r} + r_{sqrd}/r_{sqrdold}\mathbf{p}$
15 end
16 solution is x
```

2.1 Message Passing Interface (MPI)

Two of the parallel version of the CG solver presented in this work is based on a distributed system using MPI. This method uses domain decomposition, where $\mathbf{A}$, $\mathbf{x}$, $\mathbf{p}$ and $\mathbf{b}$ are divided into horizontal slices with each processor being responsible of the calculation involving in each slice. The number of sub domains is specified by the number of processes p . Now each processor take care of $\mathbf{A}^{sub}$ of size $\frac{N}{p} \times N$,and $\mathbf{x}^{sub}$, $\mathbf{p}^{sub}$, $\mathbf{b}^{sub}$ of size $\frac{N}{p} \times 1$. With the data-parallel approach, each processor has an approximately equal amount of data and operation to compute. A pseudo code of the parallel CG solver using MPI can be seen in algorithm 2.

The two major function used is :

- The dot product, each processor compute a part of the dot product. Then we use an *Allreduce* to share the computation of each processor and sum it to get the final result.
- The matrix vector multiplication, each processor compute a part of the matrix vector multiplication as each processor have a part of the matrix that is divided by row.

One major advantage of *Allreduce* is that it waits the other processor. Thus, it is safe to use. Also it ensures load balancing as the *Allgather* function that is used to share the vector $\mathbf{p}$ to every processor.
The computational complexity for the dense matrix is about $\mathcal{O}(\frac{N(N+1)}{p})$ because the matrix vector product is done in $\mathcal{O}(\frac{N^2}{p})$ and the dot product is done in $\mathcal{O}(\frac{N}{p})$. The communication complexity which is around $\mathcal{O}(log(p)+N)$. We have a final complexity of $\mathcal{O}(\frac{N(N+1)}{p}+log(p)+N)$ per iteration. Concerning the sparse matrix if NZ is the number of non-zero element, the final complexity should be around $\mathcal{O}(\frac{NZ+N}{p} + log(p) + N)$. Finally for the memory complexity it should be around $\mathcal{O}(N^2)$ for the dense matrix and $\mathcal{O}(NZ)$ for the sparse matrix.
Also a finer optimization is done using the blas library (see openblas) , which is optimal for basic linear algebra computation.

For the sparse matrix we use the CSR format wich considerably reduce the space storage from N^2 to NZ, where NZ is number of non-zero element in the matrix. It is also quick easy to do operation with this kind of storage.

Algorithm 2: Parallel conjugate gradient

```
1  $\mathbf{r}^{sub} = \mathbf{b}^{sub} - \mathbf{A}^{sub}\mathbf{x}$
2  $\mathbf{p}^{sub} = \mathbf{r}^{sub}$
3  $r_{sqrd}^{sub} = \mathbf{r}^{sub}.\mathbf{r}^{sub}$
4  MPI_Allreduce($r_{sqrd}^{sub}$, $r_{sqrd}$, MPI_SUM )
5  $\mathbf{x}^{sub} = \mathbf{x}_0^{sub}$
6  for $iter=0$ to $maxiter$ do
7      $r_{sqrdold} = r_{sqrd}$
8      $\mathbf{Ap}^{sub} = mat\_vec\_mul(\mathbf{A}^{sub}, \mathbf{p})$
9      $pAp^{sub} = \mathbf{p}^{sub}.\mathbf{Ap}^{sub}$
10     MPI_Allreduce($pAp^{sub}$, $pAp$, MPI_SUM )
11     $\alpha = \frac{r_{sqrd}}{pAp}$
12     $\mathbf{x}^{sub} = \mathbf{x}^{sub} + \alpha\mathbf{p}^{sub}$
13     $\mathbf{r}^{sub} = \mathbf{r}^{sub} - \alpha\mathbf{Ap}^{sub}$
14     $r_{sqrd}^{sub} = \mathbf{r}^{sub}.\mathbf{r}^{sub}$
15     MPI_Allreduce($r_{sqrd}^{sub}$, $r_{sqrd}$, MPI_SUM )
16     if $\sqrt{r_{sqrd}} < tol$ then
17         break
18     end
19     $\mathbf{p}^{sub} = \mathbf{r}^{sub} + r_{sqrd}/r_{sqrdold}\mathbf{p}^{sub}$
20     MPI_Allgather($\mathbf{p}^{sub}$, $\mathbf{p}$)
21 end
22 MPI_AllGather($\mathbf{x}^{sub}$, $\mathbf{x}$,Root=0)
23 solution is x for processor 0
```

2.2 Compute Unified Device Architecture (CUDA)

We proposed a cuda version of the conjugate gradient. Their is two major function that were implemented which is the matrix vector multiplication and the dot product. They both take advantage of the domain decomposition to do the computation in parallel. We may vary how the decomposition is done to see if it change the parallel efficiency of the code.
We also use the CSR format to store the sparse matrix in the sparse version.

3 Prior results: MPI version

We ran our production version on the fidis cluster at EPFL. This machine presents the following characteristics [4] :

- 408 compute nodes each with 2 Intel Broadwell processors running at 2.6 GHz, with 14 cores each (28 cores per machine),

- Infiniband FDR fully-non-blocking connectivity with a fat-tree topology
- GPFS filesystem
- Total peak performance : 475 TF

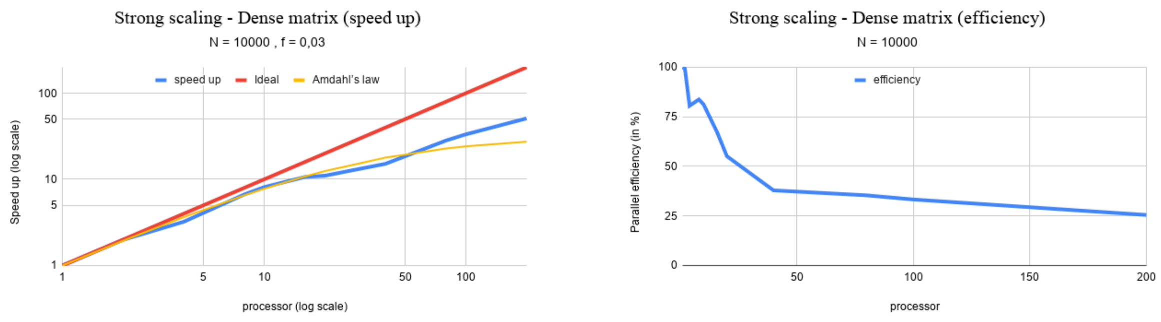
3.1 Using dense matrix

3.1.1 Strong scaling

For the strong scaling we fix the problem size and we increase the number of processor. This gives an estimation of the speed up and efficiency for a fixed size problem. Amdahl's law define the speed up as :

$$S(p) := \frac{1}{f + \frac{(1-f)}{p}} \tag{1}$$

where p is the number of processor and f is serial part of the code. Thus if we measure the speed up for the strong scaling we can estimate the serial part of the code f. In figure 1 (a), we can see that the estimation of f is approximately 0,03. This would mean that 3% of the code is serial. However it's just an estimate because as you can see the fit of Amdahl's law is bad, especially for bigger number of processor. In the efficiency plot, figure 1 (b) we can see that the efficiency drop when the number of processor is bigger than 10. Moreover we have more than 50% of parallel efficiency if the number of processor is lower than 20. In figure 2 we have increased the job size by 4 and did a strong scaling speed up and efficiency measurement. The results are slightly better especially for the efficiency plot (b). In dead the efficiency decreases much slower, we have more than 50% of parallel efficiency if the number of processor is lower than 40.

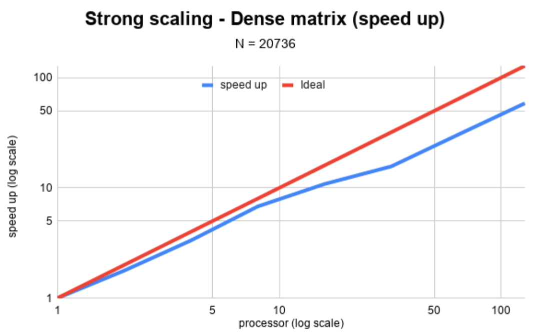


(a) Speed up (b) Efficiency

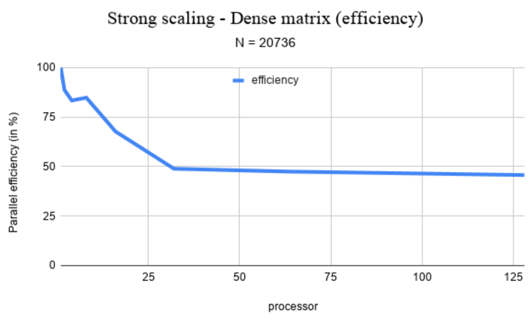
Figure 1: Strong scaling, for a fixed job size $N \times N$ with $N = 10000$. We increase the number of processor from 1 to 200. The speed up is compute such that $S(p) = \frac{t_1}{t_p}$ and the efficiency is $E(p) = \frac{S(p)}{p}$ where p is the number of processor and t_p is the time using p processor. Note also that the time t is the time per iteration

3.1.2 Weak scaling

For the weak scaling we increase the job size and the number of processor linearly. This gives an idea of how the code scale if we increase the size of the job. Gustafson's law define the scale



(a) Speed up



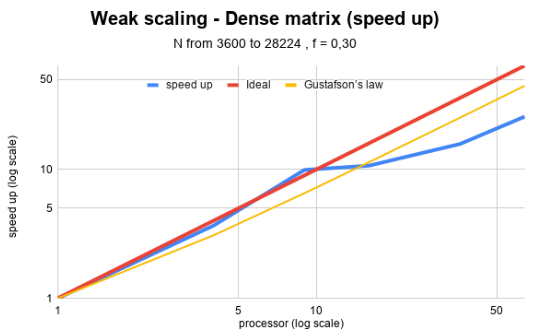
(b) Efficiency

Figure 2: Strong scaling, for a fixed job size $N \times N$ with $N = 20736$. We increase the number of processor from 1 to 128. The speed up is compute such that $S(p) = \frac{t_1}{t_p}$ and the efficiency is $E(p) = \frac{S(p)}{p}$ where p is the number of processor and t_p is the time using p processor. Note also that the time t is the time per iteration

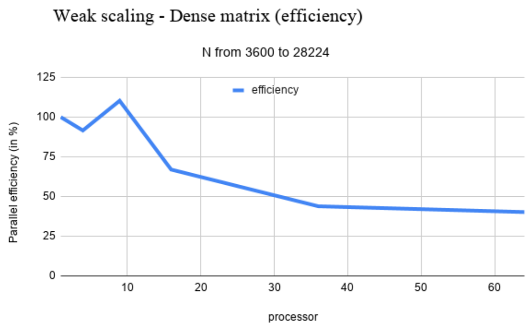
speed up as :

$$S(p) := p + f(1 - p) \tag{2}$$

Where $S(p)$ is the scale speed up , p is the number of processor and f is the serial part of the code. Thus here again if we measure the speed up for the strong scaling we can estimate the serial part of the code f. In figure **??** (a) we can see that we fitted the Gustafson's law and we had f=0,3 which is 10 times bigger than the estimation of f using the Amdahl's law. According to Gustafson's law 30% of the code is serial. Here again the result is to take carefully because the fit is also not very good. Concerning the efficiency plot in figure 3 (b) we can see a drop at 10 processor. Then it decrease quit slowly to 40%. Moreover we have more than 50% of parallel efficiency if the number of processor is lower than 30.



(a) Speed up

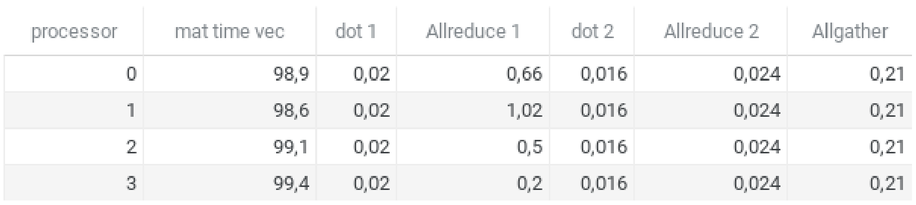


(b) Efficiency

Figure 3: Weak scaling, for an increasing job size $N \times N$ from $N = 3600$ to $N = 28224$ and we also increase the number of processor from 1 to 64. The scale speed up is compute such that $S(p) = p\frac{t_1}{t_p}$ and the efficiency is $E(p) = \frac{S(p)}{p} = \frac{t_1}{t_p}$ where p is the number of processor and t_p is the time using p processor. Note also that the time t is the time per iteration

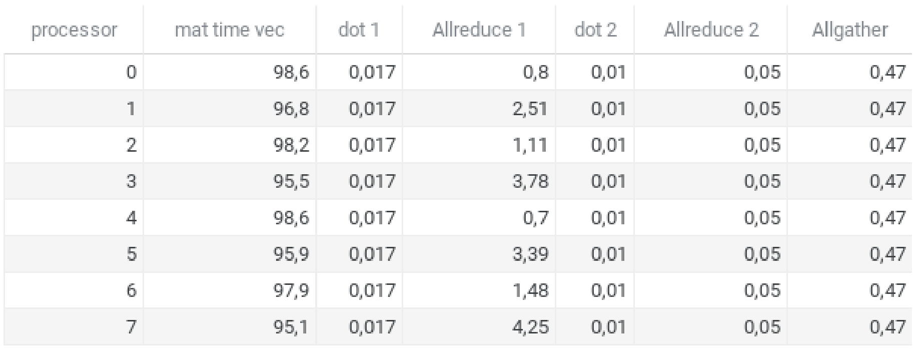
3.1.3 Investigation

In this section we try to investigate how the code react when we increase the number of processor. In figure 4 and 5 we measure the proportion of time spend in specific region of the code if we increase the number of processor. We can see that the majority of the time is spend in the matrix vector multiplication. It is expected since the complexity is $\mathcal{O}(\frac{N^2}{p})$. Also if we increase the number of processor we see that the proportion of time spend in *Allreduce* and *allgather* increase. Here again it is expected since the message pass through more processor. However we can see that the proportion of time in *Allreduce 1* fluctuate a lot. I think that the time spend in the matrix vector multiplication is not the same for all the processor as shown the figure even if they have exactly the same among of computation. As *Allreduce 1* need to wait the results of the slower processor this lead to a bit of wast of time. The final time after the reduction depend of the slower processor.
Furthermore I also try to fix the number of processor and increase the number of nodes involve to see how the bandwidth influence the code. Results are showed in figure 6. We can see that increasing the number of nodes doesn't seem to slow down the code.



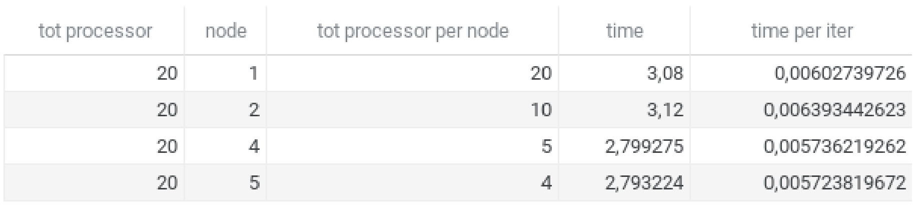
processor	mat time vec	dot 1	Allreduce 1	dot 2	Allreduce 2	Allgather
0	98,9	0,02	0,66	0,016	0,024	0,21
1	98,6	0,02	1,02	0,016	0,024	0,21
2	99,1	0,02	0,5	0,016	0,024	0,21
3	99,4	0,02	0,2	0,016	0,024	0,21

Figure 4: Proportion of time spend in specific part of the code per iteration and per processor. The result came from the mpi implementation for the dense matrix of size $N = 10000$ using 4 processors.



processor	mat time vec	dot 1	Allreduce 1	dot 2	Allreduce 2	Allgather
0	98,6	0,017	0,8	0,01	0,05	0,47
1	96,8	0,017	2,51	0,01	0,05	0,47
2	98,2	0,017	1,11	0,01	0,05	0,47
3	95,5	0,017	3,78	0,01	0,05	0,47
4	98,6	0,017	0,7	0,01	0,05	0,47
5	95,9	0,017	3,39	0,01	0,05	0,47
6	97,9	0,017	1,48	0,01	0,05	0,47
7	95,1	0,017	4,25	0,01	0,05	0,47

Figure 5: Proportion of time spend in specific part of the code per iteration and per processor. The result came from the mpi implementation for the dense matrix of size $N = 10000$ using 8 processors.



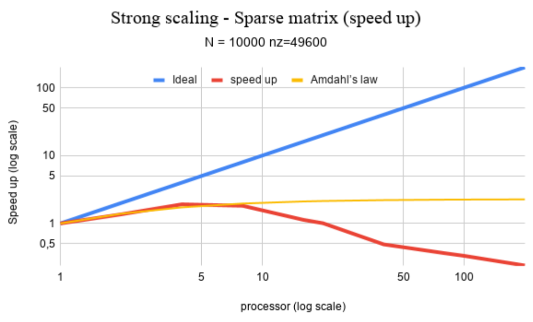
tot processor	node	tot processor per node	time	time per iter
20	1	20	3,08	0,00602739726
20	2	10	3,12	0,006393442623
20	4	5	2,799275	0,005736219262
20	5	4	2,793224	0,005723819672

Figure 6: Bandwidth test for a fix number of processor we increase the number of node and measure the time spend in the algorithm and also per iteration.

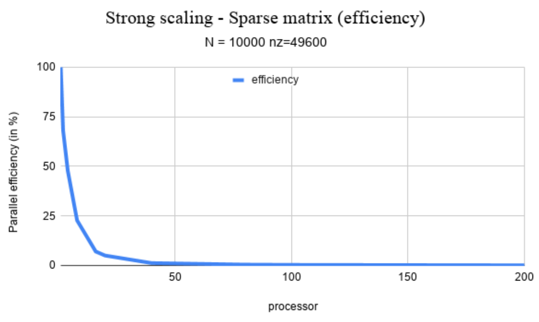
3.2 Using sparse matrix

3.2.1 Strong scaling

We did the same for the code using the sparse matrix. In figure 7 we can see that for the sparse matrix the speed up that we could achieve is very low compare to the one with the dense matrix. In fact the speed up doesn't exceed 2. Also we can see that if we use more than 4 processor then the speed up start to decrease. I decide to fit the Amdahl's law only with 1 2 and 4 processor. It gives an estimation of the serial part of the code that is $f \approx 0,43$. This means that 43% part of the code is serial wich is very bad. Also we can see in the efficiency plot in figure 7 (b) that it drop very quickly. Using 2 processors we had 68% of efficiency and 47% using 4 processors.



(a) Speed up

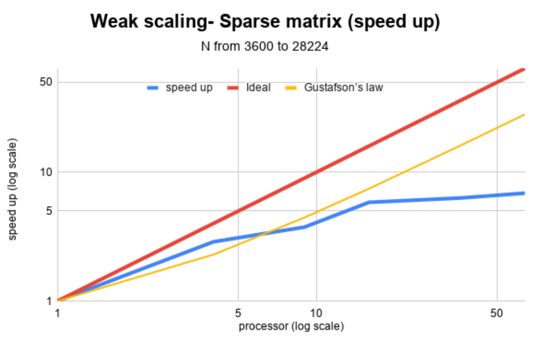


(b) Efficiency

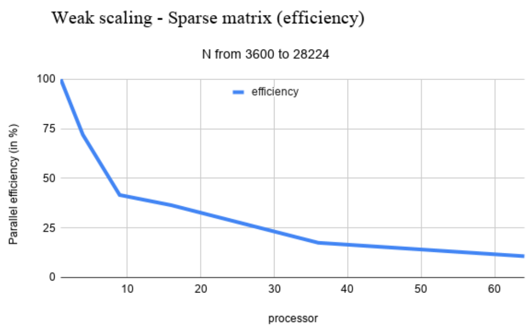
Figure 7: Strong scaling, for a fixed job size $N \times N$ with $N = 10000$. We increase the number of processor from 1 to 200. The speed up is compute such that $S(p) = \frac{t_1}{t_p}$ and the efficiency is $E(p) = \frac{S(p)}{p}$ where p is the number of processor and t_p is the time using p processor. Note also that the time t is the time per iteration

3.3 Weak scaling

We again did the same computation as for the dense matrix. Here also we could try to fit the Gustafson's law with the same number of processor that we use in the strong scaling. In figure 8 (a) we can estimate the serial part by fitting the Gustafson's law, which in this case gives $f \approx 0,56$. Here again as the results are not very good fitting the Gustafson's law doesn't give good estimate of f. In the efficiency plot in figure 8 (b) we can see that we have more than 50% of parallel efficiency if the number of processor is lower than 8. The problem seems to not scale very well.



(a) Speed up

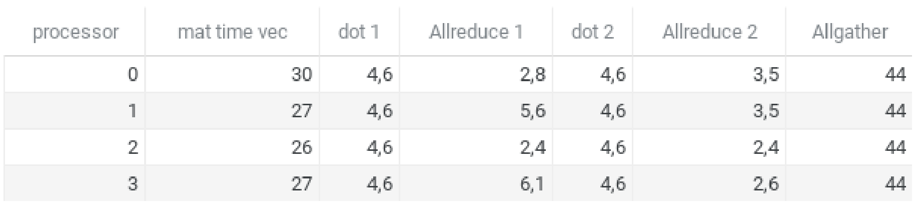


(b) Efficiency

Figure 8: Weak scaling, for an increasing job size $N \times N$ from $N = 3600$ to $N = 28224$ and we also increase the number of processor from 1 to 64. The scale speed up is compute such that $S(p) = p\frac{t_1}{t_p}$ and the efficiency is $E(p) = \frac{S(p)}{p} = \frac{t_1}{t_p}$ where p is the number of processor and t_p is the time using p processor. Note also that the time t is the time per iteration

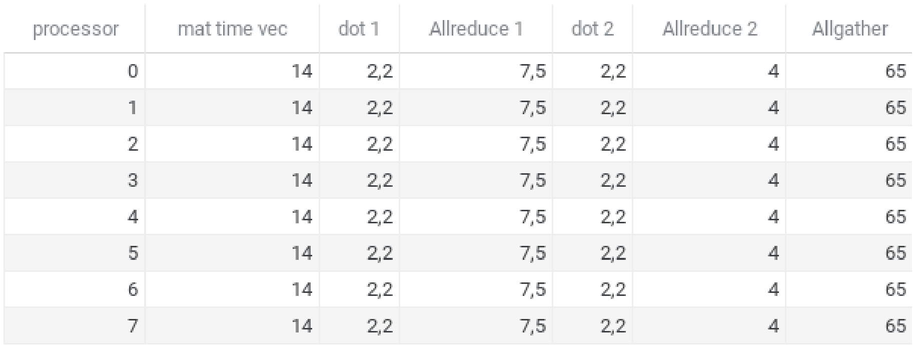
3.3.1 Investigation

In this section we try to investigate how the code react when we increase the number of processor. In figure 9 and 10 we measure the proportion of time spend in specific region of the code if we increase the number of processor. Here we can see that since the matrix is sparse the matrix vector multiplication s much faster. In fact as $NZ = 49600$ which correspond to non zero element the matrix vector multiplication should only be 5 time longer than the dot product. This is confirmed in the results in the figure. Now we can see that the code pass the major part of the time in *Allgather* and as we see this part will be longer if we increase the number of processor this why it leads to bad result in the strong and weak scaling. The code pass the majority of the time in communication rather than true computation as the number of processor increase.



processor	mat time vec	dot 1	Allreduce 1	dot 2	Allreduce 2	Allgather
0	30	4,6	2,8	4,6	3,5	44
1	27	4,6	5,6	4,6	3,5	44
2	26	4,6	2,4	4,6	2,4	44
3	27	4,6	6,1	4,6	2,6	44

Figure 9: Proportion of time spend in specific part of the code per iteration and per processor. The result came from the mpi implementation for the sparse matrix of size $N = 10000$ and $NZ = 49600$ using 4 processors.



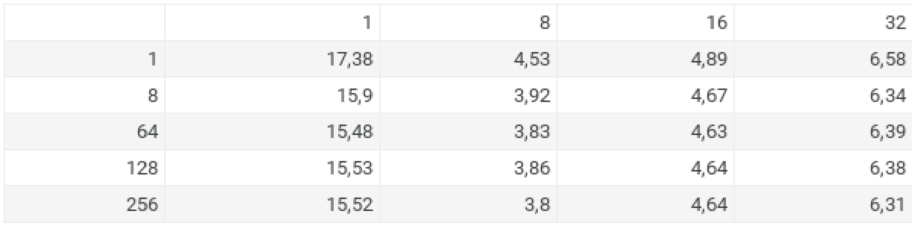
processor	mat time vec	dot 1	Allreduce 1	dot 2	Allreduce 2	Allgather
0	14	2,2	7,5	2,2	4	65
1	14	2,2	7,5	2,2	4	65
2	14	2,2	7,5	2,2	4	65
3	14	2,2	7,5	2,2	4	65
4	14	2,2	7,5	2,2	4	65
5	14	2,2	7,5	2,2	4	65
6	14	2,2	7,5	2,2	4	65
7	14	2,2	7,5	2,2	4	65

Figure 10: Proportion of time spend in specific part of the code per iteration and per processor. The result came from the mpi implementation for the sparse matrix of size $N = 10000$ and $NZ = 49600$ using 8 processors.

4 Prior results: CUDA version

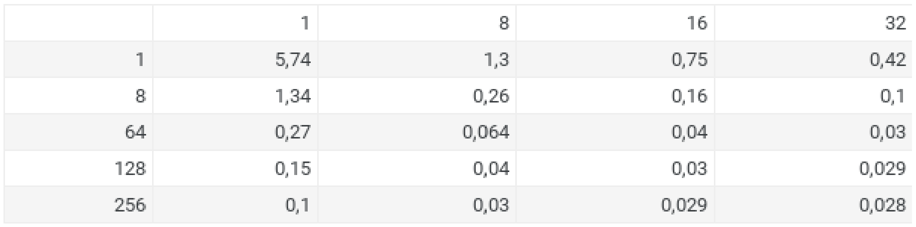
We ran our production on an NVIDIA GPU, which is the Geforce RTX 2070 MAX Q, More characteristic of this GPU can be found in [5]. Here again we propose two version one that use the dense matrix and the other that use the sparse matrix.

In figure 11 and 12 the time that it takes to run the conjugate gradient using different blocks size (column) and numbers of threads per block (row). Figure 11 shows the results for the dense matrix and figure 10 shows the results for the sparse matrix. For the sparse matrix we can see that if we increase the blocks size or the number of threads per block then it leads to better performance. However we can see that for the dense matrix if we increase the block size the time seems to increase from size 8. We find the best time if we use a block of size 8.



	1	8	16	32
1	17,38	4,53	4,89	6,58
8	15,9	3,92	4,67	6,34
64	15,48	3,83	4,63	6,39
128	15,53	3,86	4,64	6,38
256	15,52	3,8	4,64	6,31

Figure 11: Time spend to solve the Conjugate Gradient in second using different blocks size (column) and different threads per blocks (row). The dense matrix has size $N = 10000$. It takes for all computation the same number of iteration.



	1	8	16	32
1	5,74	1,3	0,75	0,42
8	1,34	0,26	0,16	0,1
64	0,27	0,064	0,04	0,03
128	0,15	0,04	0,03	0,029
256	0,1	0,03	0,029	0,028

Figure 12: Time spend to solve the Conjugate Gradient in second using different blocks size (column) and different threads per blocks (row). The sparse matrix has size $N = 10000$ and $NZ = 49600$. It takes for all runs the same number of iteration.

5 Resources budget

This section presents the resources that we need if we would like to solve the conjugate gradient on a very large matrix N. As an example we suppose that N=100000. Compare to the weak scaling computation the job size is increase by 100. We have seen that for 20 processor the efficiency was above 50% so we could take ideally 2000 core but as it doesn't scale perfectly we can take 1700 processor. If now, we consider a sparse matrix of the same size, then we saw that for 2 processors it works well. So now, we need 200 processors ideally but the weak scaling is very bad so we can consider only 100 processors.

5.1 Computing Power

?

5.2 Raw storage

?

5.3 Grand Total for a dense matrix of size N=100000

Total number of requested cores	1700
Minimum total memory	$\mathcal{O}(N^2 + 2N)$
Maximum total memory	$\mathcal{O}(2N^2)$
Temporary disk space for a single run	2GB
Permanent disk space for the entire project	10GB
Communications	MPI
License	own code (BSD)
Code publicly available ?	NO
Library requirements	openblas ompenmpi gcc
Architectures where code ran	Intel Broadwell

5.4 Grand Total for a Sparse matrix of size N=100000

Total number of requested cores	100
Minimum total memory	$\mathcal{O}(NZ + 2N)$
Maximum total memory	$\mathcal{O}(N^2)$
Temporary disk space for a single run	1GB
Permanent disk space for the entire project	5GB
Communications	MPI
License	own code (BSD)
Code publicly available ?	NO
Library requirements	openblas ompenmpi gcc
Architectures where code ran	Intel Broadwell

6 Scientific outcome

This work shows that it may be hard to evaluate the scaling of an algorithm. We see that the conjugate gradient using dense matrix scale quit well. But if using sparse matrix the scale is very bad.

References

[1] Kernighan, Brian W. and Ritchie, Dennis M.,*The C Programming Language Second Edition*, Prentice-Hall, Inc.,1988

[2] Dagum L. and Menon R.,*OpenMP: An Industry-Standard API for Shared-Memory Programming*, IEEE Computational Science & Engineering, Volume 5 Issue 1, pp 46-55, January 1998

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

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

[5] NVIDIA, `https://www.nvidia.com/fr-fr/geforce/graphics-cards/rtx-2070/`