Final Project: Progress Report 2

CSE 597

Weiming Hu

Monday, November 7, 2018

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, most of the studies focused on the application of the technique, rather than how to improve and complete the technique. Actually, 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 is usually present in a realistic physical world. To get a better understanding of the problem, a deeper inspection into the ensemble is needed.

This study proposes a matrix approach to analyze the contribution of individual ensemble member to the overall forecast distribution by setting up and solving a linear system. The results can be used to gain better understanding of ensemble members at different locations. The study also has a computational focus that it looks at different solvers and implementation, and tries to optimize the problem solving process.

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 [5] 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 [6]. 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 [6], 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 where 3-dimensional space or a single location exists might not be solved using this technique.

2 Parallelization

2.1 Parallelization Method and Justification

In parallel computing memory architectures, there are shared, distributed, and hybrid shared-distributed memory [4]. 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.

2.2 Parallel Implementation

2.2.1 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.

2.2.2 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.

2.2.3 Percentage of The Parallel Code

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.

2.3 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 Profiling

3.1 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 1 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.

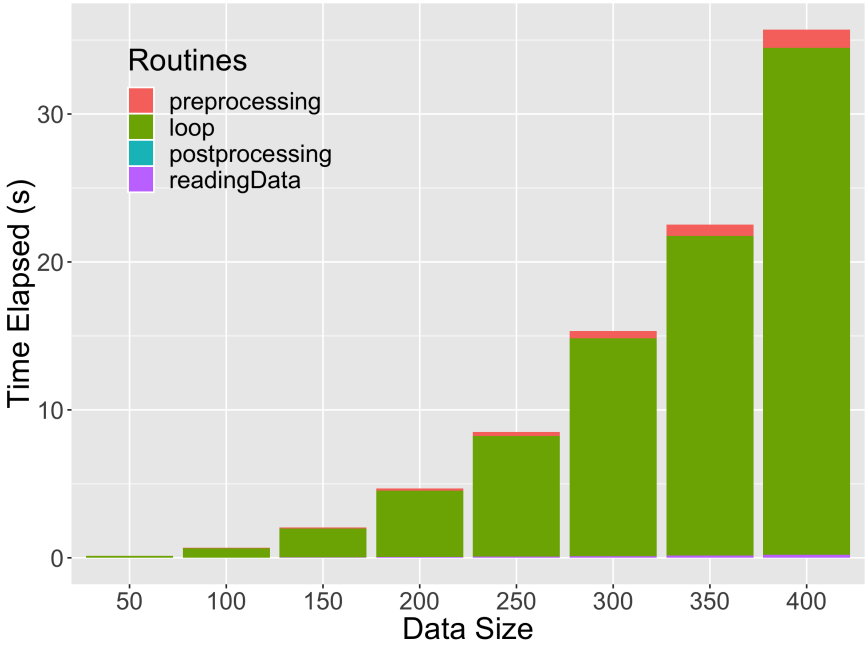

Figure 1: Time profiling for subroutines in serial Jacobi solver.

Figure 2 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.

Table 1: Serial Jacobi Solver Profile with/without Modification

Data Size	**Original (s)**	**Improved (s)**	**Saved (s)**	**Speed Up (%)**
50	0.12592	0.104998	0.020922	16.615
100	0.67309	0.576980	0.096110	14.279
150	2.04280	1.498300	0.544500	26.655
200	4.67890	4.169400	0.509500	10.889
250	8.51510	6.211200	2.303900	27.057
300	15.34400	12.121000	3.223000	21.005
350	22.52300	17.509000	5.014000	22.262
400	35.69900	23.169000	12.530000	35.099

Table 1 shows the total execution time comparison between the original and the improved Jacobi solver. The speed up is significant suggesting that it is very helpful to carry out the two potential modification steps.

3.2 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 3 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

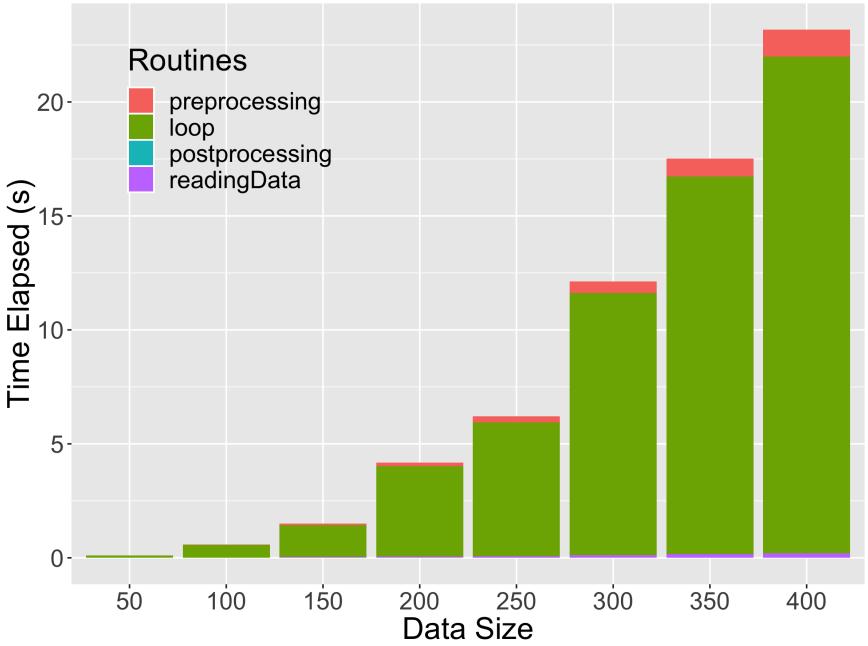

Figure 2: Time profiling for subroutines in serial Jacobi solver with inline functions and best guess initialization.

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 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 Table 1. in Figure 3, 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
```

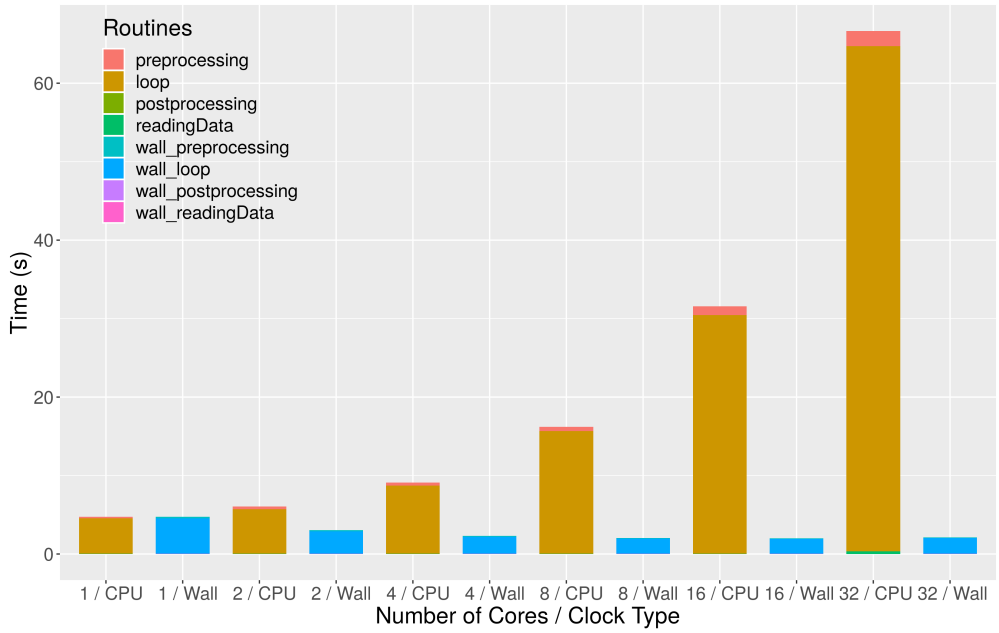

Figure 3: Time profiling for subroutines in parallel Jacobi solver with OpenMP.

```
==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%)
(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 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 4 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.

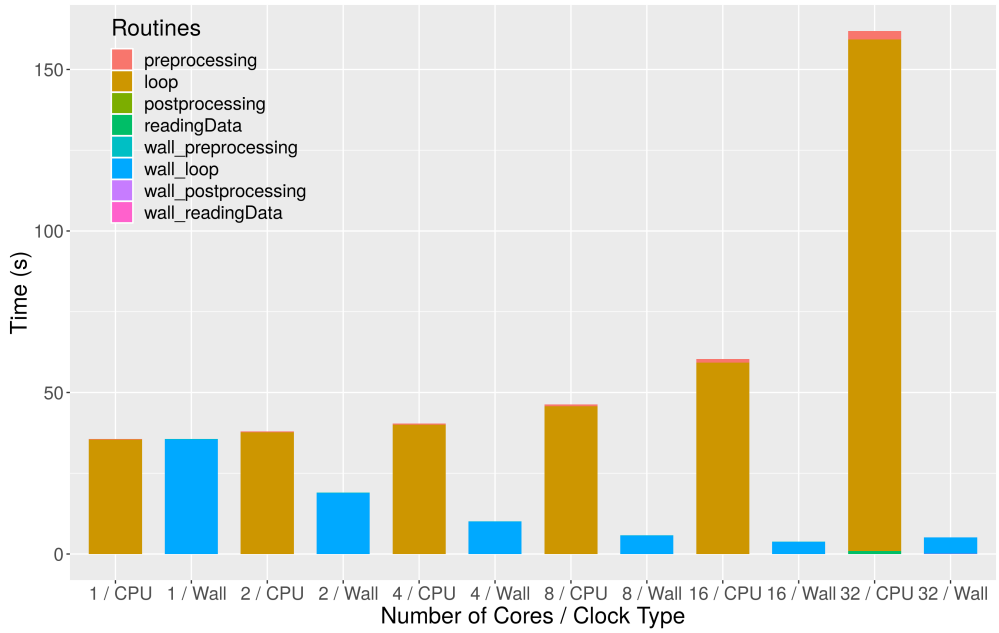

Figure 4: Time profiling for subroutines in parallel Jacobi solver with the proposed improvement step.

4 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 2: 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 2 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 2, 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 5 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 paralleization is offset by the overhead. This should be

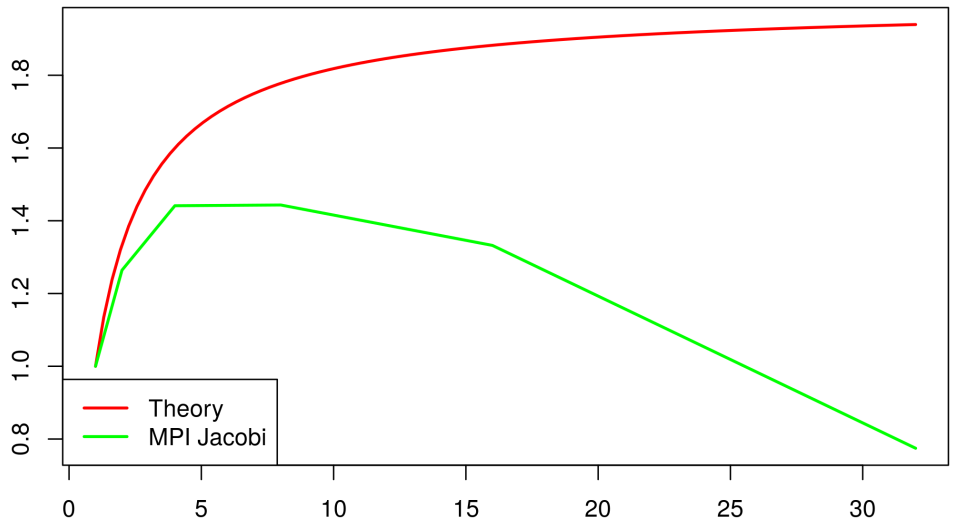

Figure 5: Comparison between Amdahl Law and the OMP Jacobi Solver.

improved in the future.

5 Discussion and Conclusions

Multiple parallelization schemes are implemented and experimented for the Jacobi solver. A list of what has been done is provided below.

- 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;
- Progress reoprt #2;

OpenMP and OpenMPI are used to implement the shared-memory and distributed memory parallelization. The combination of OpenMP and MPI is a very popular choice among the software engineering because MPI is the de facto industrial standard for multi-node infrastructure and OpenMP is the de facto industrial standard for shared memory infrastructure. The combination of these two techniques can provide the most flexibility and granularity during parallelization. Using OpenMP when possible also simplifies the process of parallelization because OpenMP requires minimal modification to the serial code. Which MPI requires a major redesign of the program, it does not limit the program by memory.

Although OpenMP and MPI provide exciting possibilities for parallelization, the profiling of the parallel programs shows only a small amount of improvement. For small scale parallelization, for example, parallelization using fewer threads or processes, the program scales well. The total amount of workload is constrained and the efficiency maintains. When more threads or processes are used and the problem scales up, the program shows incompetence that heavy overhead occurs due to the parallelization. As a result, the efficiency drops dramatically. This might have multiple reasons. For example, many barries need to be inserted for OpenMPI and OpenMP which would slow down the processes. The Matrix library might be another potential reason in which it makes element-wise parallelization hard to accomplish. But this library makes the file I/O so much easier. I have encountered a problem between a friendly user interface and a

better performance. Future work needs to be directed to these two direction: 1) a balance between a friendly API and a better performance, and 2) efficiency improvement.

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 progress report #2 at Overleaf.

Instructions for compiling and reproducing the results can be found in the *README.md* file. I have compiled the codes using the regular nodes (aci-b), and run the tests on computing nodes (aci-i).

Below is a summary of the folders and files in the repository:

- R/ contains the R scripts for data analysis and visualization.
- 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.

C Poster Draft - separate document

Please find the poster pdf file in the directory poster/.

- PDF 24x36 inches highlighting the info in the reports
- Emphasis on characterization before/after parallelization (and coupling - leave some room)
- Section from PR#1: Direct vs. Indirect solvers
- 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] 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.

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

[6] 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.