Final Project: Progress Report 2

CSE 597

Yueze Tan

Monday, November 5, 2018

Abstract

In this progress report, we discuss about the Jacobi solver for Poisson equation with Debye-Huckel's shielding term, along with its parallelization using MPI. The justification of the parallelization method and the profiling of the code are also discussed in this report, after which we show the scaling results for multi-core calculations.

1   Problem of Interest

The problem is related with solving the electrostatic equation (Poisson equation) in a system with free moving charges and shielding effects, typical examples of which would be plasma or electrolytes. From Boltzmann's distribution we can obtain the equation describing the shielded, or compensated field:

$$(\nabla^2 - \lambda_D^{-2})\Phi = -\frac{\rho}{\varepsilon},$$

in which Φ is the electric potential, ρ is the charge density which is contributed by other effects than shielding, like charge density induced by inhomogeneous distribution of polarization in a dielectric matter, and ε is permittivity of the system. λ_D is known as Debye length, which is a term coming from Debye-Huckel's linearization of the electric shielding effect.

For free-moving charge is common in various physical systems, charge compensation and spontaneous shielding effect could be widely observed in different kinds of condensed systems. In order to get a more accurate solution in these systems, solving the electrostatic equation with Debye length term is a necessary step. The distribution of electric potential is an interested topic in various fields, like materials science, condensed matter physics and chemical engineering, where electrolytes are extensively discussed.

Other numerical methods could also be applied to solve this problem, as what could be done for many other different types of PDEs. One possible way is using finite-element method (FEM), in which we discretize the system into piecewise functions, and then transform the problem into a integration form:

$$\int_V \left[(\nabla^2 - \lambda_D^{-2})\Phi + \frac{\rho}{\varepsilon} \right] v_i \; \mathrm{d}V = 0,$$

in which v_i is a set of specially chosen functions to simplify the integrations.

Or we can use fast Fourier transform (FFT) to solve this problem, by transforming the original field into Fourier spectrum and then solve the algebraic equation in the frequency domain:

$$(k_1^2 + k_2^2 + k_3^3 - \lambda_D^{-2})\Phi^K = -\frac{\rho^K}{\varepsilon},$$

and by taking inverse FFT (IFFT) on the spectrum we can get the field in real space. Such a method has already been used to solve the conventional electrostatic problem in phase-field models, see (1).

It might be of the readers' interest that this problem has an analytical solution, which is based on its Green function:

$$\Phi(\mathbf{r}) = \int_V \frac{\rho(\mathbf{r}')(\mathbf{r} - \mathbf{r}')\exp(|\mathbf{r} - \mathbf{r}'|/\lambda_D)}{|\mathbf{r} - \mathbf{r}'|^3} \; \mathrm{d}^3\mathbf{r}' + \oint_{\partial V} \frac{\sigma(\mathbf{r}')(\mathbf{r} - \mathbf{r}')\exp(|\mathbf{r} - \mathbf{r}'|/\lambda_D)}{|\mathbf{r} - \mathbf{r}'|^3} \; \mathrm{d}S.$$

Basically, this solution is a modified version of the regular electrostatic solution for Poisson equation, but equipped with an exponentially decaying factor so the electric potential drops much faster, which is what we are expecting from a shielding effect.

However, carrying out such an integration is practically unfavourable for most cases, as the time consumption is not satisfying, and precision of numerical solution is also suspicious since the expression to be integrated is divergent at $\mathbf{r} = \mathbf{r}'$. Besides, for most cases the boundary condition is of Dirichlet's type, which means $\sigma(\mathbf{r}')$ is unknown on the boundary of the integrated region. Therefore, a numerical way of solving the

original PDE is practically more favourable.

2   Parallelization

To solve this problem, we use Jacobi's iterative method. The electric potential field is smashed into a vector, and the operator is changed into its matrix form, which could be written as

$$\mathbf{A\Phi} = \frac{\rho}{\varepsilon}$$

The major time consumption is expected to take place during this iterative solver itself.

The parallelization method used in this project is MPI. Since the system matrix $\mathbf{A}$ could become considerably large for those product-size problems, it is a better practice to use a distributed memory method to carry out the parallelization. Also, such a method enables the possibility of running the program on distributed nodes, which means it could be executed with more computational resource and easier to scale compared to those shared memory methods.

As Jacobi iterative solver needs communications during solving to update the target vector on each process, Map-reduce is not an appropriate method for this problem. OpenMP and Threads are both shared-memory methods and are less favoured as discussed above. The iterative method could be decomposed to row-wise operations for the matrix, and during the calculation step (after which will be the updating of vector), there is no need to get access to other matrix elements located on other processes, while `MPI_Broadcast()` is enough to take on the task of updating of the vectors, therefore there is no need to make use of global address space, which means there is no need to use PGAS method.

MPI method requires a drastic reconstruction of the original code, therefore nearly every class needs rewriting. However, the most important parallelizations are on the initialization of A matrix and the iterative solving steps. Since each process only stores part of A matrix, this could be generated in a parallel way for this program. During the iterative solving of the linear equation, we need to multiply the vector by the matrix to get the updated value, while this could be done independently for each row of vector, therefore by storing the matrix distributively, the matrix multiplication could also be parallelized, which is the most significant part of the parallelization for this certain method.

As most of the time consumption for solving takes place in Jacobi solver itself, it is possible that the percentage of parallelization is very close to 100%, at least higher than 95%, at first glance. However, from the following discussion on profiling of the parallel version of the code, we will find that parallelization is not a free lunch with the cost of communications between processes, especially broadcasting of data, whose time consumption can even grow as the parallelization size gets larger. A small portion of the code, such as I/O operations, are intrinsically not able to be parallelized. Besides, broadcasting the right hand side of the linear equation to every process during the initialization is also a step which could not be parallelized, though all these vector operations are typically must faster than matrix operations. For a linear system with size of N, this will cost around $1/N$ of total running time.

Setting up the matrix is similar to that of sequential version of program, except that now only a piece of matrix is stored on one single process, and each process only needs to generate a part of it, therefore as could be seen in the source code, in the generation of system matrix, we add conditions to jump through the loops, so that only a small portion of elements will be covered on each process. Generation of b vector (right hand side) is similar, but as this vector is stored with a complete copy on each process, we need to use broadcast after the generation. These are the obvious differences in generation of the matrix and the RHS vector.

3 Profiling

3.1 Serial

The profiling is done with a $20 \times 20 \times 20$ grid number of 3D-field, with Dirichlet's boundary condition, therefore dimension of the linear system is 5832. The profile result for the serial code is shown as below:
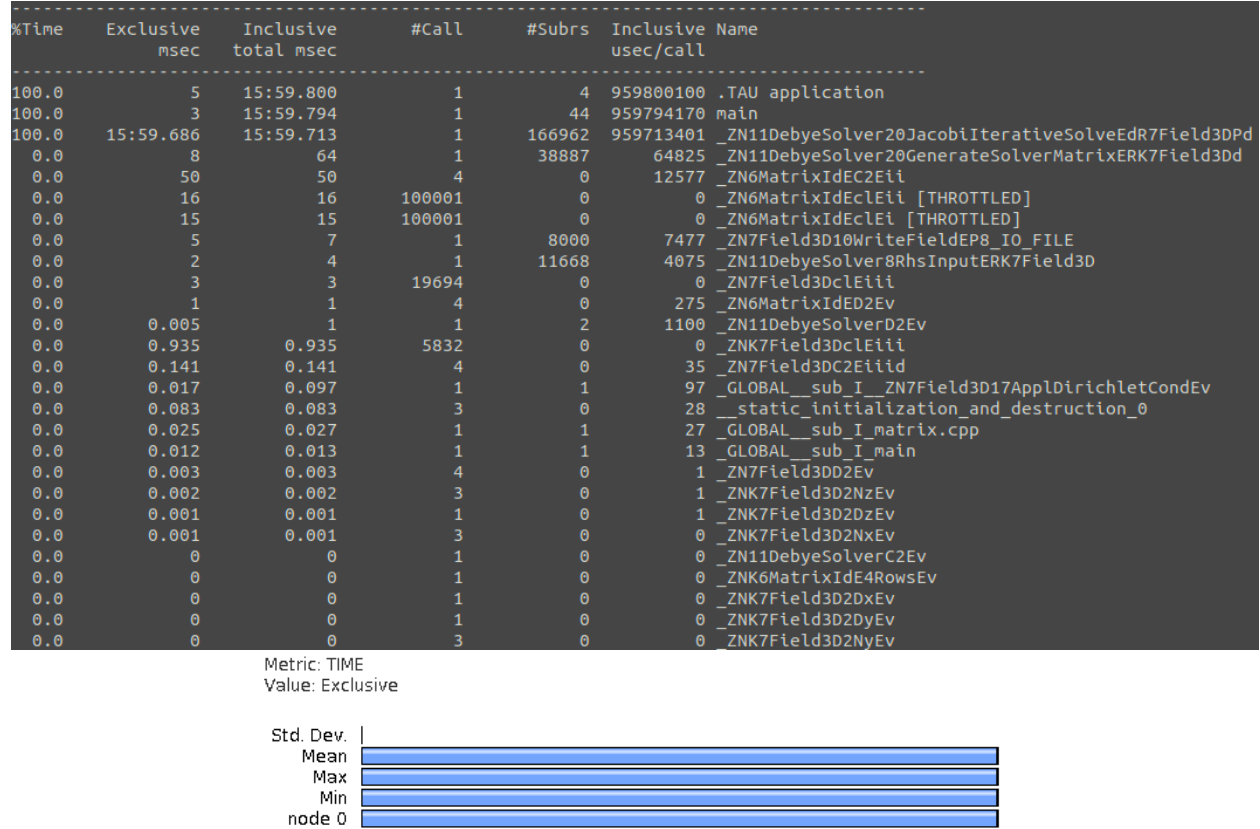
Figure 1: The profile results of serial code.

As could be seen, nearly 100% of time is consumed on the `JacobiIterativeSolve()` function, which is the major function of iterative solver itself. This is just the same as our expectation discussed previously.

As nearly all time consumption comes from the solving of equation itself, there is no obvious and simple way to upgrade the efficiency. Here we are simply changing the compiler's flag from `-O2` to `-O3`. Other possible ways might include changing the compilers, rewriting the code to make full use of CPU cache, or try to optimize the algebraic operations to make full use of FMA.

Below is the updated profile with the new optimization flag.

```
-----------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
             msec   total msec                            usec/call
-----------------------------------------------------------------------------
100.0           6    15:44.929           1           4  944929511 .TAU application
100.0           5    15:44.922           1          44  944922503 main
100.0   15:44.809    15:44.836           1      166962  944836908 _ZN11DebyeSolver20JacobiIterativeSolveEdR7Field3DPd
  0.0           8           70           1       38887      70074 _ZN11DebyeSolver20GenerateSolverMatrixERK7Field3Dd
  0.0          55           55           4           0      13846 _ZN6MatrixIdEC2Eii
  0.0          16           16      100001           0          0 _ZN6MatrixIdEclEii [THROTTLED]
  0.0          16           16      100001           0          0 _ZN6MatrixIdEclEi [THROTTLED]
  0.0           3            5           1        8000       5058 _ZN7Field3D10WriteFieldEP8_IO_FILE
  0.0           2            4           1       11668       4112 _ZN11DebyeSolver8RhsInputERK7Field3D
  0.0           3            3       19694           0          0 _ZN7Field3DclEiii
  0.0        0.98         0.98        5832           0          0 _ZNK7Field3DclEiii
  0.0       0.846        0.846           4           0        212 _ZN6MatrixIdED2Ev
  0.0       0.003         0.84           1           2        840 _ZN11DebyeSolverD2Ev
  0.0       0.117        0.117           4           0         29 _ZN7Field3DC2Eiiid
  0.0       0.015        0.092           1           1         92 _GLOBAL__sub_I__ZN7Field3D17ApplDirichletCondEv
  0.0       0.079        0.079           3           0         26 __static_initialization_and_destruction_0
  0.0       0.031        0.032           1           1         32 _GLOBAL__sub_I_matrix.cpp
  0.0       0.012        0.013           1           1         13 _GLOBAL__sub_I_main
  0.0       0.002        0.002           4           0          0 _ZN7Field3DD2Ev
  0.0       0.001        0.001           1           0          1 _ZNK7Field3D2DzEv
  0.0       0.001        0.001           3           0          0 _ZNK7Field3D2NyEv
  0.0       0.001        0.001           3           0          0 _ZNK7Field3D2NzEv
  0.0           0            0           1           0          0 _ZN11DebyeSolverC2Ev
  0.0           0            0           1           0          0 _ZNK6MatrixIdE4RowsEv
  0.0           0            0           1           0          0 _ZNK7Field3D2DxEv
  0.0           0            0           1           0          0 _ZNK7Field3D2DyEv
  0.0           0            0           3           0          0 _ZNK7Field3D2NxEv
```

Metric: TIME
Value: Exclusive

Std. Dev. |
Mean
Max
Min
node 0

Figure 2: The profile results of serial code with improved compiling flag.

As could be seen from the results, there is a slight improvement in the speed. For the same problem size, the time consumed decreases from 15 min 59 s to 15 min 45 s, shortened by around 1.5%. However, this might also be influenced by other factors such as the load of server.

3.2 Parallel

The profiling is done with a $20 \times 20 \times 20$ grid number of 3D-field (i.e., still sample size), with Dirichlet's boundary condition, therefore dimension of the linear system is 5832. The profile result for the MPI parallel version of code is shown as below, with -np=10:

```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive      #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------------
100.0           54       53,297          1          61   53297765 .TAU application
 99.9            4       53,243          1     16017.1   53243002 main
 99.7       36,301       53,113          1      108106   53113803 _ZN12MDebyeSolver20JacobiIterativeSolveEdR8MField3DPdi
 30.5       16,258       16,258       3280           0       4957 MPI_Bcast()
 29.1        0.929       15,520        164        1640      94635 _Z12MPIArraySyncPdiii
  2.4         0.91        1,258        164        3280       7671 _Z12MPIGatherMaxPdii
  1.0          540          540       1645           0        329 MPI_Barrier()
  0.2       0.0422           87          1           3      87812 _Z8MPIStartRiS_
  0.2           87           87          1           0      87769 MPI_Init()
  0.0           24           24     100001           0          0 _ZN4MMatIdE4ElemEii [THROTTLED]
  0.0           11           11          1           0      11013 MPI_Finalize()
  0.0            4            5          1      3903.1       5528 _ZN12MDebyeSolver20GenerateSolverMatrixERK8MField3Dd
  0.0            4            4      28465           0          0 _ZN8MField3DclEiii
  0.0            1            2          1        5835       2254 _ZN12MDebyeSolver8RhsInputERK8MField3D
  0.0            1            1       5832           0          0 _ZNK8MField3DclEiii
  0.0        0.562        0.722        0.1         800       7223 _ZN8MField3D10WriteFieldEPKc
  0.0        0.103        0.533          1           1        533 _ZN12MDebyeSolverD2Ev
  0.0         0.43         0.43          1           0        430 _ZN4MMatIdED2Ev
  0.0       0.0339       0.0385          2           2         19 _ZN3MPI9IntracommC2EP19ompi_communicator_t
  0.0       0.0296       0.0326          1           1         33 _GLOBAL__sub_I__ZN8MField3DC2Eiiidddii
  0.0       0.0231       0.0241          1           1         24 _GLOBAL__sub_I_main
  0.0       0.0214       0.0224          1           1         22 _GLOBAL__sub_I__Z12MPIArraySyncPdiii
  0.0       0.0214       0.0224          1           1         22 _GLOBAL__sub_I_matrix_mpi.cpp
  0.0       0.0059       0.0197          1           1         20 _ZN4MMatIdE5CleanEv
  0.0        0.008       0.0195          1           2         20 _ZN4MMatIdE4InitEii
  0.0       0.0144       0.0144          1           0         14 _ZN4MMatIdEC2Eii
  0.0       0.0138       0.0138          1           0         14 MPI_Type_free()
  0.0       0.0131       0.0131          2           0          7 _ZN8MField3DC2Eiiidddii
  0.0       0.0089       0.0089         47           0          0 _ZN3MPI8DatatypeC2EP15ompi_datatype_t
  0.0       0.0083       0.0083          1           0          8 MPI_Type_contiguous()
  0.0        0.006        0.006          4           0          2 __static_initialization_and_destruction_0
  0.0       0.0046       0.0046          2           0          2 _ZN3MPI14Is_initializedEv
  0.0       0.0035       0.0035          3           0          1 _ZN3MPI10ErrhandlerC2EP17ompi_errhandler_t
  0.0       0.0032       0.0032          1           0          3 MPI_Type_commit()
  0.0       0.0019       0.0019          2           0          1 _ZN8MField3DD2Ev
  0.0       0.0018       0.0018          3           0          1 _ZNK8MField3D2NxEv
  0.0       0.0013       0.0013          3           0          0 _ZNK8MField3D2NzEv
  0.0       0.0009       0.0009          1           0          1 _ZN12MDebyeSolverC2Eii
  0.0       0.0009       0.0009          2           0          0 _ZNK4MMatIdE11SliceOffsetEv
  0.0       0.0008       0.0008          2           0          0 _ZN3MPI5GroupC2EP12ompi_group_t
  0.0       0.0008       0.0008          1           0          1 _ZNK8MField3D2DzEv
  0.0       0.0008       0.0008          3           0          0 _ZNK8MField3D2NyEv
  0.0       0.0007       0.0007          2           0          0 _ZNK4MMatIdE9SliceRowsEv
  0.0       0.0005       0.0005          1           0          0 MPI_Comm_size()
  0.0       0.0005       0.0005          1           0          0 _ZN3MPI4InfoC2EP11ompi_info_t
  0.0       0.0005       0.0005          1           0          0 _ZN3MPI7RequestC2EP14ompi_request_t
  0.0       0.0005       0.0005          1           0          0 _ZNK8MField3D2DyEv
  0.0       0.0004       0.0004          1           0          0 _ZNK8MField3D2DxEv
  0.0       0.0001       0.0001          1           0          0 MPI_Comm_rank()
```
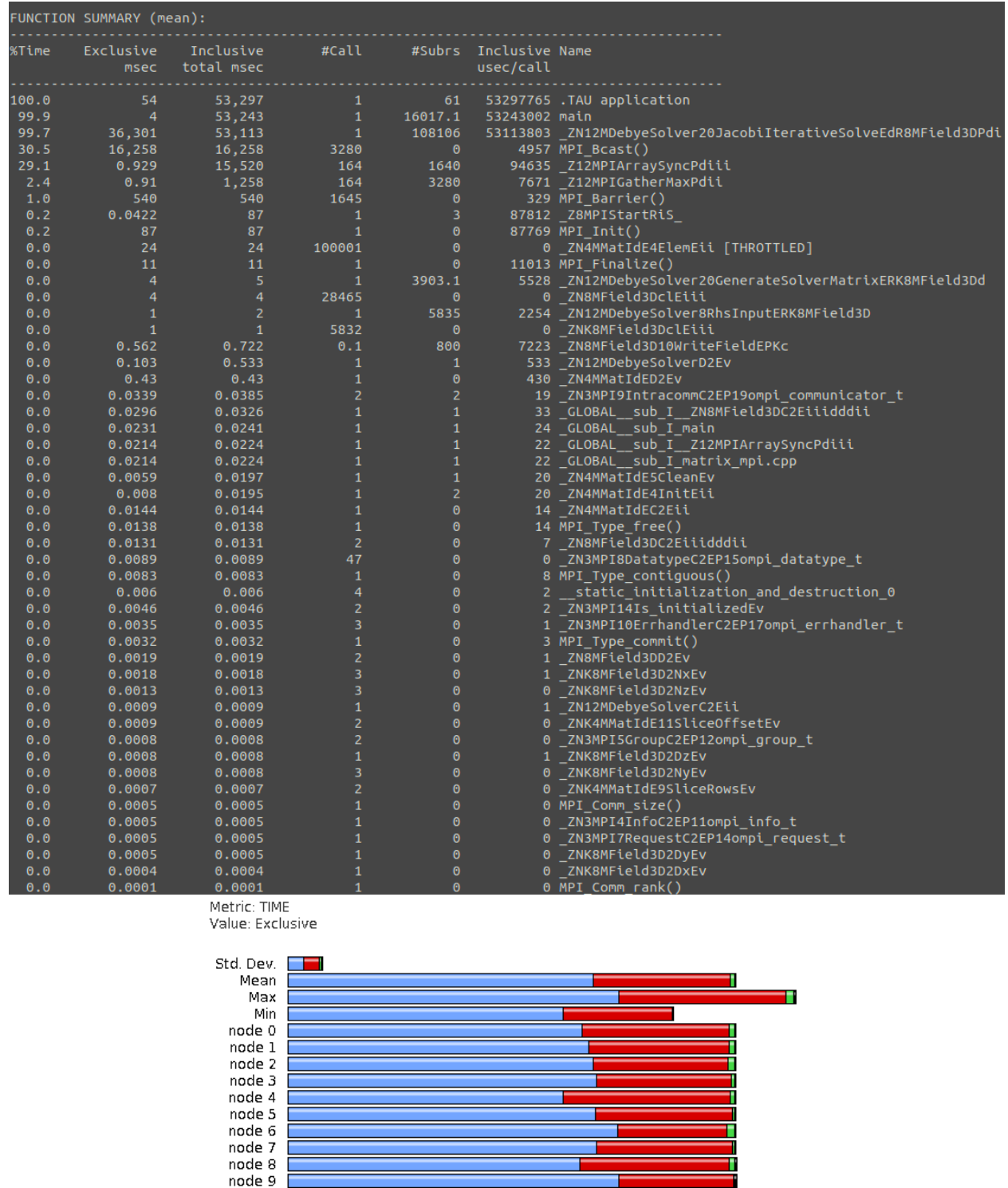
Metric: TIME
Value: Exclusive

Figure 3: The profile results of parallel code.

As shown in the profile, the major contribution of the time consumption still comes from the algebraic operations inside of the solver itself (shown in blue). However, this time we see a significant contribution from the talk between processes, i.e., time consumption on MPI_Bcast() (shown in red).

These two factors are both as expected, however, the time consumption on the talking is much higher than the initial guess, since it is obviously not at a scale of $1/N$. This might be related with specific computer

architecture, or number of nodes involved (though we are only using one node here).

A direct comparison of speed could not be provided by the executable file with the profiling tool implemented, since the profile can induce noticeable difference between serial and parallel code. Here we only compare the execution time ratio of different functions for serial and parallel codes. As could be seen, the `JacobiIterativeSolve()` method is still the most time-consuming part, while at this case it is distributed onto different nodes, and the total amount of time is therefore decreasing, as expected.

Compared with the serial code in PR #1, the parallel version of code is running with way much shorter time (the serial part of code is taken from the code in PR #1), which could be seen from the profile results. However, the memory cost is slightly rising, since each node is now holding a complete copy of the RHS vector and solution vector. But this is not an obvious cost, since the size of matrix is thousands of times larger, while that is stored piecewise and shall not induce a strong memory increase.

As MPI communication is time-consuming for parallel code with `-np=10`, it is possible to substitute `MPI_Bcast()` by other functions as `MPI_Allgatherv()`, to see which has a higher efficiency. With such a modification, the profile is shown as below:
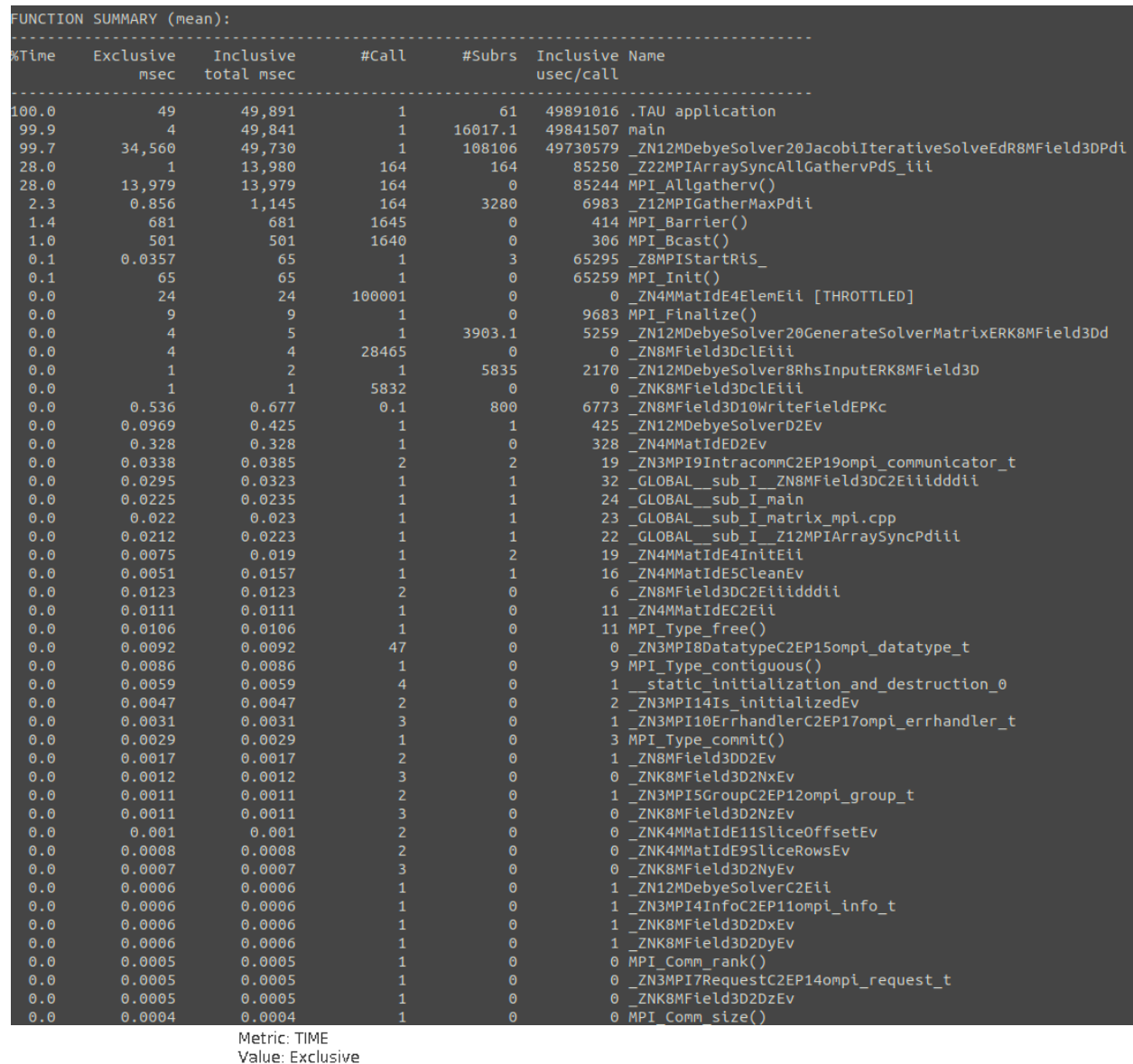
```
FUNCTION SUMMARY (mean):
-------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
-------------------------------------------------------------------------------------
100.0          49       49,891           1          61   49891016 .TAU application
 99.9           4       49,841           1     16017.1   49841507 main
 99.7      34,560       49,730           1      108106   49730579 _ZN12MDebyeSolver20JacobiIterativeSolveEdR8MField3DPdi
 28.0           1       13,980         164         164      85250 _Z22MPIArraySyncAllGathervPdS_iii
 28.0      13,979       13,979         164           0      85244 MPI_Allgatherv()
  2.3       0.856        1,145         164        3280       6983 _Z12MPIGatherMaxPdii
  1.4         681          681        1645           0        414 MPI_Barrier()
  1.0         501          501        1640           0        306 MPI_Bcast()
  0.1      0.0357           65           1           3      65295 _Z8MPIStartRiS_
  0.1          65           65           1           0      65259 MPI_Init()
  0.0          24           24      100001           0          0 _ZN4MMatIdE4ElemEii [THROTTLED]
  0.0           9            9           1           0       9683 MPI_Finalize()
  0.0           4            5           1      3903.1       5259 _ZN12MDebyeSolver20GenerateSolverMatrixERK8MField3Dd
  0.0           4            4       28465           0          0 _ZN8MField3DclEiii
  0.0           1            2           1        5835       2170 _ZN12MDebyeSolver8RhsInputERK8MField3D
  0.0           1            1        5832           0          0 _ZNK8MField3DclEiii
  0.0       0.536        0.677         0.1         800       6773 _ZN8MField3D10WriteFieldEPKc
  0.0      0.0969        0.425           1           1        425 _ZN12MDebyeSolverD2Ev
  0.0       0.328        0.328           1           0        328 _ZN4MMatIdED2Ev
  0.0      0.0338       0.0385           2           2         19 _ZN3MPI9IntracommC2EP19ompi_communicator_t
  0.0      0.0295       0.0323           1           1         32 _GLOBAL__sub_I__ZN8MField3DC2Eiiidddii
  0.0      0.0225       0.0235           1           1         24 _GLOBAL__sub_I_main
  0.0       0.022        0.023           1           1         23 _GLOBAL__sub_I_matrix_mpi.cpp
  0.0      0.0212       0.0223           1           1         22 _GLOBAL__sub_I__Z12MPIArraySyncPdiii
  0.0      0.0075        0.019           1           2         19 _ZN4MMatIdE4InitEii
  0.0      0.0051       0.0157           1           1         16 _ZN4MMatIdE5CleanEv
  0.0      0.0123       0.0123           2           0          6 _ZN8MField3DC2Eiiidddii
  0.0      0.0111       0.0111           1           0         11 _ZN4MMatIdEC2Eii
  0.0      0.0106       0.0106           1           0         11 MPI_Type_free()
  0.0      0.0092       0.0092          47           0          0 _ZN3MPI8DatatypeC2EP15ompi_datatype_t
  0.0      0.0086       0.0086           1           0          9 MPI_Type_contiguous()
  0.0      0.0059       0.0059           4           0          1 __static_initialization_and_destruction_0
  0.0      0.0047       0.0047           2           0          2 _ZN3MPI14Is_initializedEv
  0.0      0.0031       0.0031           3           0          1 _ZN3MPI10ErrhandlerC2EP17ompi_errhandler_t
  0.0      0.0029       0.0029           1           0          3 MPI_Type_commit()
  0.0      0.0017       0.0017           2           0          1 _ZN8MField3DD2Ev
  0.0      0.0012       0.0012           3           0          0 _ZNK8MField3D2NxEv
  0.0      0.0011       0.0011           2           0          1 _ZN3MPI5GroupC2EP12ompi_group_t
  0.0      0.0011       0.0011           3           0          0 _ZNK8MField3D2NzEv
  0.0       0.001        0.001           2           0          0 _ZNK4MMatIdE11SliceOffsetEv
  0.0      0.0008       0.0008           2           0          0 _ZNK4MMatIdE9SliceRowsEv
  0.0      0.0007       0.0007           3           0          0 _ZNK8MField3D2NyEv
  0.0      0.0006       0.0006           1           0          1 _ZN12MDebyeSolverC2Eii
  0.0      0.0006       0.0006           1           0          1 _ZN3MPI4InfoC2EP11ompi_info_t
  0.0      0.0006       0.0006           1           0          1 _ZNK8MField3D2DxEv
  0.0      0.0006       0.0006           1           0          1 _ZNK8MField3D2DyEv
  0.0      0.0005       0.0005           1           0          0 MPI_Comm_rank()
  0.0      0.0005       0.0005           1           0          0 _ZN3MPI7RequestC2EP14ompi_request_t
  0.0      0.0005       0.0005           1           0          0 _ZNK8MField3D2DzEv
  0.0      0.0004       0.0004           1           0          0 MPI_Comm_size()
```


Figure 4: The profile results of parallel code with MPI_Allgatherv().

After changing the MPI communication function, there is a slight accelaration in the average running time of the functions. Another way is to reduce the number of processes to no larger than number of cores on the node, see the scaling discussion below for details.

4 Scaling

4.1 Strong Scaling

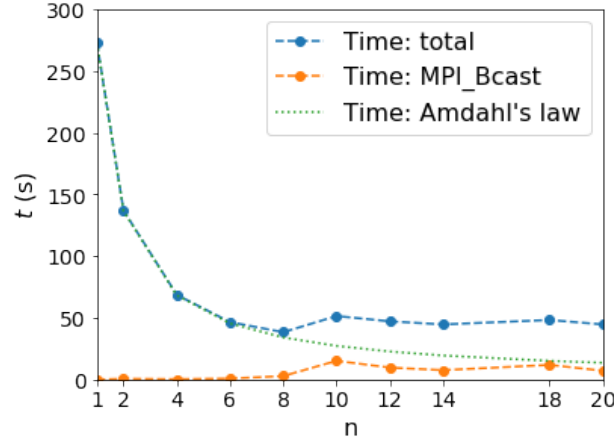The plot for execution time of strong scaling on login node of ACI is shown as below:

Figure 5: The profile results of parallel code with MPI_Allgatherv().

The plot starts with $N = 1$, which is actually serial, with the same problem size as above examples. This size is already large enough to profile the speed up of parallelization, and starting from $N = 1$ makes it easy to get the expected computation time from Amdahl's law. From this plot we could also see that the number `-np=10` is actually not a good choice for the parallel code profiling example, since it gives a large overhead of communication time between processes. However, this does not influence the validity of those discussions.

From the plot, it could be seen that `-np=8` should be used for speed for the sample-sized problem, while the best efficiency is reached at `-np=1`, where no communication happens. For a research problem, more processes with an acceptable efficiency will be preferred as this might be related with the cost of computational resources, therefore `-np=8` can be used as a result of balancing between performance and efficiency, if efficiency does not vary as the size of problem grows, which needs to be further investigated by weak scaling tests. Also, the communication cost rises drastically after $N \geq 10$, which can be related with the server and node structure, and could be varying on different architectures. For this test, since the login node has 8 cores, which could be confirmed with "`cat /proc/cpuinfo`", the speed up will be limited after that number, i.e., $N > 8$. Therefore, for nodes with more cores, using more processes, such as $N = 10$, might be a better choice.

If we take $p \approx 100\%$ from the profile of serial code, then Amdahl's law could be written as

$$S = \frac{t_{\text{serial}}}{t_{\text{parallel}}} \approx s = \min\left(N_{\text{processes}}, N_{\text{cores}}\right).$$

However, this is only possible if no communication cost, which could even arise (seen from above), is considered.

From the plot above, we can see that for $N \leq 8$, the speed is very close to ideal, but for other numbers of processes, the program is way much slower (except for the serial case, $N = 1$). The reason for a lower speed with $N > 8$ is already discussed as a limit of number of cores, and for $N = 8$ the communication cost is already arising, therefore we see a small but observable deviation from the ideal expectation.

5 Discussion and Conclusions

In this report, we have discussed the implementation of parallel code for the iterative solver for Poisson equation with Debye-Huckel's linear shielding term. Profile of the code as well as scaling of the code have been discussed. MPI is selected as the parallelization method because of the requirement of vector updating and communication, as well as the requirement of distributive storage of the matrix. As could be seen, the performance is still limited by the communication between processes, whether using `MPI_Bcast()` or `MPI_Allgatherv()`. The future work should be related to getting a better performance of communications, and also accelerating the algebraic operations if possible.

Appendices

A Acknowledgements

The bracket notation in calling of the matrix element and the stream output operator of matrix are inspired by the grammar from linear algebra template header library named Eigen (`http://eigen.tuxfamily.org`). The author appreciates these notations making debugging a more concise process.

B Code

The readers could find the code on Github with the username `sunnyssk`, under the archive `CSE597-hw2`.

Under the root of archive, we have the "sequential" and "mpi" folder, storing the serial and parallel version of the code. Also there is an iPython-notebook for visualizing the results of electric potential.

For serial version, source files are under `src` sub-folder, with `matrix.cpp` and `matrix.h` containing the matrix class, `debye.cpp` and `debye.h` containing the linear solver class, as well as the field container class, and `main.cpp` as the entrance calling the solver.

For parallel version, the structure of source file is pretty much the same, except that we are now using `_mpi` in the filenames, and there are some MPI functions defined in `mpi_util.cpp`.

The instructions could be found in `Readme.md` under the root of archive. Please check out there. To reproduce the results, please use `tau` as the profiler. Due to differences in the performance of each node, it could be possible that one gets different results from that described in this report, but the tendency should be the same.

On ACI-b logon servers, you could compile and run the code directly.

C Poster Draft - separate document

Please see the poster folder for the first draft.

References

[1] Li, Y., Hu, S., Liu, Z., and Chen, L.-Q. Effect of electrical boundary conditions on ferroelectric domain structures in thin films. *Appl. Phys. Lett. 81*, 3 (2002).