Final Project

CSE 597

Weiming Hu

Friday, December 7, 2018

Contents

Abstract

Analog Ensemble is a highly parallelizable and scalable technique to generate probabilistic forecasts using historical deterministic predictions and the corresponding observations. Since the year of 2013 when it is first brought up, it has been successfully applied to short-term wind and temperature forecasts and also to spatial and temporal down-scaling of other numeric weather prediction models, like the Global Forecast System. However, insufficient studies have focused on evaluating the quality of probabilistic forecasts. Problems have already emerged when the Analog Ensemble technique is applied to a gridded model output. Because of the intrinsic feature of the search algorithm, the produced forecasts loses the spatial continuity that should be present in a realistic physical world. Although Schaake Shuffle can be applied to the forecasts to improve the spatial continuity in visualization, new understandings are needed to better explain and resolve the problem.

This study proposes a matrix approach to analyze the quality of ensemble members by solving a linear system. Results can show the contribution of each ensemble member to the overall prediction accuracy. The linear system solver is an MPI implementation of the Gauss iterative solver. NetCDF is used for data I/O.

1 Problem of Interest

1.1 General Overview

Weather forecasts were viewed as intrinsically deterministic before the early 1990s [2]. Although numerical weather models are still routinely run today to generate deterministic values for weather predictions, however, probabilistic forecasts have shown greater advantage in various fields. Forecasts associated with probabilities allow interpreters and data analyzers to build utility functions to derive weather-related economic values, for example, power generation and load prediction, and weather risk assessment. Probabilistic forecasts simply provide end users with more information on the forecast uncertainty, therefore receiving more preference over deterministic predictions.

Ensemble modeling is one of the most popular techniques to generate probabilistic forecasts. Generally, ways to generate probability information can be divided up into either by running a single numerical weather model for multiples times with perturbed initialization, or by using different model physics. This Monte Carlo style simulation addresses the chaotic nature of the atmosphere presented [7] which can be associated with the imperfection in model instantiation and parameter propagation.

Meanwhile, The transformation from deterministic predictions to ensemble forecasts and probabilistic forecasts poses challenges on model verification. How to evaluate a probabilistic forecast then becomes a critical question. Traditional verification metrics, for example, contingency table for categorical predictions and Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) for continuous predictions, are no longer applicable for probabilistic forecast verification. Although post processing can be done to collapse a probabilistic forecast to a deterministic prediction, important information on model uncertainty will be lost after the practice. A set of well-established metrics include but are not limited to the rank histogram, the Continuous Rank Probabilistic Score (CRPS), and the Brier score [3].

The ensemble modeling technique used in this project is the Analog Ensemble (AnEn) technique [1]. AnEn is a data-driven technique to generate probabilistic forecasts using a set of historical deterministic predictions and the corresponding observations or the model analysis field. To generate a future probabilistic forecast, the technique first takes the future deterministic prediction from a numerical weather model and looks for the most similar historical predictions in the same model. The similarity is defined using a weighted multi-variate distance function. After the most similar historical predictions are found, the corresponding observations are taken to form the forecast ensemble. This technique has been successfully applied to surface wind and temperature forecasts, and have been found to be more computationally efficient than conventional ensemble modeling techniques because it does not require additional model simulation.

However, because numerical weather models are never perfect, the ensemble members will never be perfect to the observation as well. Verification of AnEn therefore becomes an important research question. This project studies a verification technique of spatial ensemble forecasts using a matrix approach. AnEn has been

successfully applied to gridded model simulation, but forecast results have shown different level of spatial discontinuity [9]. This is partly due to the practice that AnEn are generated for each spatial grid point, and the spatial correlation is not fully preserved. This verification technique of spatial ensemble forecasts can be used to quantify and diagnose the model prediction skills, for example, under-/over- predictions, as well as the spatial discontinuity of the probabilistic forecasts introduced by the AnEn technique.

1.2 Scientific Question and Merit

The goal of this project is to study a verification technique for spatial ensemble forecasts generated from the AnEn and to develop an efficient solver for the problem. Building upon the work done by [9], the project aims to improve the understanding of the contribution of each ensemble member from the AnEn. The knowledge plays an important role when the AnEn is applied to higher resolution model output and when probability information should be kept in the results for further references. Although Sperati and etc. have proposed Schaake Shuffle as a useful technique to reconstruct the spatial continuity, whether this method produces the optimal condition is unknown, and the actual practice of Schaake Shuffle still involves a lot with randomness, therefore the result quality not being guaranteed.

Conventionally, all ensemble members are used to generate distribution and probability information without inspecting which member might be more helpful in approximating the "true" distribution. Ensemble members are usually assigned with equal weights when generating probabilistic information. However, different ensemble members should vary in its predictive ability. For example, places that are closer to each other should have a similar trend in air temperature than other places that are further away. The proposed method is therefore designed to quantify the differences among various ensemble members.

This method can also be applied to other ensemble modelling statistics and verification as long as there is spatial component in ensemble members. Most of the ensemble models are run on a regular mesh that can be represented with a vector of coordinates. If the ensemble model use this paradigm to represent the 2-dimensional space, the verification method studied in this project can be applied. However, situations with a 3-dimensional space or only a single location might not be solved using this technique.

Although this is the first time to my knowledge that the accuracy contribution of each ensemble member of AnEn is studied, there are certainly other ways to solve the linear system for solution validation. One common function in R is *solver()* which solves a system given a coefficient matrix and a vector. Some other C/C++ libraries are also available for this purpose, for example, *Armaddillo* and *CLAPACK*. In this work, the R solver is used to verify results.

1.3 Numerical Set-up

The problem can be represented by the following equation:

$$\begin{bmatrix} — & F_1^n & — \\ — & F_2^n & — \\ \vdots & \vdots & \vdots \\ — & F_i^n & — \\ \vdots & \vdots & \vdots \\ — & F_m^n & — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} O_1 \\ O_2 \\ \vdots \\ O_i \\ \vdots \\ O_m \end{bmatrix}$$

where,

1. m is the number of spatial grid points;
2. n is the number of members in each ensemble;
3. F_i^n is the ith ensemble forecast vector with n members;

4. $\{x_1, x_2, ..., x_n\}$ is the verification metric vector for ensemble forecasts;
5. O_i is the ith observation;

On the left side of the equation, matrix F is the spatial forecast ensemble. Each line represent a forecast ensemble at a spatial location, or at a spatial grid point. The number of grid points is determined by the spatial resolution of the underlying numeric weather model being used which reaches to 262,792 grid points in the case of North American Mesoscale Forecast System (NAM) with a 11-km spatial resolution. Each value in a row is a ensemble member value. The number of the ensemble members can range from tens to hundreds depending on the size of the historical predictions. Vector $\vec{O}$ is the corresponding observations or model analysis. Vector $\vec{x}$ is the verification metric of the spatial forecast ensemble. It is assumed that observations come from a certain unknown distribution, and the forecast ensemble should be a sufficiently "good" approximation to the realistic unknown distribution. If the ensemble is a perfect ensemble, each ensemble member should be equally good comparing to the other ensemble members in a spatial context. Therefore, the coefficient $\vec{x}$ should place similar weights to all the ensemble members, rather than some ensemble members having a higher co-efficiency than others. In reality, because bias exists and error propagates, the actual forecast ensemble might be under- or over- predicting. And the coefficient $\vec{x}$ can be used to characterize it.

The matrix F should be a dense matrix because the most part of the matrix will be filled up with values. There will be cases that null values might exist, but it should be not be a very common situation. The matrix F might be a "tall" matrix in a sense that the number of spatial grid points will largely surpass the number of forecast ensemble members.

The size of the matrix varies dramatically based on the size of ensembles and the number of grid points. In the case of NAM, there are 262 792 grid points in total with a 11 km resolution. The size of ensembles is typically decided by the size of historical forecasts used in to generate the ensemble. A good practice is the square root of the size of historical forecasts. Therefore, the size of ensembles is estimated to be between dozens to hundreds. Although the dense matrix can be made square, it will be a dense narrow matrix in most cases. In cases of square matrices, the traditional solvers like Gauss Elimination and Jacobi methods can be directly used without modification. However, in order to solve a linear system where the number of equations and the number of variables do not match, modification to the algorithms or to the problem set up is required. For non-square linear system, the Normal method can be used to solve linear system with any shapes. For example, when the number of equations is bigger than the number of variables, the Normal method is similar to an Ordinary Least Square Regression that finds the solution that generates the least overall error. For iterative solvers, things might vary based on the shape of the coefficient matrix. In cases where the following linear system presents $Ax = b$ where $A \in \mathbb{R}^{m,n}, m < n$, a generalized Jacobi and Gauss-Seidel method can be used to solve such linear systems with any initial conditions guaranteed to converge [8]. In cases where $m > n$, a different linear system setup might be needed. In stead of solving all equations at once, they can be divided up into groups based on euclidean distance in a geographical sense to drastically decrease the number of equations to solve, and separate groups of equations can be solved in parallel. At last, solutions from different groups can then be aggregated together.

The test data sets can be found in the project data folder. You can find two sub folders, *csv* and *ncdf4*. CSV files are primarily used during the early developmental stage of this project and NetCDF files are introduced during the later stages. CSV files are named with the convention `<type>_<size>.csv`. `<type>` specifies the type of the data represented in the equation $Ax = b$. `<size>` specifies the size of the data. A matrices are all square matrices, b is the right-hand side vector of the equation, and x is the solution vector. These data sets are generated using the R script included in the folder. Basically, a size is predefined, and then a diagonally dominant matrix is randomly generated as A and the b vector is also randomly generated. The solution vector x is calculated using the R function *solve*. A single data set is not sufficient in representing the scalability of the algorithm when dealing with production problem. Therefore, a series of data sets are generated with increasing sizes to test the scalability of the program. For NetCDF files are not included by in the repository due to the file sizes but they can be easily created by running the R script included in the folder.

2 Solvers

2.1 Direct Solver

2.1.1 Justification

There are many ways of solving this linear problem. Three common methods are shown below:

There are three popular ways to solve a linear system: 1)Cramer's Rule; 2) Gaussian Elimination; 3)LU Decomposition. The Cramer's Rule is the most inefficient method out of the three. For a linear system with n equations and n unknowns, the Cramer's Rule requires to solve $n + 1$ different determinants assuming all the determinants exist. However, calculating determinants should be avoided in all efforts because it is a very computationally expensive task whose computational complexity is estimated to be polynomial[5]. Gaussian Elimination and LU Decomposition are usually integrated to solve multiple linear systems with higher efficiency. Calculating determinants can be avoided in these methods by using forward elimination and backward substitution. LU Decomposition involves one more step, the forward substitution, than Gaussian Elimination, leading to the result that LU Decomposition might take longer time to solve one linear system than Gaussian Elimination. However, Gaussian Elimination modifies the coefficient matrix and the right-hand vector internally within each loop, making it impossible to save the transformation and the factorization information of the coefficient matrix. When presented with problems of multiple linear systems where the coefficient matrix does not change and the right-hand vector changes, LU Decomposition has the advantage to save the factorization and reuse the factor matrices for different right-hand vectors. Complexity-wise, Gaussian Elimination is $\mathcal{O}(nm^3)$ and LU Decomposition is $\mathcal{O}(m^3 + nm^2)$ where n is the number of linear systems to solve and m is the size of the problem.

Considering that linear systems will have different coefficient matrices and presumably not many benefits can be gained from using the LU Decomposition method, Gaussian Elimination is selected as the solver in this project. Considering that Gaussian Elimination only works with square matrices but the coefficient matrices dealt with here are usually non-square matrices, the direct solver method is extended to Normal Equation technique where the non-square matrix can be transformed to be square. The Normal Equation can also be referred to as the Ordinary Least Square Regression when there are more equations than the number of unknowns.

The Normal Equation can be expressed as the following: For a linear system $Ax = b$ where A is the coefficient matrix, b is the output vector, and x is the unknown, we have a solution $x = A^t \times (A \times A^t)^{-1} \times b$, where A^t is the transpose of the matrix A.

2.1.2 Optimization Flags

A list of optimization flags for consideration includes: 1) $-O0$; 2) $-O1$; 2) $-O2$; 3) $-O3$; t) $-Ofast$. To decide the best optimization flag out of the five, the direct solver has been compiled and tested with different optimization flags independently. The solvers have been tested on solving a 500-by-500 coefficient matrix on a MacBook Air. The machine model is an early 2015 with 2 2.2. GHz Inter i7 cores and one 8 GB 1 600 MHz DDR3 memory card. The programs are compiled using the Clang compiler, and each test has been repeated for 10 times for statistical significance. The correctness has been verified for all tests in advance.

Figure 1 shows the execution time for the direct solver with different optimization flags. The optimization flag $-Ofast$ generates the lowest averaged execution time. However, the averaged execution time generated by using the flag $-O2$ is very close to to the lowest time, and its spread is smaller than using $-Ofast$. As mentioned in the class materials, $-Ofast$ is the most aggressive flag for code optimization, and it rips of correctness check in precision which may leads to wrong results. Therefore, $-O2$ is chosen to be the optimization flag for the direct solver.

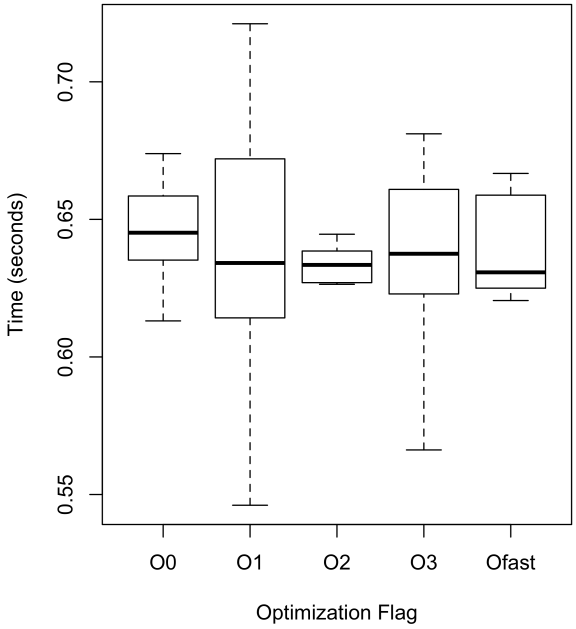

Figure 1: Execution time of the direct solver with different optimization flags.

2.1.3 Profiling

When compiled with the macro definition *-DPROFILE_TIME*, the profiling code will be added to the program and the extra standard output will be provided when the program terminates.

Time The following output is for time profiling of the code. The direct solver is used to solve a 500-by-500 coefficient matrix. The program runs for 0.531 seconds in wall time. The first section of the standard output shows the forward and backward elimination time used by the matrix inverse function.

```
-----------------------------------
Time profiling for the matrix inverse function:
Forward elimination: 0.04851s (66.59%)
Backward elimination: 0.02434s (33.41%)
Inverse function total: 0.07286s (1)
-----------------------------------
Total time for the direct method: 0.531s
```

Figure 2 shows the execution time projection for the direct solver. Each test has been repeated for 10 times, therefore the vertical points shown on the figure. Two regression lines are calculated and plotted on the figure to show the comparison to different algorithm complexity. According to the regression lines, the complexity of the direct solver lies between $O(n^3)$ and $O(n^2)$.

Space The following output comes from the Valgrind utility provided by the ICS cluster. The leak check functionality is turned on. The direct solver is used to solve a 100-by-100 coefficient matrix. The input file for the matrix is 64 KB, and there is 1 505 KB of memory allocated. The message also shows that there is no possible memory leaks. A conservative projection of the memory needed is that it will require about 20 to 30 times of the size of the input file during the calculation.

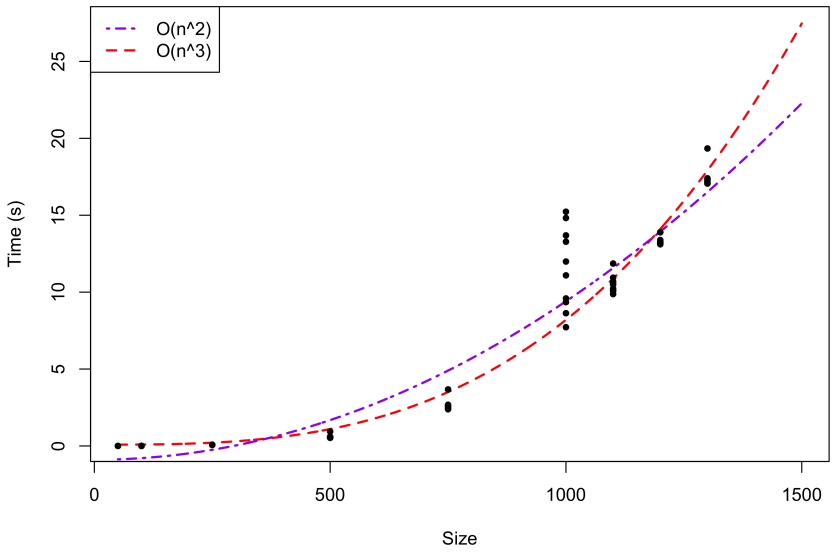

Figure 2: Execution time projection for the direct solver.

```
==18676== Memcheck, a memory error detector
==18676== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==18676== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==18676== Command: ./directSolver.O2 ../data/A_100.csv ../data/b_100.csv 0
==18676==
-----------------------------------
Time profiling for the matrix inverse function:
Forward elimination: 0.02s (66.67%)
Backward elimination: 0.01s (33.33%)
Inverse function total: 0.03s (1)
-----------------------------------
Total time for the direct method: 0.14s
==18676==
==18676== HEAP SUMMARY:
==18676==     in use at exit: 0 bytes in 0 blocks
==18676==   total heap usage: 11,242 allocs, 11,242 frees, 1,505,606 bytes allocated
==18676==
==18676== All heap blocks were freed -- no leaks are possible
==18676==
==18676== For counts of detected and suppressed errors, rerun with: -v
==18676== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

2.2 Iterative Solver

2.2.1 Justification

Popular iterative solvers come from 1) the Jacobi method and 2) the Gauss-Seidel method. They uses the similar iteration scheme. The scheme can be expressed as $x_{k+1} = M^{-1} \times (b - N \times x_k)$, where x is the solution vector, b is the right-hand-side vector, and M and N matrices are defined so that $A = M + N$. The

difference of Jacobi and Gauss-Seidel methods lies in how they define the decomposition matrices M and N. Jacobi method simply define them using the identity matrix. This is a stable approach that the algorithm convergence is guaranteed. However, the convergence rate is very low in practice. The Gauss-Seidel method defines the decomposition using the upper and lower strict triangular parts of the matrix A. This method has a faster convergence.

Although Gauss-Seidel method is in practice more efficient, both methods are implemented in the iterative solver. However, in this section, profiling and projection are only done for Gauss-Seidel method.

2.3 Convergence Criteria

Convergence is based on the residual. If the algorithm runs to the point where the solution has very small residual, the convergence is reached. Here, the small residual is defined as $1e{-}3$. That a residual is less than this small value is considered that the convergence has been reached.

2.4 Initialization

There are currently three initialization methods. The first one is all-1 initialization. The second one is random number initialization. The third one is a good guess initialization though the initialized numbers are not guaranteed to be better than the other two methods in every case. The good guess is generated as a solution x for the first row of the coefficient matrix A and the first number in the b vector.

2.4.1 Optimization Flags

With the similar approach to the previous section, same tests have been carried out for programs compiled with different optimization flags. A 500-by-500 coefficient matrix is used to test programs. Each test has been repeated for 10 times for statistical significance.

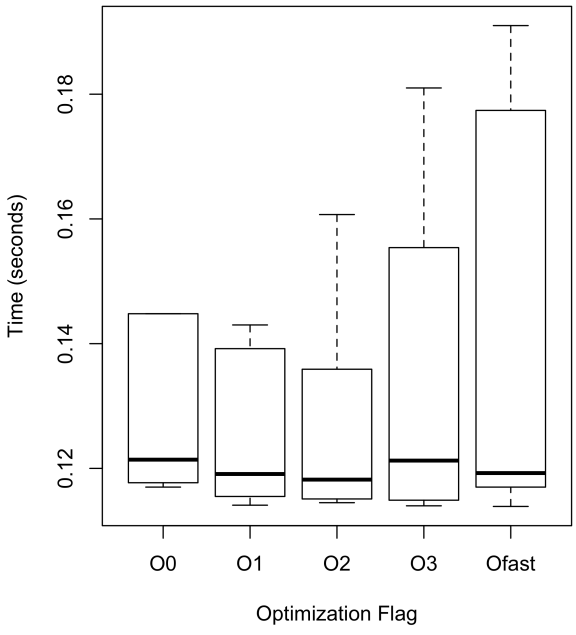

Figure 3: Execution time of the Gauss-Seidel iterative solver with different optimization flags.

Different from the case of the direct solver, $-O2$ works best with the iterative solver where it generates the lowest averaged execution time and the smallest spread. Therefore, this optimization flag is chosen.

2.4.2 Profiling

Time The following output is for time profiling of the code. The experiments were carried out on a Mac Air with a 2.2GHz Intel i7 core and 8 GB 1600 MHz DDR3 memory. The Gauss-Seidel iterative method is used to solve a 500-by-500 coefficient matrix. The program terminates after 0.1015 second, and it runs for 17 iterations. All-1 initialization is used for this example test. The first section of the standard output shows the forward and the backward elimination time used by the matrix inverse function.

```
-----------------------------------
Time profiling for the matrix inverse function:
Forward elimination: 1.057s (52.68%)
Backward elimination: 0.9499s (47.32%)
Inverse function total: 2.007s (1)
-----------------------------------
Iteration 1 residual: 3.294e+07
Iteration 2 residual: 3.897e+06
Iteration 3 residual: 1.096e+06
Iteration 4 residual: 1.873e+05
Iteration 5 residual: 4.608e+04
Iteration 6 residual: 9198
Iteration 7 residual: 1995
Iteration 8 residual: 454.8
Iteration 9 residual: 87.32
Iteration 10 residual: 22.34
Iteration 11 residual: 3.906
Iteration 12 residual: 1.084
Iteration 13 residual: 0.1781
Iteration 14 residual: 0.05145
Iteration 15 residual: 0.008354
Iteration 16 residual: 0.002367
Iteration 17 residual: 0.0004003
Total time for the iterative method: 2.187s
```

As a comparison, the profiling for the iterative method with random numbers and good guess initialization are compared and shown below. The random number initialization takes the most iterations to generate the final results. The good guess method takes slightly fewer iterations than the all-1 initialization. However, since extra time is required to generate the good guess numbers, the runtimes of both methods do not have significant differences.

```
# Random number initialization
-----------------------------------
Time profiling for the matrix inverse function:
Forward elimination: 1.061s (52.76%)
Backward elimination: 0.9496s (47.24%)
Inverse function total: 2.01s (1)
-----------------------------------
Iteration 1 residual: 3.617e+16
Iteration 2 residual: 4.309e+15
Iteration 3 residual: 1.222e+15
```

```
Iteration 4 residual: 2.085e+14
Iteration 5 residual: 5.141e+13
Iteration 6 residual: 1.024e+13
Iteration 7 residual: 2.226e+12
Iteration 8 residual: 5.065e+11
Iteration 9 residual: 9.741e+10
Iteration 10 residual: 2.489e+10
Iteration 11 residual: 4.356e+09
Iteration 12 residual: 1.208e+09
Iteration 13 residual: 1.985e+08
Iteration 14 residual: 5.735e+07
Iteration 15 residual: 9.31e+06
Iteration 16 residual: 2.64e+06
Iteration 17 residual: 4.46e+05
Iteration 18 residual: 1.165e+05
Iteration 19 residual: 2.177e+04
Iteration 20 residual: 5044
Iteration 21 residual: 1074
Iteration 22 residual: 220.1
Iteration 23 residual: 53.01
Iteration 24 residual: 9.726
Iteration 25 residual: 2.592
Iteration 26 residual: 0.4388
Iteration 27 residual: 0.1247
Iteration 28 residual: 0.02027
Iteration 29 residual: 0.005844
Iteration 30 residual: 0.0009592
Total time for the iterative method: 2.307s

# Good guess initialization
-----------------------------------
Time profiling for the matrix inverse function:
Forward elimination: 1.064s (52.84%)
Backward elimination: 0.9498s (47.16%)
Inverse function total: 2.014s (1)
-----------------------------------
Iteration 1 residual: 5.779e+05
Iteration 2 residual: 5.651e+04
Iteration 3 residual: 1.595e+04
Iteration 4 residual: 2754
Iteration 5 residual: 666.5
Iteration 6 residual: 135.7
Iteration 7 residual: 28.84
Iteration 8 residual: 6.704
Iteration 9 residual: 1.267
Iteration 10 residual: 0.329
Iteration 11 residual: 0.05684
Iteration 12 residual: 0.01592
Iteration 13 residual: 0.002603
Iteration 14 residual: 0.0007524
```

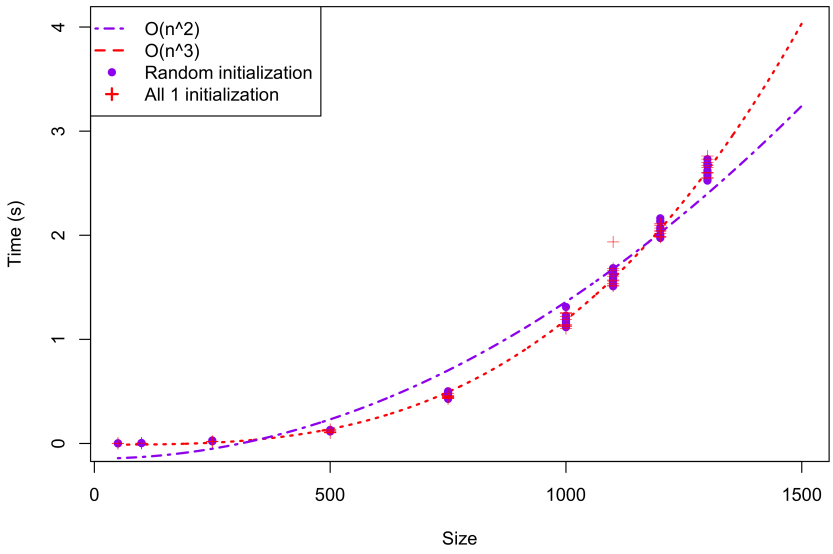

Figure 4: Execution time projection for the Gauss-Seidel iterative method.

```
Total time for the iterative method: 2.167s
```

Figure 4 shows the execution time projection for the Gauss-Seidel iterative solver. The figure contains both tests for all-1 and random initialization. The purple points indicate random initialization case and the red plus signs indicate all-1 initialization case. Results suggest that there is not a significant difference in execution time between the two initialization methods. Since two methods perform fairly comparable to each other and the regression lines are very similar to each other, only the regression lines for the random initialization case are shown in the figure. It shows that the algorithm complexity lies between $O(n^3)$ and $O(n^2)$. However, please note the absolute execution time is much smaller than that of the direct method.

Space The following output comes from the Valgrind utility provided by the ICS cluster. The leak check functionality is turned on. The Gauss-Seidel iterative method is used to solve a 100-by-100 coefficient matrix. The input file for the matrix is 64 KB, and there is 1 965 KB of memory allocated. The message also shows that there is no possible memory leaks. A conservative projection of the memory needed is that it will require about 20 to 30 times of the size of the input file during the calculation.

```
==27773== Memcheck, a memory error detector
==27773== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==27773== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==27773== Command: ./iterativeSolver.O2 G ../data/A_100.csv ../data/b_100.csv 100 2 0
==27773==
----------------------------------
Time profiling for the matrix inverse function:
Forward elimination: 0.01s (33.33%)
Backward elimination: 0.02s (66.67%)
Inverse function total: 0.03s (1)
----------------------------------
Total time for the iterative method: 0.13s
==27773==
```

```
==27773== HEAP SUMMARY:
==27773==     in use at exit: 0 bytes in 0 blocks
==27773==   total heap usage: 27,780 allocs, 27,780 frees, 1,965,632 bytes allocated
==27773==
==27773== All heap blocks were freed -- no leaks are possible
==27773==
==27773== For counts of detected and suppressed errors, rerun with: -v
==27773== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

2.5 Solver Comparison

After the above tests and profiling for the directive solver and the Jacobi (results not included in the report) and the Gauss-Seidel iterative solver, a brief summary is provided below:

- The direct solver is much more slower compared with the Gauss-Seidel iterative solver. The difference in execution time can be as large as about 10 times.
- The iterative solver uses a bit more memory than the direct solver in the case of the current implementation. However, this might be further optimized.
- The direct solver works for any type of matrix, include non-diagonally dominant matrices. However, there is an assumption for the iterative solver to work which is that the spectrum radius of the coefficient matrix should be less than 1. This assumption might not be fulfilled in real cases.
- For production scale, the Gauss-Seidel iterative solver should be the most competent out of the three because of the lowest absolute execution time and the stable scale. When the problem size gets bigger, the execution time variability of the program stay rather stable.

When put into a consideration under production, both Gauss-Seidal and the direct method have their own limits. The direct method is slow and will not be suitable for large scale computation. Gauss-Seidal performs faster but it consumes more memory and the process is hard to be distributed across multiple nodes. A memory limit might be imposed to this approach when facing large systems.

3 Parallelization

3.1 Parallelization Method and Justification

In parallel computing memory architectures, there are shared, distributed, and hybrid shared-distributed memory [6]. Standards and techniques are developed to suit different architectures. Shared memory architectures allow all processors to access all memories, which are referred to as the global memory. Uniform memory access indicates that all processors are symmetrically connected and they all have the same speed when accessing memories. Non-uniform memory access is achieved by physically linking multiple shared memory machines. In this setting, although memories are shared among processors, each of the processors does not necessarily have equal time accessing time to all memories. The multithreading technique is designed for these types of architectures. Popular techniques including Open Multi-Processing (OpenMP) and pthreads.

In distributed memory architectures, processors have their own local memory and there is no global address space. As a result, communication handles the data sharing among nodes. Popular APIs include Message Passing Interface (MPI), MapReduce, and Partitioned Global Address Space (PGAS).

In this project, OpenMP is used to handle multi-threading tasks on a single node and OpenMPI is used to handle communications between distributed memory machines. Compared with pthreads, OpenMP provides much easier interfaces for parallelization. It provides directives which take in charge of thread management. Although pthreads can define more complicated tasks for each thread, the major computation

in the linear system that we currently have lies in for-loop style computation. Minimal modification to the code is required in order to use OpenMP parallelization. MPI is by far the de facto industrial standard for distributed computing models. MapReduce can be viewed as a subset of MPI because MapReduce is designated to data intensive computation on distributed memory machines. With the MPI standard, users can have lower level control of different processors in terms of data sharing and synchronization which could be very helpful when our problem is scaled up and different partition scheme might need to be tested. OpenMPI is arguably the most popular community-maintained implementation of the MPI standard. A large community and a long history leads to a large collection of tutorials and example codes which make the learning process faster and easier.

3.2 Matrix Setup

Modification to the iterative Jacobi solver matrix setup is carried out in order to simplify the data communication between nodes.

Let the linear system be $Ax = b$, D be the diagonal matrix, k be the iteration number, and x be the solution after each iteration. The iteration scheme used in the serial code is as follow:

$$x_{k+1} = D^{-1} * (b - (A - D) * x_k) \tag{1}$$

By rearranging the terms, we can separate the error computation and the solution computation as follow:

$$\Delta = x_{k+1} - x_k = D^{-1} * (b - A * x_k) \tag{2}$$

And we can simplify the computation even further:

$$D * \Delta = b - A * x_k \tag{3}$$

By doing this modification to the iteration scheme, minimal communication is needed among nodes for only the matrix A and the vectors b and x. This also makes the partition easier. Within each loop, the error term is first calculated and then the new solution can be easily calculated. If the error is below a certain threshold, the iteration will terminate.

3.3 Parallelization Implementation

3.3.1 Multi-Threading Parallelization

OpenMP is used to implement multi-threading parallelization. For loops in the Matrix library have been parallelized for file I/O and data intensive computation. A list of functions that are parallelized is provided below:

- Matrix::readMatrix
- Matrix::inverse
- Matrix::transpose
- Matrix::operator+
- Matrix::operator-
- Matrix::operator*

An example of how multi-threading is implemented is provided in the following example.

Source Code 1: OpenMP parallelization Basic

```
    // Use Gaussian Elimination to solve the system
    //
    // Forward elimination
#if defined(_OPENMP)
#pragma omp parallel default(none) shared(nsize, mat, mat_inv)
#endif
    for (size_t k = 0; k < nsize - 1; k++) {
        if (abs(mat[k][k]) < _ZERO_LIMIT) {
            ostringstream message;
            message << "0 occurs (" << mat[k][k]
                << "). Please use row permutation.";
            throw runtime_error(message.str());
        }

#if defined(_OPENMP)
#pragma omp for schedule(static)
#endif
        for (size_t i = k + 1; i < nsize; i++) {
            double coef = mat[i][k] / mat[k][k];

            for (size_t j = k; j < nsize; j++) {
                mat[i][j] -= mat[k][j] * coef;
            }

            for (size_t j = 0; j < nsize; j++) {
                mat_inv[i][j] -= mat_inv[k][j] * coef;
            }
        }
    }
```

Example Code 1 gives an example of how to use OpenMP to create a pool of threads and then use these threads to parallel a for loop. The first code pragma *omp parallel* creates a thread pool. The number of threads to be created is not specified in the code, and therefore the environment variable OMP_NUM_THREADS is used to determine how many threads will be created. The option *default* is set to none so, by default, there is no shared variables between among threads. This practice forces programmers to think about what data are shared to avoid unexpected data sharing problems. And following that is the option *shared* which explicitly specify the variables to be shared among threads.

The second pragma distribute for loop workload to different threads. The option *schedule* determines how the workload is distributed among threads. Since each iteration is expected to take similar time to finish, a static schdule is used that a fix number of iterations is assigned to each thread.

Although there is an OpenMP pragma *omp parallel for* which combines the functionality of the above two. I used them separately to avoid creating and destroying the thread pool multiple times.

Source Code 2: OpenMP option collapse

```
#if defined(_OPENMP)
#pragma omp parallel for default(none) schedule(static) collapse(2) \
shared(lhs, mat_add, rhs)
#endif
        for (size_t i = 0; i < lhs.nrows_; i++) {
            for (size_t j = 0; j < lhs.ncols_; j++) {
                mat_add[i][j] = lhs[i][j] + rhs[i][j];
```

```
        }
    }
```

Another useful OpenMP pragma option is *collapse* as shown in the example Code 2. If two for loops are perfectly nested, this option can be used to reduce the granularity of work to be done by each thread which should provide a better performance.

3.3.2 MPI Parallelization

OpenMPI is used to implement multi-node parallelization. The main technique used here is one-sided communication, which can be also referred as Remote Memory Access (RMA). A node can create a designated memory, called window, that is exposed to all the other nodes for reading and writing as long as the nodes that need to communicate with have the same window. An example is provided in the following source code.

Source Code 3: MPI One-Sided Communication

```
// Collectively declare memory as remotely accessible
MPI_Win win_A_data, win_A_nrows, win_A_ncols, win_solution_data,
        win_errors_data, win_b_data, win_resid;
...

if (world_rank == 0) {
    ...
    MPI_Win_create(&(p_cm_A->nrows), sizeof (int), sizeof (int),
            MPI_INFO_NULL, MPI_COMM_WORLD, &win_A_nrows);
    ...
} else {
    MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &win_A_nrows);
    ...
}

int nrows = -1, ncols = -1, nrows_to_read = -1, row_start = -1, row_end = -1;

// Get number of rows from remote memory
MPI_Win_fence(0, win_A_nrows);
MPI_Get(&nrows, 1, MPI_INT, 0, 0, 1, MPI_INT, win_A_nrows);
MPI_Win_fence(0, win_A_nrows);
```

Example Code 3 shows how to create a window and read from the windows on a different node. First, a *MPI_Win* is created for all processes and the data host will expose this window with the data pointer while the other processes simply declare a NULL pointer with this window. Fence synchronization is the simplest RMA synchronization pattern and most closely resembles the *MPI_Barrier* call. It is required before and after the calls to windows, to quote from the *MPI_Win_fence* man page:

> Calls to MPI_Win_fence should both precede and follow calls to put, get or accumulate that are synchronized with fence calls.

Although exact percentage for how much code is parallelized is hard to calculate, I approximate the proportion using the the number of lines parallelized. In the file *Matrix.cpp* with serial implementation, there are in total 347 lines of codes. Roughly 85 lines have been parallelized. The percentage is 24.5%. Note that this percentage does not provide any information for the speed up from serial to parallel Jacobi because this percentage is from the Matrix library. A more informative percentage is calculated from the file *parallelJacobi.cpp*. In the function *runJacobi* in this file, although roughly 80% percent of the code has been

parallelized, a big chunk of code has then be added in order to facilitate this parallelization which might compromise some of the benefit brought from parallelization.

4 Profiling

4.1 Serial

Serial program profiling is carried out on a Mac Air with a 2.2GHz Intel i7 core and 8 GB 1600 MHz DDR3 memory. The following standard output messages show what processes are profiled and logged. The Jacobi iterative method is used to solve a 250 * 250 matrix with all-1 initialization. The algorithm converges within 5000 iterations.

```
----------------------------------
Time profiling for matrix inversion:
Forward elimination: 0.1526s (53.85%)
Backward elimination: 0.1307s (46.15%)
Inverse function total: 0.2833s (1)
----------------------------------
(Jacobi) Preprocessing: 0.2872s (3.469%)
(Jacobi) Loop: 7.99s (96.53%)
(Jacobi) Postprocessing: 2e-06s (2.416e-05%)
(Jacobi) Total time: 8.277s (100%)
Reading data: 0.07591s (0.9088%)
Iterative method: 8.278s (99.09%)
Total time: 8.354s (100%)
```

Several routines are profiled. The information between the above dotted lines are generated from the matrix inversion function which has been called for only once during the Jacobi preprocessing subroutine. The preprocessing subroutine includes matrix definition, inversion, and initialization. The postprocessing subroutine includes checking whether the coefficient matrix is diagonally dominant if the algorithm did not converge. If the algorithm converged, the postprocessing subroutine will take negligible amount of time as shown in the above case. All three subroutines (preprocessing, loop, and postprocessing) belong to the routine iterative method. The higher-most level contains two routines, reading data and iterative method.

Figure 5 shows the profile time for each subroutine in serial Jacobi solver with different data sizes. The data size indicate the size of the A matrix. Experiments are repeated for 10 times for statistical accuracy.

The loop subroutine take major chunk of the execution which is expected. The preprocessing subroutine is executed only once for each Jacobi solver call, and the major chunk of time in this subroutine is taken by the matrix inversion function (not shown in the figure). The scaling of the serial Jacobi solver is a little bit worse than linear.

Some potential steps that would probably increase the efficiency include:

- Use a different initialization method;
- Change operator functions in the Matrix library to inline functions;
- Reduce the number of function calls by changing the Matrix library into element-wise calculation without calling functions;

Since the first two steps can be easily done together, operator functions for addition, subtraction, and product are converted to inline functions. Inline functions will save function call time in sacrifice of getting a bigger execution binary file. I also used the knowledgeable guess for initialization which would decrease the number of iteration until convergence. Both methods should reduce the execution time of serial Jacobi method.

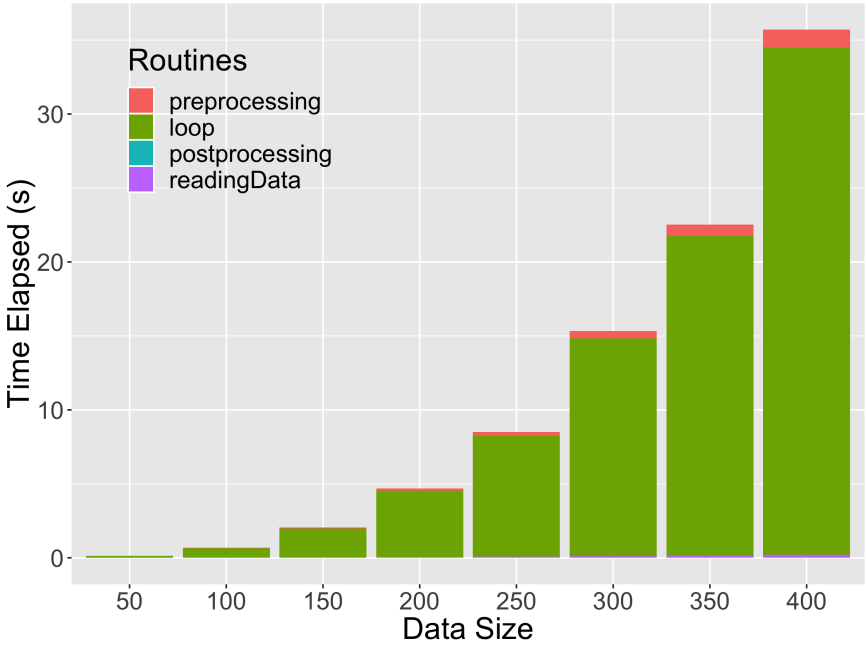

Figure 5: Time profiling for subroutines in serial Jacobi solver.

Figure 6 shows similar relationships between the execution time and different data sizes. However, the absolute execution time is significantly reduced, observed by the y-axis. Because of reduction in for loop computation, a larger proportion of time is spent on preprocessing. A detailed comparison table is provided below.

Figure 7 shows the total execution time comparison between the original and the improved Jacobi solver. After the modification steps, the speedup of the program is larger than 1.05, and the general trend is positive with the increase of data sizes. This suggests the modification is very successful.

4.2 Parallel

OpenMP parallelization profiling is carried out on a Dell Precision 7920 desktop tower workstation. It has an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz with 16 physical cores and 64 GB of memory. The following profiling experiments are carried out using a 250*250 coefficient matrix. A smaller matrix might not be sufficient to inspect the differences when using different cores because it takes too short time to finish. Experiments are run with 1, 2, 4, 8, 16, and 32 cores. Experiments are repeated for 10 times for statistic accuracy.

Figure 8 shows the wall time and CPU time for the parallel Jacobi solver. The x axis marks the combination of the clock type and the number of cores used. CPU time indicates the internal clock frequency during the execution of a program. This resembles the "total" time used by all the threads which sums up the execution time from each of the threads. Therefore, as more threads are used, longer CPU time is observed. CPU time can also indicate the amount of total work done by the program. When more threads are used, the amount of total work increases which associates with the additional work of creating and managing multiple threads. Wall time indicates the actual time elapsed in the real world. When only 1 thread is used, the wall time equals to the CPU time. when more threads are used, wall time should theoretically decrease proportionate to the number of cores. However, this is not observed from the results. The decrease of wall time becomes negligible when more than 8 cores are used. Two main reasons lie behind it. First, not all the code is parallelizeable. Therefore the decrease of wall time is not linearly related to the number of cores used. Second, the additional work introduced by multi-threading offsets the benefit of parallelization.

For both the CPU and wall time, loop takes up most of the time. This is expected from the implementation

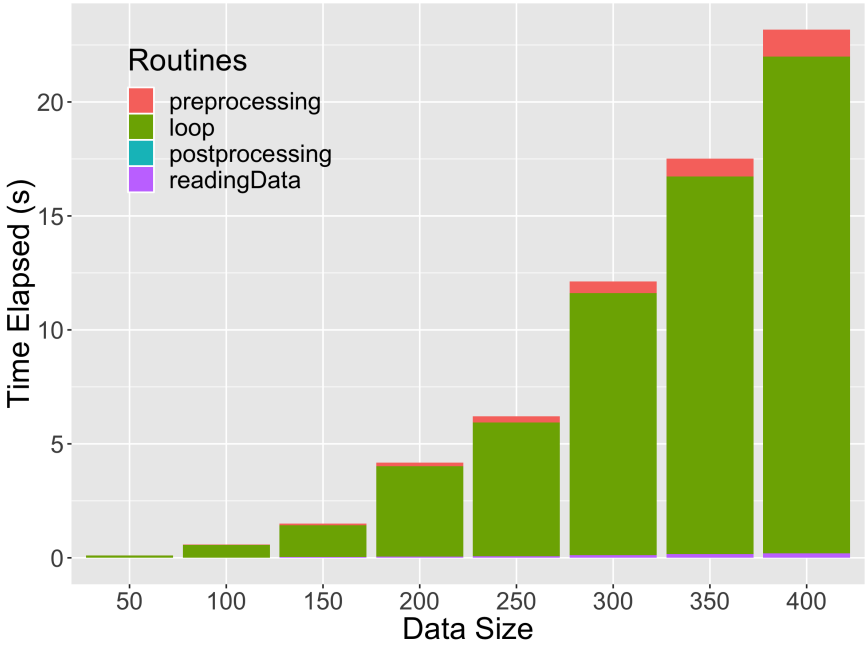

Figure 6: Time profiling for subroutines in serial Jacobi solver with inline functions and best guess initialization.

because most of the computation happens during the iterative process, which is included in loops. The experiments are carried out using the data size of 250 * 250. The serial code takes about 6.21 seconds, as shown in Figure 7 . In Figure 8, the parallel code using 2, 4, and 8 cores used 3.019, 2.268, 1.986 seconds respectively. However, the performance was expected to be better than this. By looking at the growth of the CPU time, my initial guess is that the growth of OpenMP overhead is too fast. When using more than 8 cores, the growth of the total workload is by multiplication of 2. When there is little growth in the total workload, for example, when using fewer than 8 cores, the performance is as good as expected.

To compare the memory usage of the parallel Jacobi solver to the serial Jacobi solver described in the progress report #1, the tool valgrind is used to track memory usage. The parallel Jacobi solver is used to solve a linear system with a 100*100 coefficient matrix. This size is the same with the one used in the progress report #1. 8 threads are created. The output messages are provided below.

Source Code 4: Valgrind messages for the OpenMP Jacobi Solver

```
$ OMP_NUM_THREADS=8 valgrind iterativeSolver J A_100.csv b_100.csv 7000 3 0
==18281== Memcheck, a memory error detector
==18281== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==18281== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==18281== Command: iterativeSolver J A_100.csv b_100.csv 7000 3 0
==18281==
---------------------------------
Time profiling for matrix inversion:
Forward elimination: 1.372s (34.51%)
Backward elimination: 2.604s (65.49%)
Inverse function total: 3.976s (1)
---------------------------------
(Jacobi) Preprocessing: 4.084s (4.586%)
(Jacobi) Loop: 84.97s (95.41%)
(Jacobi) Postprocessing: 0.000662s (0.0007433%)
```

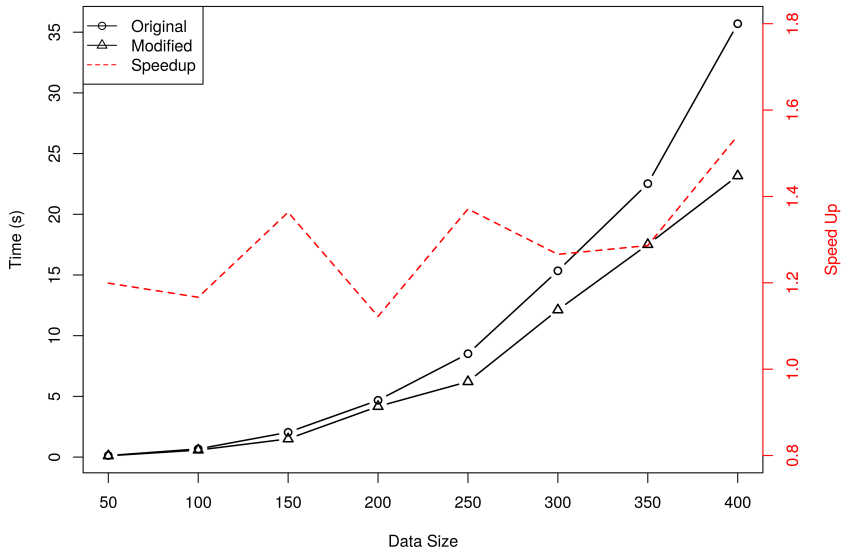

Figure 7: Time profiling for subroutines in serial Jacobi solver with inline functions and best guess initialization.

```
(Jacobi) Total time: 89.06s (100%)
(Jacobi) Wall time Preprocessing: 3.028s (4.366%)
(Jacobi) Wall time Loop: 66.34s (95.63%)
(Jacobi) Wall time Postprocessing: 0.0004167s (0.0006008%)
(Jacobi) Wall time Total time: 69.37s (100%)
Reading data: 0.4197s (0.4689%)
Iterative method: 89.08s (99.53%)
Total time: 89.5s (100%)
Wall time Reading data: 0.4088s (0.5858%)
Wall time Iterative method: 69.39s (99.41%)
Wall time Total time: 69.8s (100%)
==18281==
==18281== HEAP SUMMARY:
==18281==     in use at exit: 5,472 bytes in 11 blocks
==18281==   total heap usage: 569,569 allocs, 569,558 frees, 16,960,352 bytes allocated
==18281==
==18281== LEAK SUMMARY:
==18281==    definitely lost: 0 bytes in 0 blocks
==18281==    indirectly lost: 0 bytes in 0 blocks
==18281==      possibly lost: 2,128 bytes in 7 blocks
==18281==    still reachable: 3,344 bytes in 4 blocks
==18281==         suppressed: 0 bytes in 0 blocks
==18281== Rerun with --leak-check=full to see details of leaked memory
==18281==
==18281== For counts of detected and suppressed errors, rerun with: -v
==18281== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Compare to the progress report #1, the serial Jacobi method allocated 1,505,606 bytes to solve a 100*100 matrix. For the parallel Jacobi method, 16,960,352 bytes are allocated to solve a system with the same size. The memory requirement increased ten-fold. This is expected because parallel programs tend to take more

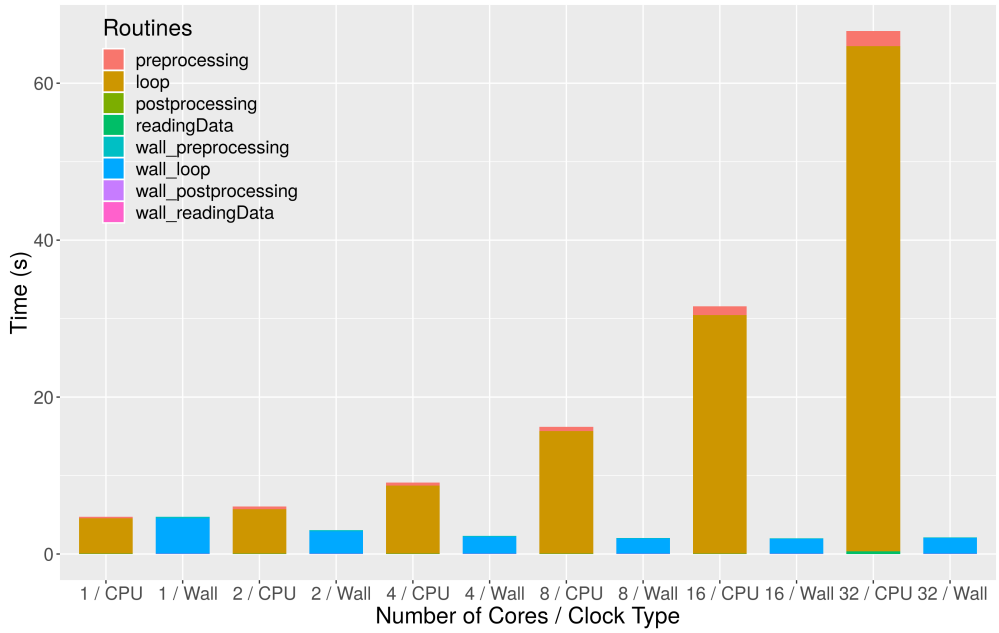

Figure 8: Time profiling for subroutines in parallel Jacobi solver with OpenMP.

memory for data sharing and thread management.

To improve the performance of the OpenMP Jacobi solver, the biggest problem is to reduce the growth of total amount of workload. This is associated with thread generation and management. A list of potential steps are listed below:

- Reduce the number of function calls;
- Avoid creating and killing threads within each loop;

The second change is carried out in the program. In the original implementation, for example, of the operator + shown in the example Code 2, every time the addition function is called, threads are generated specifically for the for loops contained in this function and then killed. If multiple matrix operators are called in a line of code, threads are created and killed multiple times which can cause a significant amount of additional work associated with threads.

Source Code 5: Element-wise Calculation with OpenMP

```
#if defined(_OPENMP)
#pragma omp parallel default(none) \
shared(max_it, small_resid, D_inv, b, R, A, verbose, solution_new, \
resids, solution, resid_metric, cout)
#endif
    {
        int thread_num = omp_get_thread_num();
        for (size_t i_it = 0; i_it < max_it && resid_metric > small_resid; i_it++) {

            if (thread_num == 0) {
                solution = solution_new;
            }

#pragma omp barrier
```

```
#pragma omp for schedule(static)
            for (int i_row = 0; i_row < solution_new.nrows(); i_row++) {

                double sum = 0.0;
                for (int i_col = 0; i_col < R.ncols(); i_col++) {
                    sum += R[i_row][i_col] * solution[i_col][0];
                }
                solution_new[i_row][0] = b[i_row][0] - sum;
            }
#pragma omp barrier
#pragma omp for schedule(static)
            for (int i_row = 0; i_row < solution_new.nrows(); i_row++) {
                double sum = 0.0;
                auto solution_old = solution_new;
                for (int i_col = 0; i_col < D_inv.ncols(); i_col++) {
                    sum += D_inv[i_row][i_col] * solution_old[i_col][0];
                }
                solution_new[i_row][0] = sum;
            }
            //solution_new = D_inv * (b - R * solution);
#pragma omp barrier
#pragma omp for schedule(static)
            for (int i_row = 0; i_row < resids.nrows(); i_row++) {
                double sum = 0.0;
                for (int i_col = 0; i_col < R.ncols(); i_col++) {
                    sum += A[i_row][i_col] * solution_new[i_col][0];
                }
                resids[i_row][0] = sum - b[i_row][0];
            }
            //resids = A * solution_new - b;
#pragma omp barrier

            if (thread_num == 0) {
                resid_metric = accumulate(resids.begin(), resids.end(), 0.0, [](
                            const double lhs, const vector<double> rhs) {
                        return (lhs + abs(rhs[0]));
                        });

                if (verbose >= 2) {
                    cout << "Iteration " << i_it + 1 << " residual: "
                         << resid_metric << endl;
                }
            }
#pragma omp barrier
        }
    }
```

Example Code 5 shows the proposed step to improve the efficiency of OpenMP parallelization. Matrix operators are avoided by using the element-wise calculation. Threads are created only once before the iteration for loop, and then used to parallelize the sub for loops. This parallel regime avoids creating and killing thread pools within each iteration. However, a number of barriers are inserted to synchronize the

reading and writing process. This might cause overhead.

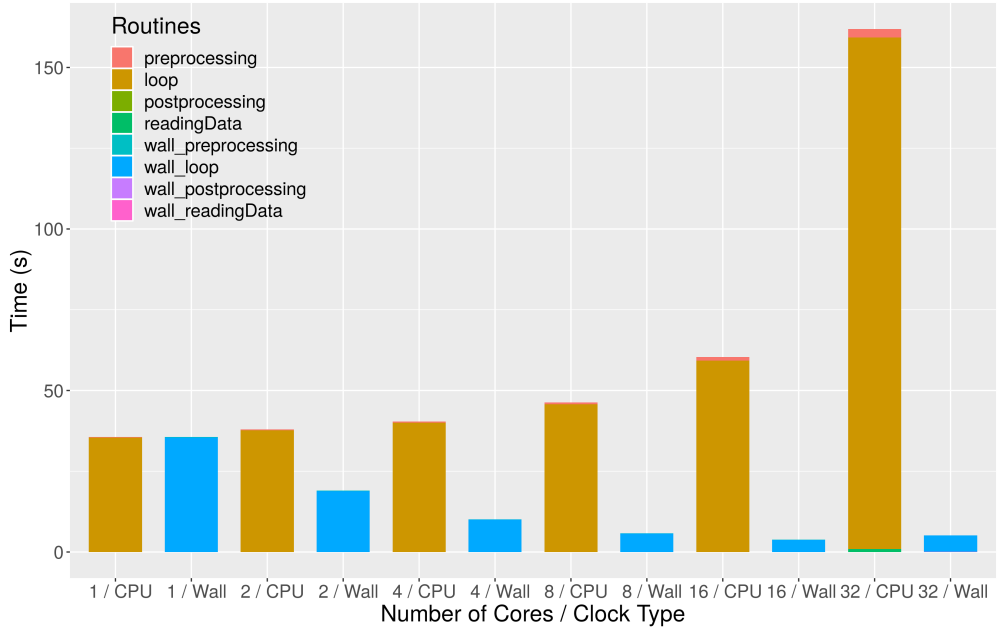

Figure 9: Time profiling for subroutines in parallel Jacobi solver with the proposed improvement step.

Figure 9 shows that with the proposed step to improve OpenMP efficiency, the program becomes much more slower than the original implementation when using only one thread. When using fewer than 32 cores, the modified program scales better than the original program. The total amount of workload does not grow as fast as the original program, and therefore the decrease of wall time of the modified program is closer to the theoretical improvement. The amount of workload roars when 32 cores are used. The reason is unclear but it seems like this is related to the number of threads created and the memory usage.

4.3 Strong Scaling

The previous section shows the profiling of OpenMP implementation, and basic strong scaling experiments and figures are shown. In this section, the strong scaling effect of the OpenMPI program is demonstrated.

The data size is 500*500. This is larger than the data size used in the OpenMP profiling because OpenMP is designed for shared memory and OpenMPI is designed for distributed memory. OpenMPI is expected to scale better when larger data are used. The starting number of processes is 1.

Table 1: Strong Scaling of the MPI Jacobi Solver

Process(es)	User	Real	System
1	31.66425	31.87700	0.11400
2	49.84175	25.16350	0.13900
4	87.34800	22.09275	0.18475
8	174.15200	22.04900	0.33300
16	378.91725	23.96800	0.51450
32	1279.73975	40.75325	1.46675

Table 1 shows the time profiling for the MPI Jacobi solver. Experiments are repeated 5 times and the average is shown. The results are shown in a table format rather than in a figure because of the scale of the plot will be skewed towards the increase of the user time. The decrease of real time cannot be clearly seen from a figure.

Similar patterns to the OpenMP profiling can be found here in the profiling of MPI. When fewer than 8 processes are used, the program stays pretty efficient and the amount of total work does not increase. However, when more than 8 processes are used, the amount of work increased dramatically and the benefit of OMP paralleization is offset.

From the Table 1, 8 processors would give a fastest answer for an input size of 500*500. 2 processors would give an answer with the highest efficiency. For my research, I would use 2 processors.

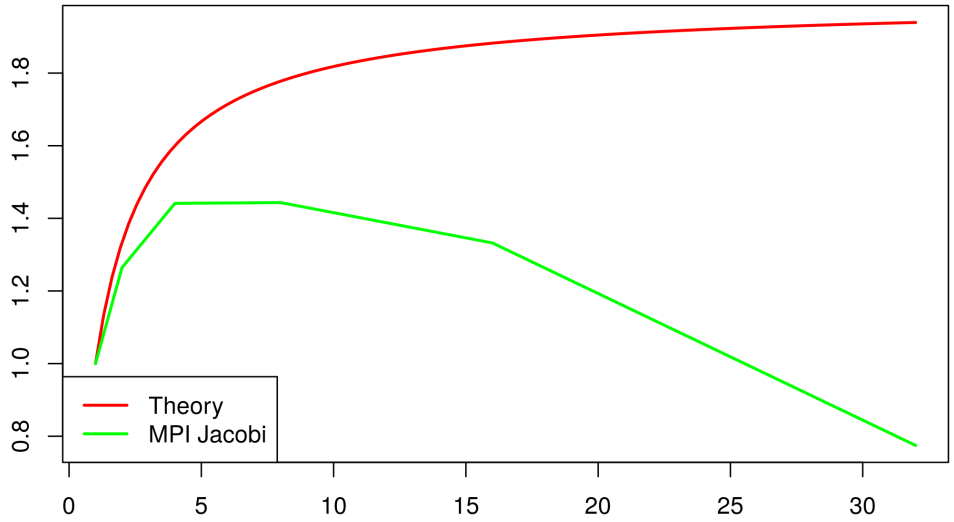

Figure 10: Comparison between Amdahl Law and the OMP Jacobi Solver.

Figure 10 shows the comparison between the "ideal" case expected from the Amdahl's Law and the real case of the MPI Jacobi solver. Results show that the program is not close to the theoretical optimum. When more than 10 cores are used, the benefit from parallelization is offset by the overhead.

5 Coupling to an External Library

5.1 Coupling

Figure 10 shows that using the parallel Jacobi solver does not scale very well with more than 5 processes. There are several factors that limit the performance. First is the *Matrix* library. At the beginning, this library is created for the purpose of convenience. It is really easy to read a matrix from and a file and carry out calculation because file I/O functions and operators have been defined and overloaded. These features make the code very clean and readable. The *Matrix* library functions can be parallelized without any changes from the client code. However, when shifting to shared memory systems and MPI parallelization, the *Matrix* library becomes harder to parallelize. The biggest problem lies in file I/O. It is not trivial to partially read data from a CSV file. To prevent recreating wheels, the program is modified to couple the NetCDF library.

NetCDF is a self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. It uses a binary format so it optimizes both the file I/O performance and the file size. NetCDF provides interfaces to various languages including Java, C/C++, and Fortran. NetCDF-4 provides parallel file access to both classic and NetCDF-4/HDF5 files. The parallel I/O to NetCDF-4 files is achieved through the HDF5 library while the parallel I/O to classic files is through PnetCDF. A few functions have been added to the NetCDF C API to handle parallel I/O.

Another popular file format in meteorology is GRIB. However, not many libraries and document are found for GRIB file I/O. Therefore, NetCDF is by far the most common and friendly scientific data format for users.

Second limit to the performance relates to the fenced functions used to achieve RMA functionalists. However, these fences create too many barriers to the program and, as a result, it ultimately slows down the process. The parallel Jacobi method is therefore modified to use *MPI_Bcast* and *MPI_Gather* to handle the data communication between processors.

As a result, the original parallel Jacobi method is refactored based on the two guidelines above. The following contents focus on the implementation of the MPI iterative Jacobi version 2 (link).

Source Code 6: MPI Implementation with distributed data

```
for (size_t i_it = 0; i_it < max_it && resid > SMALLVAL; i_it++) {

    MPI_Bcast(px, size, MPI_DOUBLE, master_rank, MPI_COMM_WORLD);

    // Each process carries out their own computation
    for (size_t i = 0; i < count[1]; i++) {
        for (size_t j = 0; j < count[0]; j++) {
            ptmp[i] += pA[j * count[1] + i] * px[j];
        }

        ptmp[i] = D_inv[i] * (pb[i] - ptmp[i]);
    }

    MPI_Gatherv(ptmp, count[1], MPI_DOUBLE, pdeltax, recvcounts, displs,
            MPI_DOUBLE, master_rank, MPI_COMM_WORLD);

    if (world_rank == master_rank) {
        // Print the sum of residuals
        resid = accumulate(pdeltax, pdeltax + size, 0.0, [](
                const double lhs, const double rhs) {
            return (lhs + abs(rhs));
        });

        if (verbose >= 2)
            cout << "Iteration # " << i_it + 1 << " residual: " << resid << endl;

        // Calculate the new solution
        transform(px, px + size, pdeltax, px, plus<double>());
    }

    MPI_Bcast(&resid, 1, MPI_DOUBLE, master_rank, MPI_COMM_WORLD);

}
```

Example Code 6 shows the computation kernel for processes with MPI only. Each process only computes the solutions for its own rows of data. At the beginning of each iteration, the solution is distributed by the master process using *MPI_Bcast* and all workers are synchronized with the updated solution. At the end of individual computation, *MPI_Gatherv* gathers the separate solutions from workers to the master process to construct the updated solution. The master solution will then determine whether the iterative process should go on.

This is a typical *MapReduce* problem where data are distributed across processes and each process will return a partial solution to the master process. The parallelization scheme with *MPI_Bcast* and *MPI_Gather* turns out to be much more efficient than *MPI_Send-MPI_Recv* and RMA with fenced functions (you can

find the implementation in the file parallelJacobi.cpp), both in terms of time and memory performance. Fenced functions like *MPI_Get* do impose extract barriers to the code. It sacrifices code performance in return of convenience. But in fact, the code becomes large because of the introduced fences. Therefore, using *MPI_Bcast* and *MPI_Gather* when possible seems to be a good practice.

Source Code 7: MPI Implementation with distributed data

```
// Read NetCDF file meta data
int ncid = -1, dimid = -1, varid = -1, res = -1;
res = nc_open_par(nc_file.c_str(),
    NC_NOWRITE, MPI_COMM_WORLD, MPI_INFO_NULL, &ncid); ERR;

// Query size from the nc file
size_t size = 0;
res = nc_inq_dimid(ncid, "size", &dimid); ERR;
res = nc_inq_dimlen(ncid, dimid, &size); ERR;

// Define the rows to read for this process
// Note that the first dimension is column and the second dimension is row.
//
start[1] = world_rank * size / world_size;
count[1] = world_rank == world_size - 1 ?
    size - (world_rank * size / world_size) :
    size / world_size;

// Always read the full columns
start[0] = 0;
count[0] = size;

// Read the sampled data
ptrdiff_t stride[2] = {1, 1};
double *pA = new double[count[0] * count[1]]();
double *pb = new double[count[1]]();
double *px_correct;

res = nc_inq_varid(ncid, "A", &varid); ERR;
res = nc_get_vars_double(ncid, varid,
    start, count, stride, pA); ERR;

res = nc_inq_varid(ncid, "b", &varid); ERR;
res = nc_get_vars_double(ncid,
    varid, start + 1, count + 1, stride + 1, pb); ERR;

res = nc_close(ncid); ERR;
```

Example Code 7 shows the code snippet for reading part of the NetCDF file. Because data are distributed by rows, each process will read all the columns and only the designated rows instead of all the data. This approach saves the time to distribute the data by the master process because each process reads their own share from the beginning, and lift the memory limit because the master process reads less data than before.

5.2 Updated Profiling and Scaling

All the following experiments were running using **basic computing nodes** on ICS ACI-B clusters. The resource required for parallel processing includes 4 nodes and 16 processes per node. Each experiment is carried out 3 times for statistical accuracy.

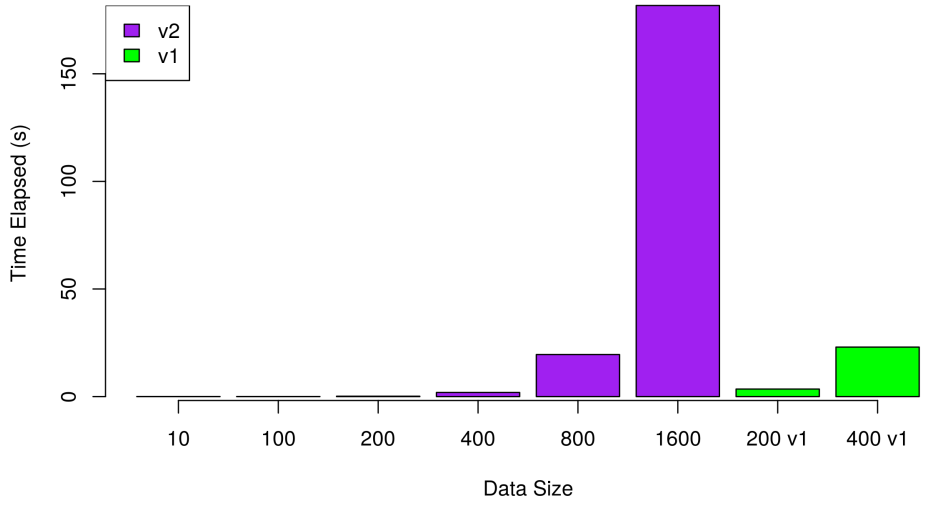

Figure 11: Time profiling for the parallel Jacobi solver version 2.

Time Figure 11 shows the performance of the parallel Jacobi solver version 2 with only one processor (serial). The purple bars show the time for the modified program when running different sizes of example data with only one processor. The green bars show two comparable serial profiling from the original parallel program with sizes of 200 and 400. With the reduced amount of codes and a different file I/O, version 2 achieved better serial performance out of the box. Another factor to this can be that version 2 is not using any matrix libraries and the computation is done element-wise. This avoids a lot of functions calls which would also save time.

Space Figure 12 shows the memory space profiling with the version 2 program. To solve a problem with size 100*100, serial Jacobi allocated 1,505 kb, parallel Jacobi version 1 allocated 16,960 kb, and the version 2 allocated 7,132 kb. Serial Jacobi uses the least memory and version 1 program uses the most. The version 2 program uses more memory than the serial version and less than the version 1 which are expected. More memory should be allocated for parallelization. Due to the element-wise calculation which happens in-place without requiring data copy and extra memory space, less memory is need for the version 2 program. Another improvement from the version 2 program is the scaling of the memory requirement. It almost scales linearly with the file size. This is a huge advantage because this profiling is done per processor. Because data are distributed across multiple processors, the memory requirement for each process will be a lot smaller than the actual data size. This feature enables the program to deal with larger problems.

Strong scaling Figure 13 shows the strong scaling of the version 2 program with different data sizes of 800, 1600, and 3200. Box plots show the distribution of time from repeated experiments. With all sizes, the program scales pretty well when using fewer than 24 processes. This is an improvement compared with the version 1 program when it only scales well using 5 processes. When using fewer than 24 processes, the bigger the problem is, the slope of the decreasing time is steeper, and the better the program scales. Because

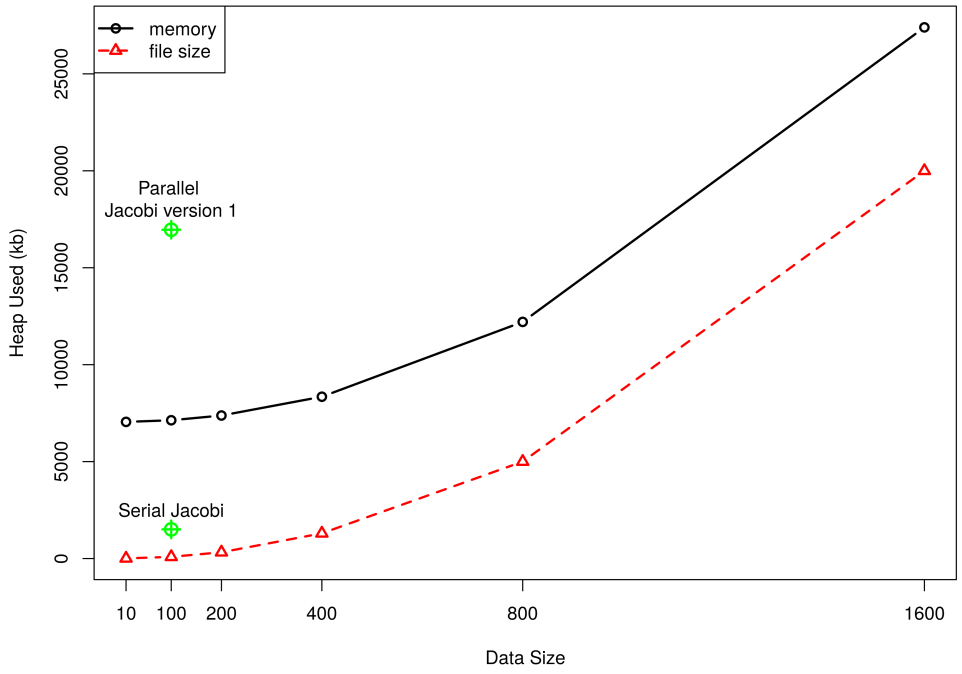

Figure 12: Memory profiling for the parallel Jacobi solver version 2.

each process only reads a portion of the entire data set, and carries out computation using its own share of data, the benefit of parallelization will be more manifest when each process compute a relatively large amount of data. When the subset of the original data set becomes small, the parallelization becomes trivial as well. When using more than 24 processes, the overhead brought by communication outweights the benefit of parallelization and therefore the profiling appears not consistent across different experiments.

Amdahl's Law Figure 14 shows the speed-up comparison when the version 2 program is tested with different data sizes. Similar results can be observed. When using fewer than 24 processes, the larger the data size is, the higher the curves are, and therefore the better the program scales. It is noted that when using about 28 and 40 processes, the program achieved super-scaling. This happens when the data are distributed and each portion of the data can be effectively padded [4] into a lower level of memory caches. Because caches are much faster than memory and lower level of caches tend to be faster than higher level, it is observed that the speed up at 40 processes is higher than that at 28 processes. However, this factor is not observed consistently. When more processes are used, the overhead of synchronization again outweighs the benefit of parallelization.

5.3 Potential Coupling

With the current implementation of the parallel Jacobi solver version 2, the file I/O process is handled by the NetCDF library. There is a possibility to replace this library with the Boost Serialization library. The word "serialization" means the reversible deconstruction of an arbitrary set of C++ data structures to a sequence of bytes which indicates that this library can be used to save theoretically any type of data structures to a binary file. However, more guidance is needed in order to run the library in parallel.

The *MapReduce* parallelization scheme can also be applied to other advanced decomposition methods. In the current implementation with the Jacobi method, the original coefficient matrix is decomposed to the

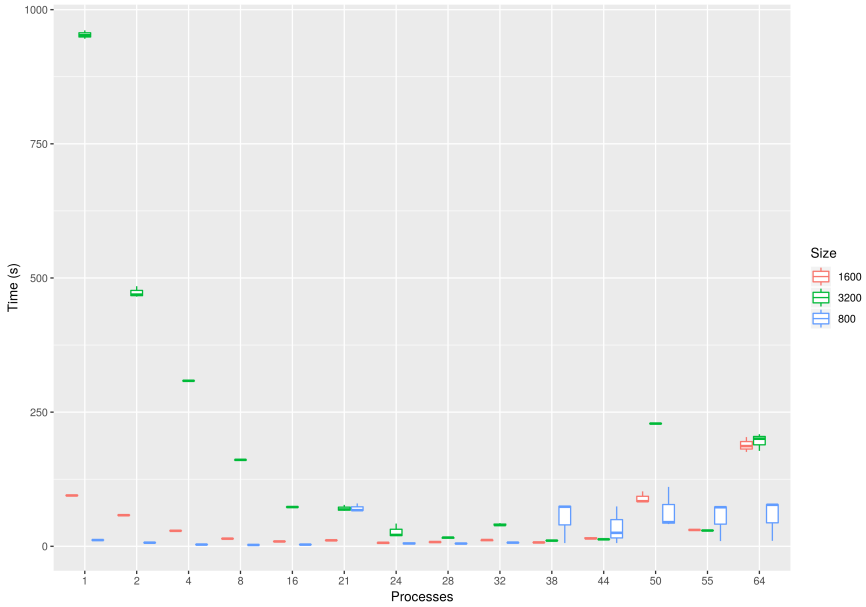

Figure 13: Strong scaling for parallel Jacobi version 2.

diagonal matrix and the remaining matrix. With this approach, the error term can be easily separated to one side of the equation. With LU decomposition, for example, the iteration scheme is $x_{k+1} = (D+L)*(b-U*x_k)$ and it is still applicable to this code though modification is needed to change the computation and equation.

A high level approach is to replace the entire solver code with another solver library. LPACK, BLAS, and Boost all provides functions to store matrices and carry out matrix calculation. The *MapProduce* parallelization scheme can certainly be used to parallelize these solver libraries.

Accelerators, like CUDA GPUs, are similar technologies to OpenMP with directives and pragmas to be inserted into codes. Theoretically, if the program works well with OpenMP, the program should also work well with accelerators like OpenACC. However, with OpenACC, attention needs to be directed to avoid redundant data transfering between GPU and CPU memories because it is very time-consuming. The thoughless use of OpenACC directies might even bring negative effect to the performance of the program.

The next step of the code development is to couple the linear system solver with a different solver library. With the Jacobi iteration method, the solvable matrix is limited to diagonal dominant matrices which only include a subset of matrices. The other matrices can be solved by using normal equations which is hard to parallelize.

The potential coupling with Boost libraries appears to be the most time worthy. Boost include a set of libraries that are well defined with the C++ standard library and are optimized in performance. The Serialization library provides functionality for file I/O with different data types, *uBLAS* is a C++ template class library that provides BLAS level 1, 2, 3 functionality for dense, packed and sparse matrices, and the Thread library facilitate parallelization. The time investment into Boost libraries will be worthwhile because it provides a wide range of possibilities.

6 Discussion and Conclusions

Below is a list of what I have done for the entire project:

- Implemented the direct solver using Gaussian Elimination and the iterative solver using Jacobi method and Gauss-Seidel method;
- Created test scripts and data sets using R. Data sets contain tests with sizes ranging from 5-by-5 to 1300-by-1300 matrices as input to different solvers;

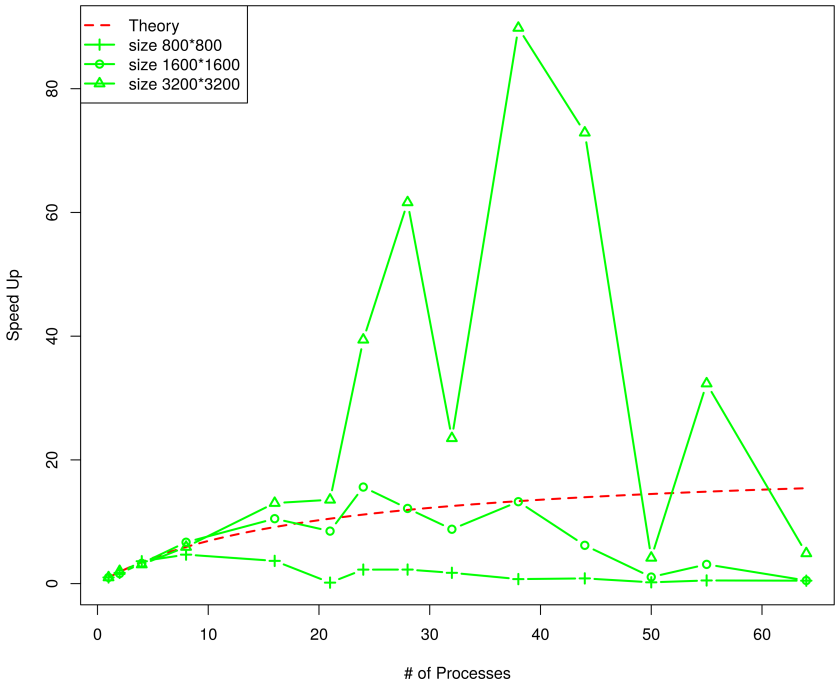

Figure 14: Speed up for parallel Jacobi version 2.

- Carried out time and space profiling both on MacBook Air and the ACI ICS clusters using code-based profiling functions and the Valgrind utility;
- Wrote analysis and visualization scripts in R;
- Conducted writing of this report;
- Parallelize the Matrix library using OpenMP;
- Parallelize the iterative Jacobi solver using OpenMP and OpenMPI;
- Memory and time profiling of the paralle Jacobi solvers;
- Analysis and visulization of the profiling results to identify performance bottleneck;
- Coupling the parallel Jacobi method with NetCDF library;
- Progress reoprt #2, reoprt #1, and the file report;

The optimized parallel Jacobi method appears to be very successful in terms of memory and time profiling. The memory required by the version program reduced from 10x to 2x. The time performance is also 12x better than the version 1 program. Some of the most important factors include: 1) the broadcast-and-gather synchronization is faster than fenced functions with RMA when only a small amount of data are communicated; 2) element-wise computation reduces the amount of memory required by the program and avoids extra copy of data during calculation; 3) parallel access functionality provided by the NetCDF library gives another boost to the performance because file I/O has usually been a bottleneck during optimization. However, constraints still remain in the solution. a lot of the work has been directed to optimize and parallel the heavy for-loop where most of the computation happens, but Jacobi method can be intrinsically slow as a solver compared with the Gauss-Seidal solver. It is used in the program because of the simplicity of the

parallelization of the Jacobi method. However, reduce the number of iteration needed for convergence is another important avenue to optimize the algorithm. Future work includes applying the lessons learned to parallel a different solver. At the current implementation, results are only compared to the correct answers, but not exported to files. In the future, file I/O should be more complete.

To prepare this work for advancing technology, extra attention should be paid to the algorithm per se. For example, with the emergence of exascale computing, even large number of nodes are connected together each with a different computing capability and network bandwidth. As a result, synchronization and resilience becomes a challenging task. It is much more possible to happen when one process is taking tremendously longer time than the others while the workload across them is similar. The program needs to also take node failure into consideration. This can happen due to power outage, resource exhaust, and other unexpected reasons. The popular way to resume a program is by inserting check points and exporting progress files.

Appendices

A Acknowledgements

I would like to express my gratitude to the instructors of CSE 597, Dr. Adam Lavely and Dr. Christopher Blanton for the excellent organization and guidance for the course. I would like to also thank my advisor, Prof. Guido Cervone, for mentoring and guiding me.

B Code

Please find the full code and data sets in the GitHub repository. Please find the source code to the following progress reports:

- Progress report #1 at link.
- Progress report #2 at link.
- Final report at link.

Instructions for compiling and reproducing the results can be found in the *README.md* file. Below is a summary of the folders and files in the repository:

- R/ contains the R scripts for data analysis and visualization.
- cmake/ contains the cmake models to find packages.
- data/ contains the csv data files and the R script used to generate the data files.
- reports/ contains the progress reports in pdf format.
- scripts/ contains miscellaneous scripts for compiling and profiling. They are organized by progress report number.
- poster/ contains the project poster files.
- src/ contains the header and source files for the library Matrix and the several utilities.
- .gitignore is the file specifying which files should be ignored in Git.
- CMakeLists.txt guides CMake to generate a make file. Please see README.md for detailed usage.
- LICENSE.txt is the MIT license.
- README.md contains basic information for the repository and detailed information for how to compile and reproduce the results.

Experiemnts have been run on the following computing nodes:

- ICS ACI basic computing nodes, 2.2 GHz Intel Xeon Processor, 24 CPU/server, 128 GB RAM, 40 Gbps Ethernet;
- Desktop machine Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz;

C Poster

- PDF 24x36 inches highlighting the info in the reports
- Include your contact information
- Science/Problem definition is important
- Give overview of direct/iterative, parallelization (scaling) and library choices
- Acknowledgements/bibliography/resources used
- Published code location

References

[1] Luca Delle Monache, F Anthony Eckel, Daran L Rife, Badrinath Nagarajan, and Keith Searight. Probabilistic weather prediction with an analog ensemble. *Monthly Weather Review*, 141(10):3498–3516, 2013.

[2] Tilmann Gneiting and Adrian E Raftery. Weather forecasting with ensemble methods. *Science*, 310(5746):248–249, 2005.

[3] Thomas M Hamill. Interpretation of rank histograms for verifying ensemble forecasts. *Monthly Weather Review*, 129(3):550–560, 2001.

[4] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J Ramanujam, and Ponnuswamy Sadayappan. Effective padding of multidimensional arrays to avoid cache conflict misses. In *ACM SIGPLAN Notices*, volume 51, pages 129–144. ACM, 2016.

[5] Erich Kaltofen and Gilles Villard. On the complexity of computing determinants. *computational complexity*, 13(3-4):91–130, 2005.

[6] Sol Ji Kang, Sang Yeon Lee, and Keon Myung Lee. Performance comparison of openmp, mpi, and mapreduce in practical problems. *Advances in Multimedia*, 2015:7, 2015.

[7] Edward Lorenz. Chaos in meteorological forecast. *J. Atmos. Sci*, 20:130–144, 1963.

[8] Manideepa Saha. Generalized jacobi and gauss-seidel method for solving non-square linear systems. *arXiv preprint arXiv:1706.07640*, 2017.

[9] Simone Sperati, Stefano Alessandrini, and Luca Delle Monache. Gridded probabilistic weather forecasts with an analog ensemble. *Quarterly Journal of the Royal Meteorological Society*, 143(708):2874–2885, 2017.