PROJECT:
CONJUGATE GRADIENT METHOD BY STENCIL APPLICATION

Parallel and Distributed Programming 1TD070, 5 hp

Erik Bjerned

2023-05-24

UPPSALA
UNIVERSITET

Contents

1 Introduction

Solving systems of equations has gone from being a slow and tedious example of mathematics to become an integral part of many areas of science. As more and more work is done in software and complexity rises, it is necessary to be able to solve these systems with accuracy and speed. There are of course many ways and algorithms to do this, but a common and well known method is the conjugate gradient method. This method is in theory capable of solving very large systems exactly, but in practice this is bounded by rounding errors made by computers. But still, the method is capable of presenting a solution that is very close to the exact and is thus very useful. As the systems get larger the computation time increases exponentially, therefore it is preferable to solver many smaller system, which is why it is interesting and useful to parallelize this method.

In this report the conjugate gradient method is implemented to run in parallel and is performance tested on UPPMAX, with the goal of implementing an optimized and scalable solution for problems of varying sizes.

2 Problem description

The conjugate gradient (CG) method is an algorithm used to solve linear systems of equations $A\mathbf{u} = \mathbf{b}$, under the prerequisite that A is symmetric and positive definite. In this task, the method is applied to a mesh with predetermined values of $\mathbf{b}$ where the task is to solve for $\mathbf{u}$. A could be described as a stencil being applied to each mesh node, if then the mesh were to be $n \times n$ sized, A would be $N \times N = n^2 \times n^2$ and thus $\mathbf{u}, \mathbf{b} = N \times 1$. The CG-method could be performed as a direct or iterative solver, where the iterative is more suited towards larger systems such as this one. The given iterative method can be seen below.

Algorithm 1 The CG algorithm

1: Initialize $\mathbf{u} = \mathbf{0}, \mathbf{g} = -\mathbf{b}, \mathbf{d} = \mathbf{b}$
2: $q_0 = \mathbf{g}^T\mathbf{g}$
3: **for** $it = 1, 2, \cdots$ until convergence **do**
4: $\quad \mathbf{q} = A\mathbf{d}$
5: $\quad \tau = q_0/(\mathbf{d}^T\mathbf{q})$
6: $\quad \mathbf{u} = \mathbf{u} + \tau\mathbf{d}$
7: $\quad \mathbf{g} = \mathbf{g} + \tau\mathbf{q}$
8: $\quad q_1 = \mathbf{g}^T\mathbf{g}$
9: $\quad \beta = q_1/q_0$
10: $\quad \mathbf{d} = -\mathbf{g} + \beta\mathbf{d}$
11: $\quad q_0 = q_1$
12: **end for**

The matrix A is in this case a sparse matrix as a 3×3 stencil is applied to each mesh node. Thus can the matrix-vector multiplication be transformed into only handling the non-zero elements and reduce the complexity from N^2 to $10N$.

Table 1: Stencil applied on mesh nodes.

$(j+1)$		-1	
j	-1	4	-1
$(j-1)$		-1	
	$(i-1)$	i	$(i+1)$

The solution vector $\mathbf{b}$ has its elements described by $b_{ij} = 2h^2(x_i(1-x_i) + y_j(1-y_j)$ apart from where $x_i = 0,\ y_j = 0,\ x_i = nh,\ y_j = nh \implies b_{ij} = 0$

3 Solution method

The choice of topology for this implementation is a non-periodic Cartesian grid, thus each process has a corresponding square of nodes in the mesh. This is performed by using `MPI_Cart_create` and `MPI_Cart_shift` as this determines the appropriate ranks of neighbouring processes in the topology. This is relevant as the implementation is not trivially parallel. CG is limited by three inter-process communication two of which are global and the third requires communication with its neighbours.

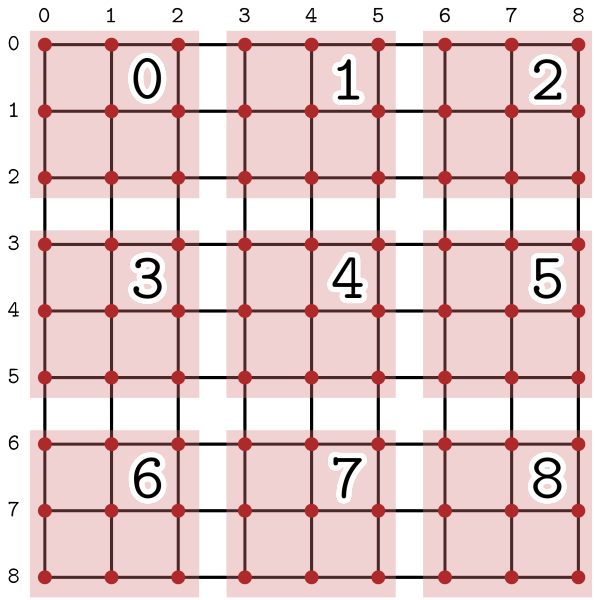

Figure 1: Mesh and process layout

The stencil application in step 4 requires the most neighbouring rows and columns of the mesh in all four directions, with the exception of processes lay on the edge of the mesh which has a known values. As a result between two and four communications are needed for this step.

The scalars τ, β both require two inner products performed on the entire data set. It is thus not possible to avoid this step, without losing accuracy or rate of convergence. To reduce the time spent communicating globally and to reduce the data size, the inner products are calculated first on local data reducing the data size to `sizeof(double)` $\times p$. The local data is then gathered, summed and broadcast to all processes using `MPI_Allreduce` with the sum operation. By this, all processes have received the result of the inner product as if it would have been calculated on all data together. After that the last bit of calculation is done on each process to receive τ, β.

Since these to barriers which cause the processes to sync, it is suitable to have as little as possible between the two points as the closer they are the more synced they will be. Because of this step 6 in the algorithm is moved between step 9 and 10. This does not impact the result but reduces the amount of computation between the global barriers.

Many of the loops used in this program are quite simple, performing simple operations on many elements in an array. Due to this, it is possible to take advantage of the compilers ability to auto-vectorize these operations which enables a single process to perform a set of these

operations in parallel local in the process. This is contributes to a faster serial performance of the implementation.

3.1 Load balancing

As the mesh is divided in to equal squares containing the same amount of elements the load on the processors should be very similar. There is however some difference between the corner, side and middle processes of the grid as slightly reduces the number of calculations that has to be handled differently.

4 Experiments

To measure the performance of this implementation the parallel segment of the implementation was timed. The generation of the data and collection of it, is interesting but it is not relevant to the actual algorithm. How these part are implemented can vary depending on the problem at hand. For the scaling tests the process with the longest execution time is used in this data collection, but for the load balance test another approach has to be taken as the global barriers will conflict with this measurement. Every time measurement, independent of test, was done on 200 iterations on UPPMAX Snowy.

4.1 Scaling tests

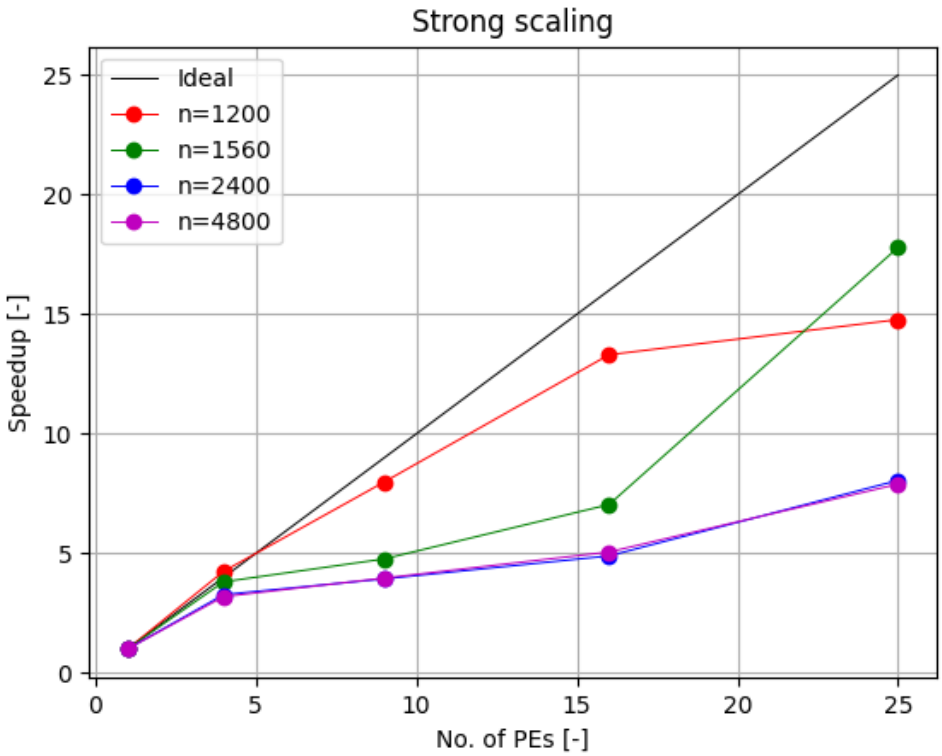

Figure 2: Strong scaling of CG implementations with multiple n

Table 2: Data of the strong scaling test, g-norm ($\sqrt{\mathbf{g}^T\mathbf{g}}$) and execution time of parallel section

n	1200		1560		2400		4800	
PE [−]	g [−]	Time [s]	g [−]	Time [s]	g [−]	Time [s]	g [−]	Time [s]
1	0.007618	1.892	0.007972	3.401	0.008285	7.646	0.007690	31.093
4	0.007618	0.445	0.007972	0.892	0.008285	2.333	0.007690	9.733
9	0.007618	0.237	0.007972	0.713	0.008285	1.942	0.007690	7.833
16	0.007618	0.142	0.007972	0.483	0.008285	1.561	0.007690	6.143
25	0.007618	0.128	0.007972	0.191	0.008285	0.949	0.007690	3.936

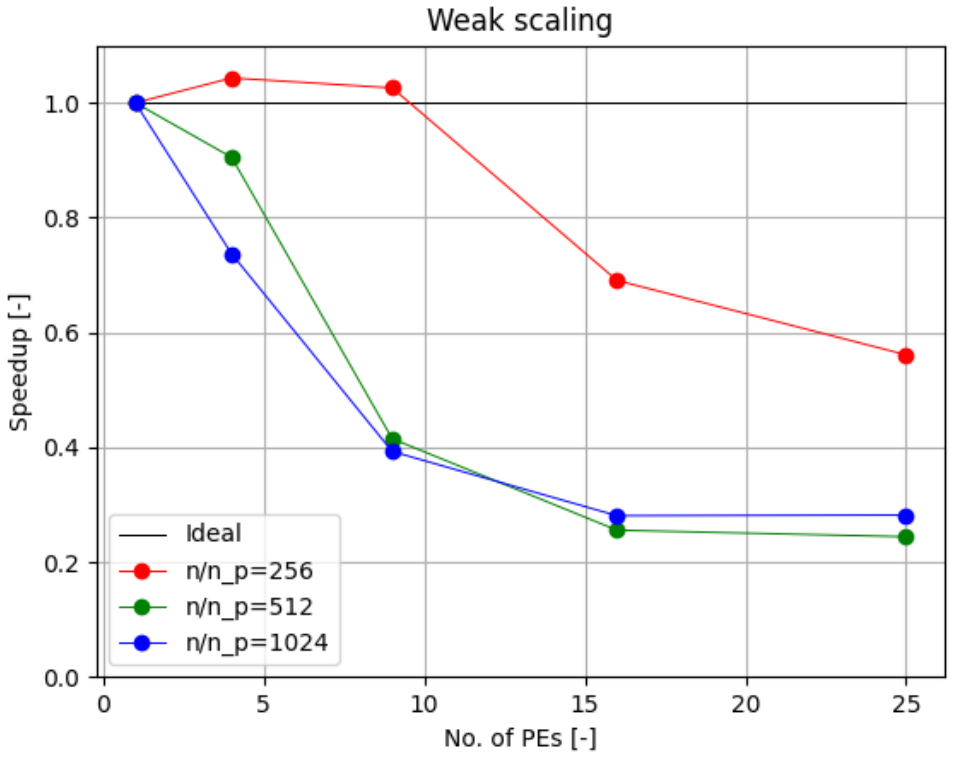

Figure 3: Weak scaling of CG implementations with multiple n/n_p, (elements per side / processes per side)

Table 3: Weak scaling with 256×256 mesh per process

p (n)	1 (256)	4 (512)	9 (768)	16 (1024)	25 (1536)
g-norm [−]	0.000067	0.004465	0.006442	0.007309	0.007719
Time [s]	0.0.079	0.076	0.077	0.114	0.141

Table 4: Weak scaling with 512×512 mesh per process

p (n)	1 (512)	4 (1024)	9 (1536)	16 (2048)	25 (2560)
g-norm [−]	0.004465	0.007309	0.007955	0.008219	0.008287
Time [s]	0.284	0.314	0.686	1.114	1.166

Table 5: Weak scaling with 1024×1024 mesh per process

p (n)	1 (1024)	4 (2048)	9 (3072)	16 (4096)	25 (5120)
g-norm $[-]$	0.007309	0.008219	0.008210	0.007899	0.007605
Time $[s]$	1.262	1.713	3.215	4.497	4.479

4.2 Allinea MAP

To try to identify factors which could cause inefficiencies for more processes and general bottlenecks of the program, several profiling tests in Allinea MAP were performed. A note for this is that different versions of `gcc` and `openmpi` had to be used due to compatibility compared to the rest of the test.

Table 6: Shares of execution time for inter-process communication

Configuration	`MPI_Allreduce` [%]	`MPI_Recv` [%]	Execution time $[s]$
$n = 1200,\ p = 4$	25	13	0.445
$n = 2400,\ p = 4$	40	15	2.333
$n = 1200,\ p = 9$	75	5	0.237
$n = 2400,\ p = 9$	75	15	1.942
$n = 1200,\ p = 16$	75	8	0.142
$n = 2400,\ p = 16$	75	17	1.561

Not only was this performed on parallel runs but also running on a single process, $n = 2400$, to identify serial bottlenecks. There were two lines which contributed a lot to the execution time. Step 6 in the algorithm took 29.3% of the time, of which 92.8% where spent on memory access. As the second biggest entry was applying the stencil to the centre elements, as a part of step 4, which took 27.6% of which 48.7% was spent on memory access, 36.7% and 15.2% on vector floating point and scalar integer respectively.

4.3 Load balancing

In the previous section regarding load balancing, aspects of the corners, sides and middle processes were mentioned as factor in potential unbalanced loads. Time measurements were taken on those sections that are depended on the process' placement on the mesh which corresponds to step 4 in the algorithm. Two tests where performed with 16 processes on $n = 1200, n = 2400$ and the time spent on step 4 over 200 applications can be seen in table

Table 7: Summed time of computation at step 4 for ranks and types corner (C), side (S) and middle (M). The four slowest for each n are marked in **bold**

PE (type)	0 (C)	1 (S)	2 (S)	3 (C)	4 (S)	5 (M)	6 (M)	7 (S)
$n = 1200$	**0.051**	0.037	0.034	0.030	**0.053**	0.038	0.034	0.030
$n = 4800$	1.397	1.711	**2.127**	1.691	1.411	1.726	**2.140**	1.692
PE (type)	8 (S)	9 (M)	10 (M)	11(S)	12 (C)	13 (S)	14 (S)	15 (C)
$n = 1200$	**0.052**	0.039	0.036	0.031	**0.052**	0.036	0.031	0.029
$n = 4800$	1.411	1.727	**2.128**	1.700	1.363	**1.780**	1.643	1.192

5 Conclusion

5.1 Analysis of results

Regarding the results from the strong scaling test, it is for $n = 1200$ performing quite well when increasing the number of processes. It is until 25 PEs it is a drastic difference. By this point each process has a 240×240 square of the mesh at which the actual stencil application takes up a small amount of time in relation to the communication with the other processes. However as n is increased a clear change in performance is seen. As both $n = 2400$ and $n = 4800$ are performing almost the same, some limit has been reached and limits the performance. As seen in the profiling, the share of `MPI_Recv` does increase with n, which is not surprising as larger messages are being sent and thus will require more time to receive. The tests of strong scaling showcases two limits, one upper in the case of $n = 1200$ and a lower for $n = 2400, n = 4800$. The effect of the upper bound is lessened when $n = 1536$, but the effect of the lower bound is clear as for $p = 9,\ 16$ the strong scaling is not very good.

This indicates that there has to be a balance between the time spent on communication and the time spent on pure computing to receive a uniform scaling. Which can be seen in the weak scaling tests as for $n/n_p = 256$ it scales well to begin with but as the amount of communication is increased, the efficiency drops.

There is not a clear pattern to this profiling that is inline with the test results. One thing to consider about the `MPI_Allreduce` is that it is synchronous. This is necessary and wanted, but it might explain the significant times spent on these lines. While a process is waiting for processes to catch up it contributes to the time spent on the allreduce. Especially when the size of data handled by the command grows with 8 bytes per process, which is insignificant. This is strengthened by the result of the tests of load balance, where some processes are significantly slower than other at almost the double execution time. As the receive calls still have some significance in the profiling it is clear that the non-blocking sends are a good approach to avoid blocking unrelated computations.

The load balance test had a surprising result as it was not expected to be such large differences between the processes. Further, the difference in time does not seem to be associated with the placement on the mesh, which was discussed previously. This was also tested with $p = 9$ but no pattern could be established there either.

5.2 Areas of improvements

The cause of the differing results are somewhat hard to identify and further establish clear areas of improvements. However because of the serial profiling that was done, it has been established that memory access is a limiting factor. A smaller improvement could be to choose to stack allocate some parts of the data, for example the receiving data. This could possibly speed up the receiving as well as the computations of those segments. These are more likely to fit onto the stack than the center elements. But as mentioned the most time consuming steps 6 and 4, the problem with step 6 is that it is the only time $\mathbf{u}$ is used and thus has to load in. It is possible that allocating this vector to the stack, compared to the heap, would use the stack memory better than the previous proposal since it is time consuming and a critical part as it is the solution of the algorithm. With out this, CG does not produce any usable results. In the case of step 4, it is expected to take a lot of time as it one of the bigger loop with a difficult access pattern to optimize for. There was attempts at splitting the loop in to several simpler loops but no performance increase was seen. This probably reduced the effect of memory access, but outweighed by the additional passes of the elements cause by the loops Instead one could consider implementing some sort of cache blocking, such that instead of computing based on the rows, computing by squares thus utilizing the above and below elements in the stencil better

as they do not have to be unloaded and loaded several times. This would ideally reduce the problem of memory access greatly.

As a general improvement for this entire implementation, which is largely based on vector and optionally matrix operations, one could consider using a library such as BLAS [1]. BLAS is very optimized for linear algebra and even though many of this implementation's inner products are vectorized, this library would probably perform better. This could also be applied stencil application as well

An inherent hurdle of the conjugate gradient algorithm is that it is limited by global barrier. Whilst it is possible to reduce the size of these, and reduce the size of data being sent it is hard to avoid all of them. There are however variant of the algorithm, as mentioned in [2] and [3] that try to reduce these.

To conclude this report, the implemented solutions does achieve the desired goals to a degree. For the cases of $n \leq 1200$ the implementation performs well. But there are defiantly improvements to be made as mentioned above to have better scaling even for larger problems. As further work, more investigation should be done to better identify the causes of the unexpected result in the load balance tests. This a long the suggested improvements would strengthen this implementation and the performance of it.

References

[1] *BLAS (Basic Linear Algebra Subprograms)*. 2022. URL: https://www.netlib.org/blas/.

[2] J. Rantakokko H. Löf. "Algorithmic optimizations of a conjugate gradient solver on shared memory architectures". In: *International Journal of Parallel, Emergent and Distributed Systems* 21 (2006), pp. 345–363.

[3] Dianne P O'Leary. "Parallel implementation of the block conjugate gradient algorithm". In: *Parallel Computing* 5.1 (1987). Proceedings of the International Conference on Vector and Parallel Computing-Issues in Applied Research and Development, pp. 127–139. ISSN: 0167-8191. DOI: https://doi.org/10.1016/0167-8191(87)90013-5. URL: https://www.sciencedirect.com/science/article/pii/0167819187900135.