UPPSALA
UNIVERSITET

Project

PARALLEL AND DISTRIBUTED PROGRAMMING 5 CREDITS 1TD070 62007

Group members:

Ibrohim HAMOUD ibrohimmn1994@gmail.com

UPPSALA

March 22, 2022

1 Introduction and Problem description

Conjugate gradient (CG) solver is an iterative method for solving large sparse systems of linear equations $\mathbb{A}\mathbf{u} = \mathbf{b}$, and it is a useful tool in approximating solutions to linearized partial differential equations. A full description of the method's steps is given in Algorithm 1. Direct computation of $\mathbb{A}^{-1}$ is time-consuming and memory-demanding, especially for high dimensions. therefore, iterative methods are deployed and which can overcome these two drawbacks of the direct method[1].

In the scope of this project, two-dimensional mesh od size $n \times n$ is considered as a backward sitting for the CG-application. The coordinates are given by $x_i = ih$, $y_i = ih$, $i, j = 1, 2, ...n$ and $h = \frac{1}{n-1}$, where h is the size of the interval and n is the mesh-size. Thus, vectors $\mathbf{u}$ and $\mathbf{b}$ and the matrix $\mathbb{A}$ are of dimension $N = n^2$. The entries of the vector $\mathbf{b}$ are given by

$$b_{ij} = 2h^2(x_i(1 - x_i) + y_j(1 - y_j)), \quad i, j = 1, ..., n.$$

matrix is symmetric and positive definite given by five-point stencil in the form:

$$\underbrace{\begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & \dots \\ -1 & 4 & -1 & 0 & -1 & 0 & \dots \\ 0 & -1 & 4 & 0 & 0 & -1 & \dots \\ -1 & 0 & 0 & 4 & -1 & 0 & \dots \\ 0 & -1 & 0 & -1 & 4 & -1 & \dots \\ 0 & 0 & -1 & 0 & -1 & 4 & \dots \\ & \ddots & & \vdots & & \ddots & \end{pmatrix}}_{n \times n, n=3}.$$

A vector multiplication with the matrix $\mathbb{A}$ can be performed using a one-dimensional stencil application where every element in the vector is multiplied by 4 and then subtract from the value of its neighbor above, neighbor below, the element that is of distance n in the above direction and the element that is of distance n below.

Algorithm 1 Conjugate gradient algorithm.

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

2 Conjugate gradient parallel algorithm

The initial attempt to parallelize the CG-algorithm involved applying matrix-vector multiplication locally in each processor (Algorithm 2). The matrix A is sparse and thus it was stored in COO-format with the multiplication procedure performed as well in COO-format[2]. this parallelization method requires every processor to possess all elements of $\mathbf{b}$. Therefore, three communication lines were required between the processors that are MPI_Allgather() to get collect all elements of $\mathbf{b}$ in every processor, two MPI_Allreduce() to get the values of τ and β in every processor. All the other steps are performed locally. Data initialization with a single processor by assembling $\mathbf{b}$ and $\mathbb{A}$ locally. The vector $\mathbf{b}$ is divided into equal chunks and distributed equally among the processors with MPI_Scatter(), while the matrix $\mathbb{A}$ is divided row-wise. Thereafter, every chunk of $\mathbb{A}$ is converted into COO-format locally in each processor.

Alternative parallelization algorithm involves using stencil application on the $\mathbf{d}$ instead of matrix-vector multiplication (Algorithm 3). Thus, the function MPI_Allgather() that assembles $\mathbf{d}$ can be skipped and thereby reduces the communication links into two. A major advantage of this method over the previous one is to overstep data preparation represented by computing $\mathbf{b}$ and assembling $\mathbb{A}$ which proved to be a big overhead and a serious obstacle. Take for example a mesh size of 500×500 that gives 25×10^4 points and hence matrix $\mathbb{A}$ of size $(625 \times 10^8) \times (625 \times 10^8)$. Here, The assembly of such matrix and later on distributing it and thereafter converting into COO-format amount for a considerable work. A possible workaround would be to assemble the matrix locally and directly into COO format. This way, both methods would appear similar as both matrix-vector multiplication in COO-format and stencil application are of the same complexity $\mathcal{O}(n)$. Nevertheless, stencil application provides a fully parallelized algorithm as the local $\mathbf{d}$'s do not need to be collected in each iteration step.

To enable stencil application locally, elements of the local vector $\mathbf{d}$ that resides in other processors need to be collected. This is circumvented by letting each processor to allocate their respective local $\mathbf{d}$ in a shared memory space. For that purpose, the Mpi_win_allocate_shared() provides a collective call executed by all processes, such that, it allocates memory of specific size that is shared among all processes, and returns a pointer to allocated segment that can be used for storing the local $\mathbf{d}$. This locally allocated memory can be accessed by a different processor using MPI_Win_shared_query() function. Here, non-contiguous memory allocation is used as there is no possibility of calculating the address to other processes memory in the non-contiguous model. Instead, to acquire the process adjacent memory space, the MPI_Win_shared_query is used with the higher and lower processors' ranks that hold the adjacent boundary rows. Thusm obtaining the pointers from the processor that has the window position lower and higher than current processor[3].

The logical architecture of Algorithm 2 involves equally dividing the mesh points along the row-line and distributing them among the processors. For five-point stencil, every processor has only two adjacent processors that controls either the rows above or the rows below. this architecture implies that applying the five-points stencil on the boundary rows in every processors requires accessing the boundary rows in the adjacent processor. Therefore, every processors provided by two ghost rows where the boundary rows from the adjacent processors are stored. All of which occurs in the shared memory space.

The above-mentioned architecture bears a downside that is the number of deployed processors can not be smaller than the number of rows which is big a constraint on the parallelization for a very large mesh. A superior algorithm can be suggested by dividing the mesh point following checkerboard architecture with each processor required to access the information in four surrounding processors. Such architectures will enable parallelization of the code sown into one mesh-point per processor. However, the row-wise divide was considered in this work solely for the reason that it is easier and faster to implement.

All processors have equal load balance as the number of arithmetic operations is exactly the same even for the processors that hold the two boundary rows in the mesh. if the mesh points are rearranged row-wise as a long array of points and each processor holds a chunk of it plus ghost points from the adjacent processors. The rightmost processor will have ghost points to its right that holds value 0 for the elements of **d**, similarly for the leftmost processor. As a result stencil application will in all processors comprise of the same workload

Algorithm 2 Conjugate gradient with matrix-vector multiplication.

```
1:  Data preparation
    - Assemble $\mathbb{A}$, and $\mathbf{b}$ locally on on processor and distribute them with MPI_Scatter().
    - Locally convert the distributed parts of $\mathbb{A}$ onto COO-format.
2:  Initialize locally $\mathbf{u} = 0 \quad \mathbf{g} = -\mathbf{b} \quad \mathbf{d} = \mathbf{b}$
3:  Locally $q_0 = \mathbf{g}^T\mathbf{g}$
    - Get the global $q_0$ with MPI_Allreduce().
4:  for $it$ = 1, 2,... until convergance do
5:      $\mathbf{q} = \mathbb{A}\mathbf{d}$
        - Get the global $\mathbf{d}$ on all processors with MPI_Allgather().
        - Locally perform matrix-vector multiplication in COO-format.
6:      $\tau = q_0/(\mathbf{d}^T\mathbf{q})$
        - Compute $(\mathbf{d}^T\mathbf{q})$ locally.
        - Sum all the local values into a global with MPI_Allreduce().
        - locally global $\tau$ locally on all processors.
7:      $\mathbf{u} = \mathbf{u} + \tau\mathbf{d}$
        - Computed locally.
8:      $\mathbf{g} = \mathbf{g} + \tau\mathbf{g}$
        - Computed locally.
9:      $q_1 = \mathbf{g}^T\mathbf{g}$
        - Computed locally.
        - Sum up on all processors with MPI_Allreduce().
10:     $\beta = q_1/q_0$
        - Computed locally.
11:     $\mathbf{d} = -\mathbf{g} + \beta\mathbf{d}$
        -Computed locally.
12:     $q_0 = q_1$
        - Executed locally.
13: end for
```

Algorithm 3 Conjugate gradient using stencil application.

```
 1: Data preparation
    - Let every processor generate a chunk of $\mathbf{b}$ locally and store it in a shared memory accessible
      by other processors with the following steps,
        - MPI_Win to open a shared memory window.
        - MPI_Win_allocate_shared to allocate the local $\mathbf{b}$.
        - Generate and Store local portion of $\mathbf{b}$.
        - MPI_Win_shared_query() to get the address of the adjacent parts of $\mathbf{b}$ in the shared
          memory that was generated by the other processors.
        - later encapsulate the conjugate gradient iteration inside MPI_Win_lock_all() and
          MPI_Win_unlock_all() to enable shared memory access.
 2: Initialize locally $\mathbf{u} = 0 \quad \mathbf{g} = -\mathbf{b} \quad \mathbf{d} = \mathbf{b}$
 3: Locally $q_0 = \mathbf{g}^T\mathbf{g}$
    - Get the global $q_0$ with MPI_Allreduce().
 4: for $it$ = 1, 2,... until convergence do
 5:     $\mathbf{q} = \mathbb{A}\mathbf{d}$
        - Deploy stencil application for that purpose enabled by shared memory access.
 6:     $\tau = q_0/(\mathbf{d}^T\mathbf{q})$
        - Compute $(\mathbf{d}^T\mathbf{q})$ locally.
        - Sum all the local values into a global with MPI_Allreduce().
        - locally global $\tau$ locally on all processors.
 7:     $\mathbf{u} = \mathbf{u} + \tau\mathbf{d}$
        - Computed locally.
 8:     $\mathbf{g} = \mathbf{g} + \tau\mathbf{g}$
        - Computed locally.
 9:     $q_1 = \mathbf{g}^T\mathbf{g}$
        - Computed locally.
        - Sum up on all processors with MPI_Allreduce().
10:     $\beta = q_1/q_0$
        - Computed locally.
11:     $\mathbf{d} = -\mathbf{g} + \beta\mathbf{d}$
        -Computed locally.
12:     $q_0 = q_1$
        - Executed locally.
13: end for
```

3 Results and discussion

Parallel scalability is often visually represented by plots of observed speedup. The ideal behavior of code for a fixed problem size N using p parallel processes is that it is p times as fast as serial code. If $T_p(N)$ denotes the wall clock time for a problem of a fixed size parameterized by N using p processes, then the quantity $\frac{T_1(N)}{T_p(N)}$ represent the speedup. Figure 1 shows the plot of a speedup as a function of the number of processors used for a mesh of $n = 1024$. Similar plot is given for a smaller problem size of $n = 128$ in figure 2. Comparing both plots, it can be seen that the speedup is better for larger problems since the increased communication time for more parallel processors does not dominate over the calculation time as quickly as it does for small problems.

In both figures 1 and 2, a drop in the speedup is observed as the number of nodes employed exceeds 1. However, it is noticed that the drop is relatively larger for smaller problem sizes. This reflects that the communication of processors between two nodes takes longer than that within one node. Comparing both plots again gives the conclusion that for a small problem size, the speedup is efficient if the number of processors is within the limits of the number of processors/node. Whereas for a large problem size, the drop in speedup is less severe and hence the speedup gain corresponding to the number of processors could potentially be worthy if the problem is very large.

In figure **??**, it is noticed that the speedup increase corresponding to the 16 to 32 processor number's increase is very small. Although it is small, the runtime still does decrease. This behavior can potentially be explained by the limitation in-memory access, as a bottleneck can be expected when more processes on each CPU attempt to access the memory simultaneously than the available memory channels per CPU[4][5].

Figure 3 shows a plot of the problem size as a function of wall clock time. The adjacent table gives the residual calculated fro different mesh dimensions. Here, the residual increases for a larger problem size due to a fixed number of iterations that is 200 throughout this work. The increase in the residual is expected behavior as the initial value of vector **d** entries get smaller and hence the value of β in step 9 in Algorithm 1 gets larger. Thus, the path toward the minima takes a more jiggly path and therefore more iterations are required. However, the low residual value for $n = 1024$ remains difficult to explain.

Plot 3 shows a linear increase in time execution with the size of the problem. Although the number of processors was scaled up proportionally. this is expected behavior as the parallel part scales linearly with the number of resources.

Table 1: Time measurements for a mesh of $n = 1024$ for different processor number.

processors	Time
1	1.471596
2	0.887373
4	0.425559
8	0.179067
16	0.096387
16×2	0.089656

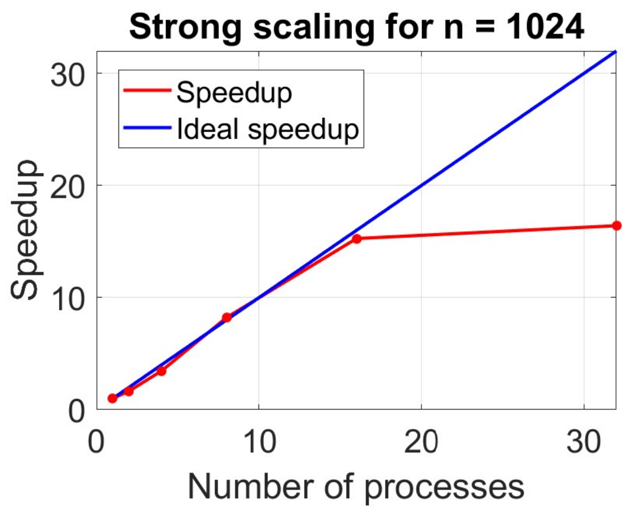

Figure 1: Plot of the speedup as a function of the number of processors for a mesh of $n = 1024$.

Table 2: Time measurements for a mesh of $n = 128$ for different processor number.

processors	Time
1	0.014963
2	0.011608
4	0.006613
8	0.005502
16	0.003154
16×2	0.038747

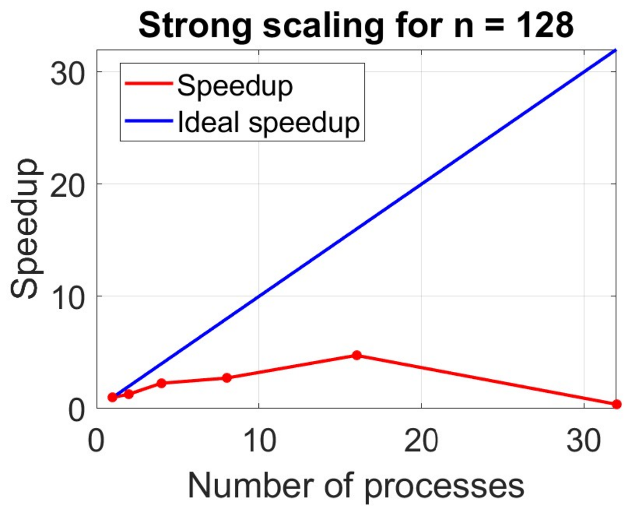

Figure 2: Plot of the speedup as a function of the number of processors for a mesh of $n = 128$.

Table 3: Table of time measurements and residual after 200 iteration for different mesh sizes with the number of processors scaled proportionally.

n	Processors	Time	Residual
64	1	0.003668	1.3684×10^{-8}
128	4	0.006302	6.6298×10^{-7}
256	16	0.008205	2.5401×10^{-5}
512	64	0.073423	1.1083×10^{-4}

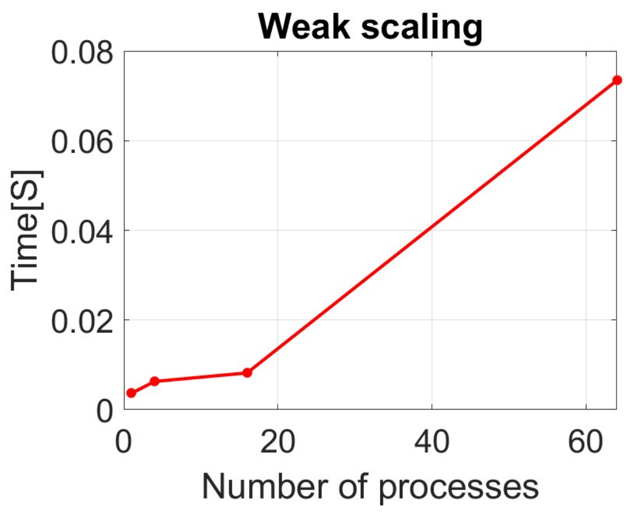

Figure 3: Plot of the execution time as a function of number of processes for a mesh of $n = 1024$.

4 Conclusion

In this project, we developed an MPI-based conjugate gradient method for a two-dimensional mesh. Experimentation on the run-time was performed to realize the weak and strong scalability of the algorithm and analyze the MPI performance. To obtain a good performance in relation to the utilized resources, we need to limit the number of nodes for small problem sizes as well to fever having all processors on the same node instead of on many nodes due to the favorable conditions for inter-node connections.

References

[1] J. Vienne, “Introduction to parallel programming with mpi,” *Introduction to Scientific and Technical Computing*, pp. 171–186, 2016.

[2] D. Lass, *Implementation of some parallel algorithms arising in sparse matrix and other applications.* The University of Alabama, 2017.

[3] T. H. R. T. Pavan Balaji, William Grop, “Advanced mpi programming tutorial,” 2016.

[4] V. D. Nguyen, “Implementation of mpi-based conjugate gradient methodfor solving the poisson equation on cartesian grid,” 2015.

[5] C. Barajas and M. K. Gobbert, “Strong and weak scalability studies for the 2-d poisson equation on the taki 2018 cluster,” *UMBC Student Collection*, 2019.