Parallel Random Linear System and Conjugate Gradient method using MPI

Parallel and Distributed Programming

Max Reeves

May 25, 2017

1 Introduction

The *conjugate gradient method* (CG) [1] is an algorithm for numerical solving of systems of linear equations, as in (1), where the $N \times N$ matrix A is symmetric and positive definite (SPD). Symmetric meaning that $A = A^T$ and positive definite that if, for every non-zero vector $\mathbf{z}$, the scalar $\mathbf{z}^T A\mathbf{z} > 0$. It is an iterative method, meaning that it starts off with an initial guess of the solution vector $\mathbf{x}$ and iteratively improves the guess.

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

A parallel implementation of the CG method has the potential to consume a lot of memory when N is large. Especially if A is dense and a sparse matrix representation would not be helpful in reducing memory usage. The same goes for generating random test data that can be solved by the CG method, considering that the matrix A must be SPD.

This project evaluates the speed and memory usage when generating pseudorandom linear systems and solving them using the CG method in a parallel MPI implementation. The approach uses 1D partition with block algorithms.

2 Problem description

The problem is divided into two parts: generating the random linear system and solving it using the CG method.

2.1 Generating the linear system

One way to create a pseudorandom SPD matrix A, where A has dimensions $N \times N$, is to first fill it with random numbers in the range $(0, 1)$ and then calculate $A := 0.5(A^T + A) + NI$. The addition $(A^T + A)$ ensures that the matrix is symmetric, while multiplying the result with 0.5 and adding NI, where I is the identity matrix, ensures that it is positive definite.

Once A is positive definite, a random solution vector $\mathbf{x}$ can be generated and the r.h.s. vector $\mathbf{b}$ produced by calculating $\mathbf{b} := A\mathbf{x}$. The result is a complete linear system solvable with the CG method, whose approximate solution $\mathbf{x}^*$ can be verified by comparison to $\mathbf{x}$.

The most interesting part of this problem is how to parallelize the calculation of the final A matrix in a fast and memory efficient way. The serial implementation for this procedure is described in Algorithm 1.

```
   Data: Desired matrix size $N$.
   Result: Pseudorandom $N \times N$ SPD matrix $A$.
 1 $A$ := new $N \times N$ matrix
 2 for $i$ := 0 ... $N-1$ do
 3     for $j$ := 0 ... $N-1$ do
 4         $A[i][j]$ := random(0, 1)
 5     end
 6 end
 7 for $i$ := 0 ... $N-1$ do
 8     for $j$ := 0 ... $N-1$ do
 9         if $i > j$ then
10             $A[i][j] := A[j][i]$
11         else
12             $A[i][j] := 0.5 \cdot (A[i][j] + A[j][i])$
13         end
14         if $i = j$ then
15             $A[i][j] := A[i][j] + N$
16         end
17     end
18 end
```

Algorithm 1: Pseudorandom SPD matrix generation.

2.2 The conjugate gradient method

The serial implementation of the CG method is described in Algorithm 2 below. In theory the solution will converge after N steps, why it is common to set the maximum number of steps, *max_steps*, to N. Apart from that, it will iterate until the given tolerance ε is reached.

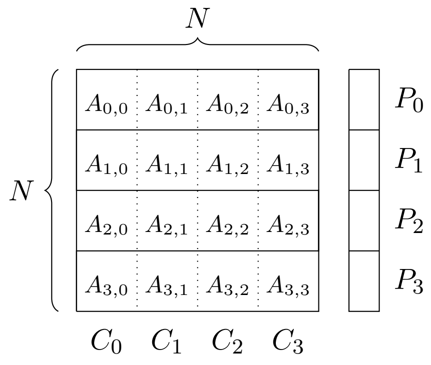
The most time consuming step of this algorithm is the matrix-vector multiplication $A_p := A\mathbf{p}$ in line 6. One more thing worth to notice is that in a parallel implementation, all processes will have to stop and gather the results in lines 4, 7 and 10 before being able to proceed on their own. Other than that, most of the work can be done in parallel without interprocess communication.

```
Data: SPD matrix $A$, vector $\mathbf{b}$, maximum number of steps
      $max\_steps$, and tolerance $\varepsilon$.
Result: An approximate solution $\mathbf{x}^*$ to the linear system $A\mathbf{x} = \mathbf{b}$.
$\mathbf{x}^* :=$ new $N$ vector $[0 \ldots 0]$
$\mathbf{r} := \mathbf{b} - A\mathbf{x}^*$
$\mathbf{p} := \mathbf{r}$
$\gamma := \mathbf{r}^T\mathbf{r}$
for $i := 0 \ldots max\_steps$ do
    $A_p := A\mathbf{p}$
    $\alpha := \gamma/(\mathbf{p}^T A_p)$
    $\mathbf{x}^* := \mathbf{x}^* + \alpha\mathbf{p}$
    $\mathbf{r} := \mathbf{r} - \alpha A_p$
    $\gamma_{new} := \mathbf{r}^T\mathbf{r}$
    if $\sqrt{\gamma_{new}} < \varepsilon$ then
        break
    end
    $\beta := \gamma_{new}/\gamma$
    $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$
    $\gamma := \gamma_{new}$
end
```

Algorithm 2: The conjugate gradient method.

3 Solution method

The implementation uses a 1D topology as described in in Figure 1. Each of p number of processes gets a row of the A matrix with dimensions $N \times q$, where $q = \lfloor N/p \rfloor$, or $q = \lfloor N/p \rfloor + 1$ if $i < (N \mod p)$ for P_i. The division ensures that the sizes are reasonably close to each other and that the dimensions of block $A_{i,j}$ is equal to that of $A^T_{j,i}$.



N
N
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
P_0
P_1
P_2
P_3
C_0 C_1 C_2 C_3

Figure 1: Partitioning strategy. The matrix is divided into even sized rows, one for each process P_i, the rows are then divided into "virtual" blocks $A_{i,j}$ such that the number of blocks in each row is equal to the number of rows. Each vector in the implementation is divided into rows in a similar way.

3.1 Parallel pseudorandom SPD matrix generation

The core of the parallel implementation of the pseudorandom linear system generation is the generation of the generation of the SPD matrix A, as described in Algorithm 1. In order to save memory, the calculations are done "in place". This means that each process only stores its local part of the matrix plus a (roughly) $q \times q$ sized buffer during this step.

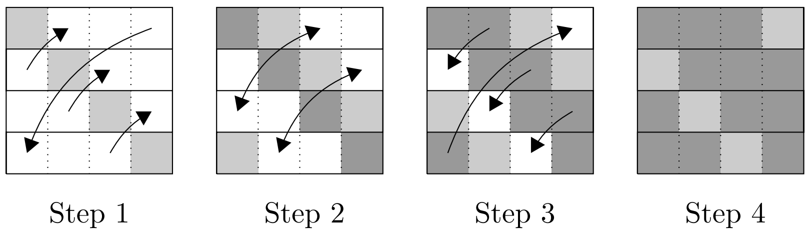


Figure 2: Illustrating the communication pattern during each step in the calculation of $0.5(A^T + A) + NI$, as implemented in the parallel version of the linear system generating algorithm. The $N \times N$ matrix is distributed over a 1D partition, into four rows, but treated as if it was divided into blocks. Light gray blocks are being processed in the current step while the dark gray blocks have been finished.

The algorithm starts off by assigning random numbers to each matrix element, in parallel, after which $0.5(A^T + A) + NI$ is calculated. The algorithm is described in Figure 2 and is requires a total of p steps. Blocks are sent transposed using a special MPI_Datatype. In the first step, each process calculates the diagonal blocks of its row using local data. In the following steps, if a process P_i is going to calculate using its local block $A_{i,j}$ it must have received block $A_{j,i}$ from process P_j in the previous step. If the received block has already been calculated, then it will simply be copied.

Calculations can be performed while blocks are being sent between the processes, using `MIP_ISend` and `MIP_Recv`, apart from rare cases where the processes have to send the block that it has to calculate. (The latter will happen in at most one step during the total calculation for certain values of p.)

The generation of $\mathbf{x}$ is done by simply assigning random numbers in the range $(0, 1)$. The calculation of $\mathbf{b}$ uses the same matrix-vector multiplication procedure as described the following section.

3.2 Parallel CG method

The most essential part of the parallel CG method is to optimize the matrix-vector multiplication, as most of the other calculations can be done in parallel without interprocess communication. The sharing of the variables γ, α and γ_{new} among the processes is difficult to do anything about in a parallel message passing environment, so for that `MPI_Allreduce` is used.

The parallel matrix-vector multiplication is implemented such that in the first step the local part of the vector is used to calculate the diagonal block of the row, and in each consecutive step the vector is shifted one circular step upwards. The vector parts are sent during calculation, using `MIP_ISend` and `MIP_Recv`. This only requires a buffer of size q. The execution of this algorithm is illustrated in Figure 3.

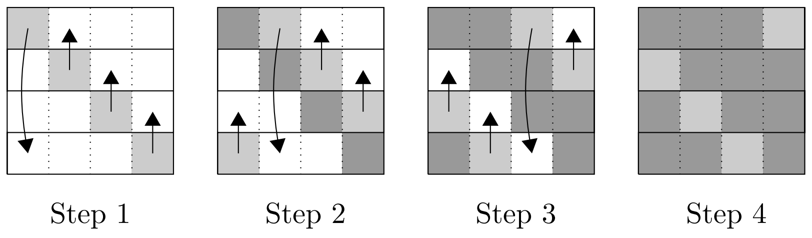
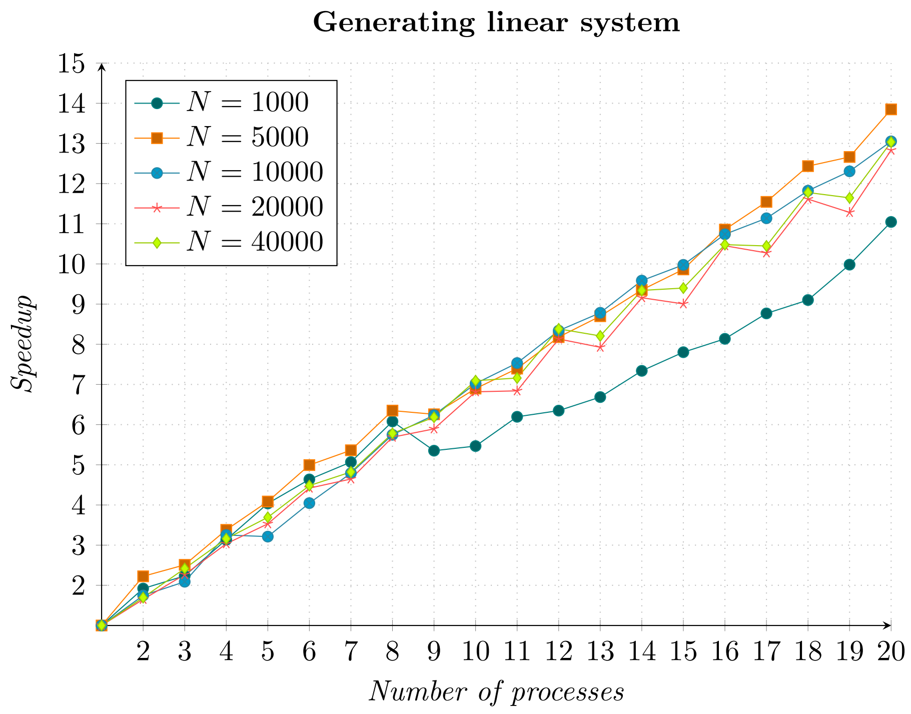


Figure 3: Illustrating the message passing in calculating a matrix-vector multiplication, where the vector is being shifted circularly one step upwards.

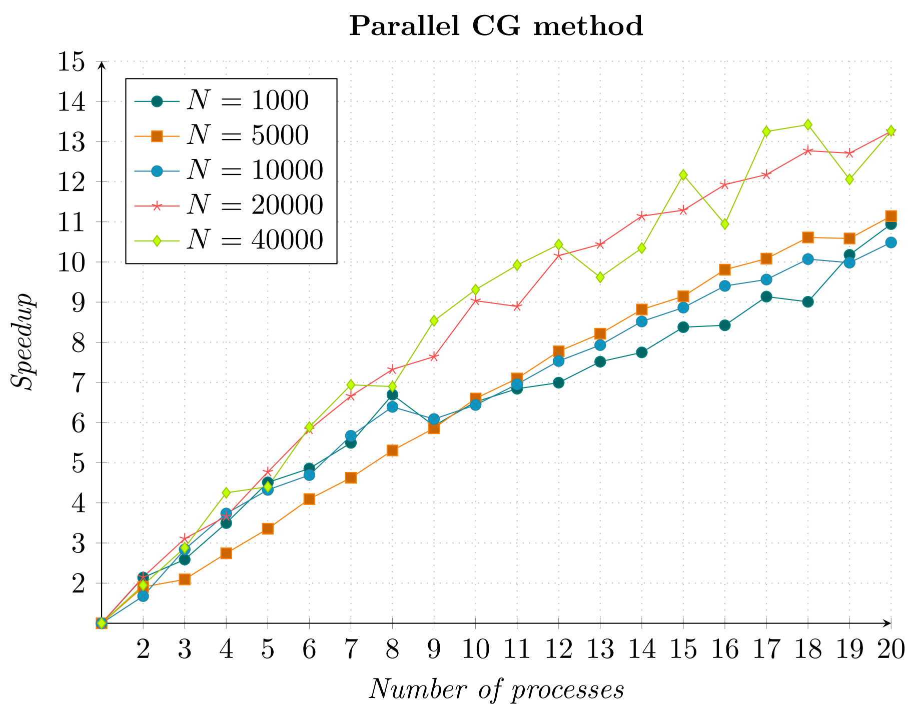
4 Experiments

The following experiments were conducted using a tolerance of 10 decimal digits, that is $\varepsilon = 1/(10 + 1)$, and N as maximum number of iterations in the CG method. The speedup has been measured for both the linear system generation and the CG method for up to 20 processes, which is the maximum number of processes in a node in the Rackham cluster.

Each run allocated a full node with no other processes running. Five runs per number of processes and problem size has been benchmarked, of which the minimum is used in the presentation of the plots and tables below. Speedup of a specific problem size S_p is measured as $S_p = T_1/T_p$, where p is the number of processes used and T_p the minimum time when running with p processes.



The data generation seems to scale well with the problem size. However, for the smallest problem size of $N = 1000$ there is a dip for more than eight processes.



The parallel CG method seems to perform much better when the problem sizes are very large. The scaling does not seem to be completely linear, however. Benchmarks are reported in the tables below.

Table 1: Generating linear system benchmarks.

$N\backslash p$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1000	0.040	0.020	0.017	0.012	0.009	0.008	0.007	0.006	0.007	0.006
5000	0.923	0.415	0.367	0.273	0.225	0.184	0.172	0.145	0.147	0.124
10000	3.297	1.888	1.578	1.013	1.026	0.814	0.688	0.573	0.528	0.437
20000	13.89	8.423	6.129	4.579	3.936	3.144	2.990	2.443	2.357	2.031
30000	31.26	18.94	13.52	10.52	8.873	7.306	6.794	5.626	5.333	4.609
40000	58.40	34.46	24.15	18.48	15.81	13.03	12.11	10.10	9.444	8.157

$N\backslash p$	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
1000	0.006	0.006	0.005	0.005	0.005	0.004	0.004	0.004	0.004	0.003
5000	0.124	0.112	0.106	0.098	0.093	0.085	0.079	0.074	0.072	0.066
10000	0.437	0.395	0.375	0.343	0.330	0.307	0.296	0.278	0.267	0.252
20000	2.031	1.708	1.753	1.517	1.543	1.329	1.352	1.196	1.231	1.082
30000	4.609	3.834	3.936	3.446	3.453	3.049	3.103	2.707	2.781	2.448
40000	8.157	6.966	7.116	6.252	6.213	5.572	5.590	4.958	5.015	4.481

Table 2: Parallel CG method benchmarks.

$N\backslash p$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
1000	0.012	0.005	0.004	0.003	0.002	0.002	0.002	0.001	0.002	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
5000	0.207	0.109	0.099	0.075	0.061	0.050	0.044	0.039	0.035	0.029	0.029	0.026	0.025	0.023	0.022	0.021	0.020	0.019	0.019	0.018
10000	0.717	0.427	0.252	0.192	0.165	0.152	0.126	0.112	0.117	0.103	0.103	0.095	0.090	0.084	0.080	0.076	0.074	0.071	0.071	0.068
20000	3.349	1.555	1.077	0.914	0.702	0.575	0.502	0.457	0.438	0.376	0.376	0.329	0.320	0.300	0.296	0.280	0.275	0.262	0.263	0.252
30000	7.542	3.851	2.345	1.742	1.512	1.399	1.140	1.156	1.078	0.859	0.859	0.882	0.742	0.716	0.694	0.669	0.683	0.568	0.553	0.617
40000	13.47	6.916	4.676	3.170	3.062	2.289	1.942	1.954	1.579	1.358	1.358	1.291	1.401	1.302	1.107	1.231	1.017	1.004	1.117	1.016

5 Conclusions

The dip for problem size of $N = 1000$ in the linear system generation could be because of the overhead of message passing, which becomes more apparent when the problem size is small and the number of processes is large.

The parallel CG method seems to perform much better when the problem sizes are very large, which is interesting. This could be because messages passed are small, and since it can be done in parallel with the fairly time consuming calculations, they are barely noticeable. The reason why scaling does not seem to be completely linear could be because of the need to synchronize when calling `MPI_Allreduce`. When the problem sizes are large, there might be a greater chance that they become more unsynchronized and require larger waiting times. Another reason could be difference in convergence speed for the numerical method used.

In the implementation, a special `MPI_Datatype` was used to send blocks transposed. Whether how this affects the performance is unknown, but it would be interesting to know.

In the implementation of the parallel CG method, message passing, including a buffer, and `MPI_Allgather` is used even when running a single process. This could also be improved.

One of the reasons a 1D partitioning strategy was used was to save memory, although it is not a huge amount for the CG method. Another reason was that in a 2D implementation there will either be a lot of duplicate work or idle process time. However, it would be interesting to see how the two partitioning strategies would perform against each other.

It would also be interesting to implement a version for sparse matrices, using a sparse matrix representation, and compare the two.

References

[1] Jonathan Richard Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, edition 1-1/4*. Technical report, School of Computer Science, Carnegie Mellon University, August 1994. `http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`