Parallel Tridiagonal Matrix Solver Using MPI

Bradley Landherr*

Rensselaer Polytechnic Institute
landhb@rpi.edu

Abstract

After communication improvements the Divide and Conquer algorithm exhibits strong scaling against the serial Thomas Algorithm in the 32 core and under range. With most parallel programs there will eventually be a point when the communication cost of adding more cores begins to outweigh the additional speedup gained and the program will begin to become less efficient, however we did not observe this turning point in our trials due to limited resources on the DRP cluster. Performance of the cyclic reduction algorithm still needs to be investigated.

I. Introduction

In this work we seek to develop a parallel method to solve a tridiagonal matrix. The problem itself is a system of n linear equations of the form $Ax = b$ where A is a tridiagonal matrix:

$$\begin{pmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & 0 & \\ & a_2 & b_2 & c_2 & & \\ & & . & . & . & \\ & 0 & & . & . & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix}$$

The Thomas Algorithm is an efficient method to solve such a system, however it is inherently serial. The algorithm iterates through each row in a forward elimination phase computing the following:

$$c'_1 = \frac{c_1}{b_1} \quad , \quad d'_1 = \frac{d_1}{b_1}$$
$$c'_i = \frac{c_i}{b_i - c'_{i-1}a_i} \quad , \quad d'_i = \frac{d_i - d'_{i-1}a_i}{b_i - c'_{i-1}a_i}$$

Then iterates back through the matrix solving for x at each row:

$$x_n = d'_n \quad , \quad x_i = d'_i - c'_i x_{i+1}$$

Notice how in each phase, the result is dependent on the solution of the row before it, giving the algorithm its serial nature.

*Special thanks to Professor Henshaw and Professor Banks

II. Cyclic Reduction Method

Cyclic reduction consists of the same two phases as the Thomas Algorithm, however the approach is slightly different. Instead of iterating through each row, cyclic reduction seeks to reduce the system to a smaller system with half the number of unknowns at each stage until a system of 2 unknowns is reached. Then the other half of the unknowns (the even indexed rows) are solved for during back substitution.

In each step of the forward reduction phase, odd-indexed equations i are updated in parallel to be linear combinations of equations i, $i+1$, and $i-1$ where $i+1$, and $i-1$ are the equations on the nearest neighbouring active processors. Each processor updates its equation with the following values:

$$m = \frac{a_i}{b_{i-1}} \quad , \quad k = \frac{c_i}{b_{i+1}}$$

$$\begin{aligned} a'_i &= -a_{i-1}m \\ b'_i &= b_i - c_{i-1}m - a_{i+1}k \\ c'_i &= -c_{i+1}k \\ d'_i &= d_i - d_{i-1}m - d_{i+1}k \end{aligned} \tag{1}$$

And in the back substitution phase:

$$x_i = \frac{d'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}$$

III. Divide and Conquer Method

Stefan Bondeli's divide and conquer strategy takes the tridiagonal matrix and divides it into blocks of size kxk where k is the problem size divided by the number of processors being used.

If the tridiagonal matrix is as follows:

$$T = \left(\begin{array}{cc|cc|cc} a_1 & c_1 & & & & \\ b_2 & a_2 & \ddots & & & \\ & \ddots & \ddots & c_{k-1} & & \\ & & b_k & a_k & c_k & \\ \hline & & & b_{k+1} & a_{k+1} & c_{k+1} \\ & & & & b_{k+2} & a_{k+2} \\ & & & & & \ddots \\ & & & & & c_{2k-1} \\ & & & & b_{2k} & a_{2k} \\ \hline & & & & & b_{2k+1} \\ & & & & & \end{array} \begin{array}{cc} & \\ & \\ & \\ & \\ & \\ & \\ & \\ c_{2k} & \\ a_{2k+1} & c_{2k+1} \\ b_{2k+2} & a_{2k+2} \\ & \ddots \\ & c_{n-1} \\ b_n & a_n \end{array}\right)$$

The system will be partitioned as:

$$\left(\begin{array}{c|c|c} T_1 & C_1 & 0 \\ \hline B_2 & T_2 & C_2 \\ \hline 0 & B_3 & T_3 \end{array}\right) \left(\begin{array}{c} \vec{x}_1 \\ \hline \vec{x}_2 \\ \hline \vec{x}_3 \end{array}\right) = \left(\begin{array}{c} \vec{d}_1 \\ \hline \vec{d}_2 \\ \hline \vec{d}_3 \end{array}\right)$$

After partitioning, the first step is to compute the solutions to tridiagonal systems on each individual processor, namely:

$$T_i\vec{y}_i = \vec{d}_i \quad \text{, for} \quad i = 1, 2...p$$

for processor 1 calculate Z_1 as:

$$T_1\vec{z}_1 = \vec{e}_k$$

for processors 2 through p-1, where m = 2p-2:

$$T_p\vec{z_m} = \vec{e}_1$$
$$T_p\vec{z_{m+1}} = \vec{e}_k$$

and for processor p:

$$T_p\vec{z_{2p-2}} = \vec{e}_1$$

where the unit vectors are of size k: $e_1 = \{1, 0, 0, ...0\}$ and $e_k = \{0, 0, 0, ...1\}$
This results in a reduced matrix of size 2p-2 given by:

$$\begin{pmatrix} s_1 & t_1 & & \\ r_2 & s_2 & ... & \\ & ... & ... & t_{2p-3} \\ & & r_{2p-2} & s_{2p-2} \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_{2p-2} \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ ... \\ \beta_{2p-2} \end{pmatrix}$$

where the elements are defined using the results from the previous step:

$$s_i = \begin{cases} \vec{e}_k{}^T\vec{z}_i, & \text{if } i \text{ is odd} \\ \vec{e}_1{}^T\vec{z}_i, & \text{if } i \text{ is even} \end{cases} \quad where\ i = 1, 2, ...2p-2$$

$$r_i = \begin{cases} \vec{e}_k{}^T\vec{z}_{i-1}, & \text{if } i \text{ is odd} \\ \frac{1}{c_{\frac{i}{2}k}}, & \text{if } i \text{ is even} \end{cases} \quad where\ i = 2, 3, ...2p-2$$

$$t_i = \begin{cases} \frac{1}{b_{\frac{i+1}{2}k+1}}, & \text{if } i \text{ is odd} \\ \vec{e}_1{}^T\vec{z}_{i+1}, & \text{if } i \text{ is even} \end{cases} \quad where\ i = 1, 2, ...2p-3 \tag{2}$$

$$\beta_i = \begin{cases} -(\vec{e}_k{}^T\vec{y}_{\frac{i+1}{2}}), & \text{if } i \text{ is odd} \\ -(\vec{e}_1{}^T\vec{y}_{\frac{i+2}{2}}), & \text{if } i \text{ is even} \end{cases} \quad where\ i = 1, 2, ...2p-2$$

Then using the Thomas Algorithm to solve the reduced matrix, $\vec{\alpha}$ is obtained and used to compute the solution $\vec{x}$ by iterating through the processors and computing the following:

$$\vec{x}_i = \begin{cases} \vec{y}_1 + \alpha_1\vec{z}_1, & \text{if } i = 1 \\ \vec{y}_i + \alpha_{2i-2}\vec{z}_{2i-2} + \alpha_{2i-1}\vec{z}_{2i-1}, & \text{if } i = 2, 3, ...p-1 \\ \vec{y}_i + \alpha_{2i-2}\vec{z}_{2i-2}, & \text{if } i = p \end{cases} \tag{3}$$

IV. Results

The results indicate strong scaling, as the parallel efficiency is fairly constant and run time decreases as the number of cores increases with a given problem size. The speedup was calculated as:

$$Speedup = \frac{Thomas\ Algorithm\ Run\ time}{Parallel\ Run\ Time}$$

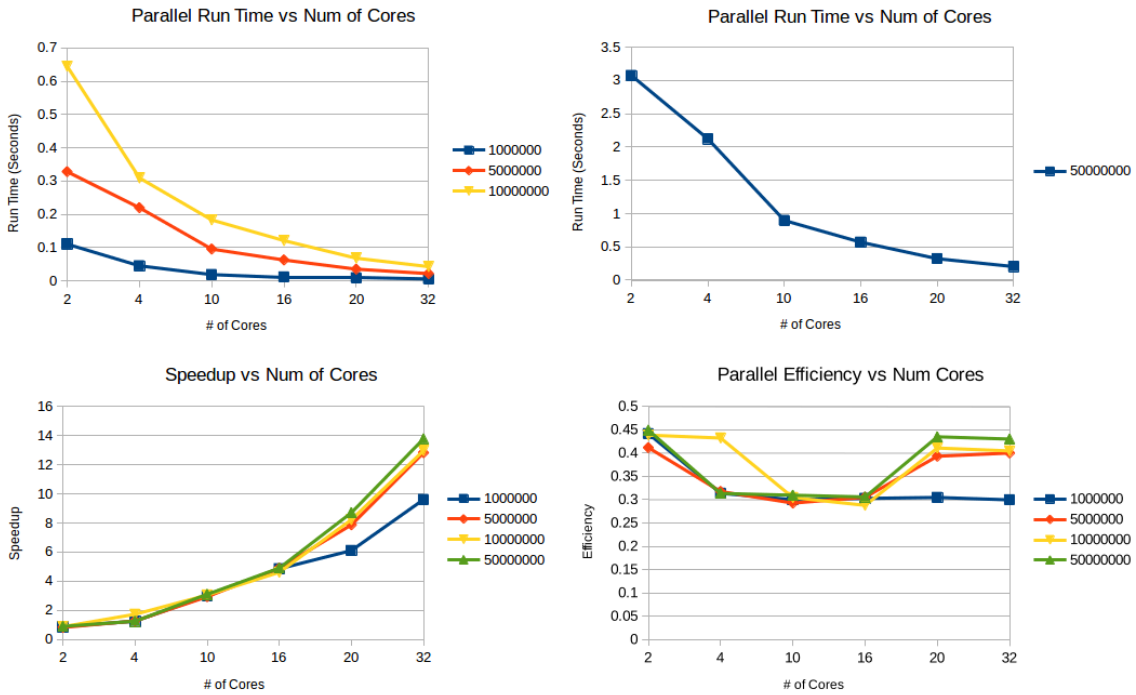
and the Parallel Efficiency was calculated as:

$$ParallelEfficiency = \frac{Speedup}{Num\ of\ Cores} = \frac{Thomas\ Algorithm\ Run\ Time}{Num\ of\ cores * Parallel\ Run\ Time}$$

the results of each trial are below in Table 1, and a graphical summary is presented in the figures following the table:

Table 1: *Performance Results*

Trials				
Problem Size (N)	Number of Processors	Run Time	Speedup	Efficiency
1,000,000	Serial time: 0.057035			
	2	0.110394	0.88302806	0.44151403
	4	0.0454381	1.25522414	0.31380604
	10	0.018939	3.00200644	0.30020064
	16	0.0120239	4.85033974	0.30314623
	20	0.00927114	6.10399584	0.30519979
	32	0.00590301	9.60506589	0.30015831
5,000,000	Serial time: 0.28042			
	2	0.327978	0.82309179	0.4115459
	4	0.219792	1.27037836	0.31759459
	10	0.095686	2.92946722	0.29294672
	16	0.0629101	4.88983486	0.30561468
	20	0.0353861	7.87165017	0.39358251
	32	0.021872	12.8209583	0.40065495
10,000,000	Serial time: 0.561426			
	2	0.644677	0.8768747761	0.438437388
	4	0.309206	1.7297141711	0.4324285428
	10	0.183201	3.0481493005	0.30481493
	16	0.121082	4.61117259	0.28819828
	20	0.0682559	8.2253109255	0.4112655463
	32	0.042963	12.9618043433	0.4050563857
50,000,000	Serial time: 2.79557			
	2	3.07504	0.8981053905	0.4490526952
	4	2.12244	1.2541226136	0.3135306534
	10	0.895866	3.0985660802	0.309856608
	16	0.569611	4.8993962546	0.3062122659
	20	0.321817	8.7063144582	0.4353157229
	32	0.202909	13.7774568895	0.4305455278



V. Discussion

I. Memory Management

Initially the program encountered segmentation faults while using arrays for both vectors and matrices when the problem size became too large (in this case for N > 2000). This was due to the large array sizes quickly filling up the available memory in the stack. Dynamically allocating memory to the heap using malloc instead of statically declaring the arrays solved this. Ultimately vectors, which automatically allocate contiguous dynamic memory on the heap (although header info is still on the stack) proved to be the best implementation strategy. Alternatively, during development, even the heap became overloaded while using a 2-dimensional vector implementation for the matrix for N = 50000. For this reason it proved favourable to restructure the program to handle a set of 3 vectors as a representation of the 2-dimensional matrix; saving both memory and time as many functions were reduced to having only one loop instead of two.

II. Communication Overhead

During preliminary trials it was apparent that a significant portion of the total runtime was due to communication overhead, especially as the number of processors increased. There were two major overhauls of the program, each significantly improving communication costs. The first, throwing out 2D arrays for a 3-vector representation, has already been discussed. This ultimately reduced the communicated data from size $(N^2 + N)$ to $4N$. However four individual broadcasts of length N can still be incredibly taxing as N increases, so eventually the program was further improved to 1) package all data to be sent into one "package" vector so that only one communication was necessary 2) Partition the problem as much as possible before sending so that instead of dealing

with data of size N, data of size $k = N/p$ or even less could be sent. Even with the packing and unpacking times included this approach proved to take only $5 - 8\%$ as long as the original data broadcasting.

III. Cluster Architecture

The DRP cluster of AMOS that the scaling study was performed on is comprised of 64 nodes connected via 56Gb FDR Infiniband. Each node has two eight-core 2.6 GHz Intel Xeon E5-2650 processors and 128GB of system memory. For the purpose of the study hyper-threading was disabled, allowing only 16 processes to run on each node at a time.

IV. Performance and Scaling

The limiting factor in a parallel program is almost always communication time. As shown in the results section the program exhibits strong scaling and this can largely be attributed to the nature of the improved communication in the program. The data sent in both the gather and broadcast in the program are constant in size and therefore only depend on the amount of cores 'p'; the gather is of size $8 * p$ and the broadcast is of size $2 * p - 2$. With most parallel programs there will eventually be a point when the communication cost of adding more cores begins to outweigh the additional speedup gained and the program will begin to become less efficient, however we did not observe this turning point in our trials due to limited resources on the DRP cluster.

V. How To Run

First ssh into a landing pad. To do this you will either need to be on campus or vpn onto the campus network and have login credentials. Then ssh into the cluster where you will run the program.

Modules

In order to successfully compile your code you must ensure that the necessary modules are loaded into your environment. For our purposes we will be needing gcc and OpenMPI. The commands to load these are:

module load gcc/4.9.1_1
module load openmpi/1.8.1_1

Keep in mind that the version is subject to change, to check the most up to date version use:

module spider gcc

SLURM

Submitting jobs requires communication with SLURM (Simple Linux Utility for Resource Management). SLURM is a free and open-source job scheduler that is commonly used on supercomputers and large clusters. The main components of a job allocation through SLURM are the number of nodes being requested, the number of processes being run, and the desired time on the nodes requested. Salloc and an SBATCH script are the main methods of doing this. An Salloc example requesting 1 node for 2 processes and 3 seconds is as follows:

$mpic++program\_name.cpp$
$salloc\ -N\ 1\ -n\ 2\ -t\ 3\ \ mpirun\ -np\ \ 2\ \ ./a.out\ \ problemsize$

Alternatively you can create an SBATCH file such as the following:

Listing 1: *SBATCH Example*

```
#!/bin/bash -x
#SBATCH -J scalingstudy              // Title
#SBATCH --output=scalingstudy.%j.out // Output file
#SBATCH --error=scaling-err.%j.err   // Error output file
#SBATCH --time=00:00:30              // Time requested on node(s)
#SBATCH --partition=drp
#SBATCH --mail-type=begin            // Mail user when program begins
#SBATCH --mail-type=end              // Mail user when program ends
#SBATCH --mail-user=landhb@rpi.edu   // User email address

#SBATCH -N 1                         // Number of Nodes requested
#SBATCH --ntasks-per-node=16         // Set max MPI tasks per node, 16 based on DRP architecture

module load gcc/4.9.1_1              // Load gcc module
module load openmpi/1.8.1_1          // Load openmpi module

mpic++ enhanced_version.cpp          // Compile Program

mpirun -np 2 ./a.out 1000            // Run Program with 2 mpi processes
```

To run the SBATCH file use the following command:

$sbatch\ \ file\_name.sh$

VI. References

[Stefan Bondeli, 1990] Bondeli, Stefan, Zurich Departement Informatik (1990). Divide and Conquer: A new parallel algorithm for the solution of a tridiagonal linear system of equations.

VII. Psuedocode

Algorithm 1 Cyclic Reduction Method

```
1:
2: Begin the Reduction phase of the algorithm
3:
4: for (i = 1; i <= log2(n+1); i++) do                    ▷ Loop through number of stages to solve
5:     for j = 0 to numactivep do                          ▷ Loop through active processors
6:         if this processor is active then
7:             Reduce system of 3 equations to 1 equation
8:             if not reduced then
9:                 send/receive information to/from neighbours
10:            end if
11:        end if
12:    end for
13:    Reduce active processor list
14: end for
15:
16: Begin the Back substitution phase of the algorithm
17:
18: for (i = log2(n+1)-1; i >= 1; i−−) do          ▷ Loop back through number of stages to solve
19:    MPI::Allgather(result)                                ▷ send result to all processors
20:    refill active processors
21:    for j = 0 to numactivep do                  ▷ Loop through processors without solutions
22:        if this processor is active then
23:            Solve for local solution on this processor
24:        end if
25:    end for
26: end for
27: MPI::Allgather(local result)                             ▷ send local result to all processors
28:
29: if processor 0 then          ▷ Processor 0 loops through even rows and computes final values
30:    for all even rows do
31:        find solution
32:    end for
33:    Print solutions
34: end if
```

VIII. Code Attachments

Listing 2: *Divide and Conquer Implementation*

```
// Divide and Conquer Algorithm Implementation :
//
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <sstream>
#include <cmath>
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stddef.h>



using namespace std;

//const int N = 500;
double start_time;
double finish_time;
double gather_start;
double gather_finish;
double bcast_start;
double bcast_finish;


//Generate random numbers between fMin and fMax
double fRand(double fMin, double fMax)
{
   double f = (double)rand() / RAND_MAX;
   return fMin + f * (fMax - fMin);
}

//Multiply a matrix represented by 3 vectors by a vector
void multiply_vec(vector<double> a, vector<double> b, vector <double> c, vector <double> d,vector <
     double> &sol, int N){
  for (int i = 0; i < N; i++){
    if (i ==0){
      sol[i] = b[i]*d[i] + c[i]*d[i+1];
    }
    else if (i == N-1){
      sol[i] = a[i-1]*d[i-1] + b[i]*d[i];
    }
    else{
      sol[i] = a[i-1]*d[i-1] + b[i]*d[i] + c[i]*d[i+1];
    }
  }
}

//Generate the tridiagonal matrix represented by 3 vectors
void vec_generator(vector<double> &a, vector <double> &b, vector <double> &c, vector <double> &sol,
     int N){
  srand(time(NULL));

  //Create diagnoally dominate set of 3 vectors representing the matrix
```

```
55    for (int i = 0; i < N; i++){
56      if(i ==0){
57        b[i] = fRand(0,100);
58        c[i] = fRand(0,b[i]);
59      }
60      else if (i == N-1){
61        b[i] = fRand(0,100);
62        a[i-1] = fRand(0,b[i]);
63      }
64      else{
65        a[i-1] = fRand(0,100);
66        c[i] = fRand(0,100);
67        b[i] = fRand(a[i-1]+c[i],200);
68      }
69    }
70
71    //find solution vector
72    vector <double> d;
73
74    for (int i = 0; i <N; i ++){
75      d.push_back(i+1.0);
76    }
77
78    multiply_vec(a,b,c,d,sol,N);
79
80  }
81
82
83  //Function to print a single vector
84  void print_a_vector(vector <double> a, int N){
85    for (int i = 0; i < N; i++){
86      cout << a[i] << ",␣";
87    }
88    cout << endl;
89  }
90
91
92  //Print 3 vectors as a matrix
93  void print_vec(vector <double> a,vector <double> b, vector<double> c, int N){
94    int index = -1;
95    for (int i =0; i < N; i++){
96      for (int j =0; j<N;j++){
97
98        if (j == 0 && index == -1){
99          cout << b[i] << ",␣␣" << c[i];
100         j += 1;
101       }
102       else if (i == N-1 && index == j) {
103         cout << a[i-1] << ",␣␣" << b[i];
104
105       }
106       else if(index == j){
107         cout << a[i-1] << ",␣␣" << b[i] << ",␣␣" << c[i];
108         j += 2;
109       }
110       else {
111         cout << "0,␣␣";
112       }
113
114     }
115     cout << endl;
```

```
    index++;
  }
}

//Serial Thomas Algorithm implementation
vector <double> thomas_algorithm(vector<double> a,
                                 vector<double> b,
                                 vector<double> c,
                                 vector <double> d,
                                 int size) {

  double m;
  //forward elimination phase
  for (int i = 1; i < size; i++){
    m = a[i-1]/b[i-1];
    b[i] = b[i] - m*c[i-1];
    d[i] = d[i] - m*d[i-1];
  }

  //backward substitution phase
  d[size-1] = d[size-1]/b[size-1];
  for (int i = size-2; i >= 0; i--){
    d[i] = (d[i]-c[i]*d[i+1])/b[i];
  }
  return d;

}



int main (int argc, char *argv[])
{

        MPI::Init(argc,argv);                   //  Initialize MPI
        MPI::Comm & comm = MPI::COMM_WORLD;     //

        int comm_sz = comm.Get_size();          //  Total number of processors
        int my_rank = comm.Get_rank();          //  Rank of current processor


        //Get the desired problem size
        int N = atoi(argv[1]);                  //  Problem Size N
        const int k = N/comm_sz;                //  k-value









        vector<double> y(k);            //
        vector<double> ek(k);           //
        vector<double> e1(k);           // All used to calulculate local variables y and zm/zm1 on
             each processor
        vector<double> zm(k);           //
        vector<double> zm1(k);          //




        vector <double> a(N);           //
        vector <double> b(N);           // Matrix element vectors
        vector <double> c(N);           //
```

```
176         vector <double> sol(N);         //
177
178         vector <double> a1(k);          //
179         vector <double> b1(k);          // Local matrix element vectors
180         vector <double> c1(k);          //
181         vector <double> d1(k);          //
182
183         vector <double> package;        // Used to scatter a,b,c, and sol as a single message to
                reduce overhead
184         package.reserve(4*N);           // Allocate memory to the package
185
186         vector<double> recieve1(4*k);   // Used to recieve initial scaller
187
188         vector <double> package2;         //
189         package.reserve(8);               // Used to gather local reduced matrix solutions
190         vector<double> recieve(8*comm_sz); //
191
192
193         vector<double> s(2*comm_sz-2);  //
194         vector<double> r(2*comm_sz-2);  // Vectors of the reconstructed Matrix of
195         vector<double> t(2*comm_sz-2);  // size 2*(# of processors)-2
196         vector<double> u(2*comm_sz-2);  //
197
198
199
200
201         //initialize E1 and Ek
202         e1[0] = 1;
203         ek[k-1] = 1;
204
205
206         int f = 0;
207         //Processor 0 creates tridiagonal matrix
208         if (my_rank == 0){
209
210           vec_generator(a,b,c,sol,N);
211
212
213           //Partition the problem for each processor
214
215           for(int j = 0; j < comm_sz; j++){
216             f = 0;
217             for (int i = j*k; i < (j*k)+k; i++){
218               d1[f] = sol[i];
219               b1[f] = b[i];
220               if (i != (j*k)+k-1){
221                 a1[f] = a[i];
222                 c1[f] = c[i];
223               }
224               else{
225                 a1[f] = a[i];
226                 if (i < k){
227                   c1[f] = 0;
228                 }
229                 else{
230                   c1[f] = c[i-k];
231                 }
232
233               }
234               f++;
235             }
```

```
236           //Add the processor partition just created to the scatter package
237           package.insert(package.end(),a1.begin(),a1.end());            //
238           package.insert(package.end(),b1.begin(),b1.end());            //  Combine vectors into one
239           package.insert(package.end(),c1.begin(),c1.end());            //  vector before the
                  scatter
240           package.insert(package.end(),d1.begin(),d1.end());            //
241          }
242
243         }
244
245
246        //Scatter information amongst Processors
247        comm.Scatter(&package.front(),4*k,MPI_DOUBLE,&recieve1.front(),4*k,MPI_DOUBLE,0);
248
249
250
251        //Unpack the vectors from the container after scatter
252        double a_const = 0;
253        double c_const = 0;
254        f = 0;
255        for (int i =0; i < 4*k; i++){
256
257
258            if (i < k){
259              if (f == k-1) {
260                a_const = recieve1[i];
261                a1[f] = 0;
262              }
263              else{
264                a1[f] = recieve1[i];
265              }
266            }
267            else if (i < 2*k){
268              b1[f] = recieve1[i];
269
270            }
271            else if (i < 3*k){
272              if (f == k-1) {
273                c_const = recieve1[i];
274                c1[f] = 0;
275              }
276              else{
277                c1[f] = recieve1[i];
278              }
279            }
280            else {
281              d1[f] = recieve1[i];
282
283            }
284            f++;
285            if (f == k){ f = 0;}
286
287        }
288
289        /*Code to check Scatter
290        if(my_rank ==0){
291          cout << "after scatter: " << endl;
292          print_a_vector(recieve1,4*k);
293          cout << endl << endl;
294          print_a_vector(a1,k);
295          print_a_vector(b1,k);
```

```
296         print_a_vector(c1,k);
297         cout << "a_const: " << a_const << " c_const: " << c_const << endl;
298         }*/
299
300       //Start timing soluton computation once problem is on each processor
301       start_time = MPI::Wtime();
302
303
304       //Solve local systems on each processor
305
306       y = thomas_algorithm(a1,b1,c1,d1,k);
307       if (my_rank == 0){
308         zm = thomas_algorithm(a1,b1,c1,ek,k);
309       }
310       else if (my_rank == comm_sz-1){
311         zm = thomas_algorithm(a1,b1,c1,e1,k);
312       }
313       else{
314         zm = thomas_algorithm(a1,b1,c1,e1,k);
315         zm1 = thomas_algorithm(a1,b1,c1,ek,k);
316       }
317
318
319
320       /*Code to check solution on individual processor
321       if(my_rank != 0){
322         cout << "solution on p " << my_rank << "is: " << endl;
323         print_a_vector(y,k);
324
325         cout << "Zm is equal to: " << endl;
326         print_a_vector(zm,k);
327
328         cout << "zm1 is equal to: " << endl;
329         print_a_vector(zm1,k);
330         }*/
331
332
333       //Create local reduced matrix elements on every processor
334       vector<double> s_local(2);
335       vector<double> r_local(2);
336       vector<double> t_local(2);
337       vector<double> u_local(2);
338
339       int count =0;
340
341       if(my_rank ==0){
342         s_local[0] = (zm[k-1]);
343         t_local[0] = (1/a_const);
344         u_local[0] = (-y[k-1]);
345       }
346       else if(my_rank == comm_sz-1){
347         if ((2*comm_sz-2)%2 == 0){
348           s_local[0] = (zm[0]);
349           r_local[0] = (1/c_const);
350           u_local[0] = (-y[0]);
351         }
352         else {
353           s_local[0] = (zm[k-1]);
354           u_local[0] = (-y[k-1]);
355         }
356       }
```

```
357         else {
358 
359             s_local[0] = zm[0];
360             s_local[1] = zm1[k-1];
361             r_local[0] = (1/c_const);
362             r_local[1] = (zm[k-1]);
363             t_local[0] = (zm1[0]);
364             t_local[1] = (1/a_const);
365             u_local[0] = (-y[0]);
366             u_local[1] = (-y[k-1]);
367 
368         }
369 
370         gather_start = MPI::Wtime();
371 
372         //Pack into communication packet
373         package2.insert(package2.end(),s_local.begin(),s_local.end());
374         package2.insert(package2.end(),r_local.begin(),r_local.end());
375         package2.insert(package2.end(),t_local.begin(),t_local.end());
376         package2.insert(package2.end(),u_local.begin(),u_local.end());
377 
378         //Gather results back to rank 0
379         comm.Gather(&package2.front(),8,MPI_DOUBLE,&recieve.front(),8,MPI_DOUBLE,0);
380 
381 
382         //Unpack on rank 0
383         if (my_rank ==0){
384           int count = 0;
385           for (int i =0; i < 8*comm_sz; i+=8){
386             if (i == 0){
387               s[0] = recieve[0];
388               t[0] = recieve[4];
389               u[0] = recieve[6];
390               count ++;
391             }
392             else if (i == 8*(comm_sz-1)){
393                 s[count] = recieve[i];
394                 r[count-1] = recieve[i+2];
395                 u[count] = recieve[i+6];
396                 count++;
397             }
398             else{
399               s[count] = recieve[i];
400               s[count+1] = recieve[i+1];
401               r[count-1] = recieve[i+2];
402               r[count] = recieve[i+3];
403               t[count] = recieve[i+4];
404               t[count+1] = recieve[i+5];
405               u[count] = recieve[i+6];
406               u[count+1] = recieve[i+7];
407               count+=2;
408             }
409           }
410         }
411 
412         gather_finish = MPI::Wtime();
413 
414         //Solve new matrix on rank 0
415         if(my_rank == 0){
416 
417           u = thomas_algorithm(r,s,t,u,2*comm_sz-2);
```

```
418
419         }
420
421         //Bcast new matrix solutions
422         bcast_start = MPI::Wtime();
423         comm.Bcast(&u.front(),2*comm_sz-2,MPI_DOUBLE,0);
424         bcast_finish = MPI::Wtime();
425
426         //Solve for local solution "x" using solution of the reduced matrix
427         vector<double> x(k);
428         int index = 2*(my_rank+1)-3;
429         if (my_rank ==0){
430           index =0;
431           for (int i = 0; i < k; i++){
432                x[i] = y[i] + u[index]*zm[i];
433           }
434         }
435         else if (my_rank == comm_sz-1){
436           for (int i = 0; i < k; i++){
437                x[i] = y[i] + u[index]*zm[i];
438           }
439         }
440         else {
441           for (int i =0; i < k; i++){
442             x[i] = y[i] + u[index]*zm[i] + u[index+1]*zm1[i];
443           }
444         }
445
446
447         //Time finish
448         finish_time = MPI::Wtime();
449
450         vector<double> x_all(N);
451
452         //Gather all solutions back to rank 0 to print and check values
453         comm.Gather(&x.front(), k, MPI_DOUBLE, &x_all.front(),k,MPI_DOUBLE,0);
454
455         //Code to print and check solution
456         if(my_rank ==0){
457
458           double err_sum = 0.0;
459           double value = 0.0;
460           for (int i= 0; i < N; i++){
461             value = x_all[i] - double(i+1);
462             //cout << "x: " << x_all[i] << " i " << double(i+1) << " " << value << " ";
463             value = abs(value);
464             //cout << value << " ";
465             err_sum += value;
466             //cout << "err_sum " << err_sum << endl;
467           }
468           err_sum /= N;
469
470
471           //Print Results
472           cout << endl << "For␣problem␣size␣N␣=␣" << N << "␣and␣" << comm_sz << "␣processors:␣" <<
                   endl;
473           cout << "Average␣error␣is:␣";
474           cout << err_sum << endl;
475           cout << "Run␣time␣is␣" << finish_time-start_time << endl;
476           cout << "Communication␣time␣is:␣" << (bcast_finish - bcast_start) + (gather_finish-
                   gather_start) << endl;
```

```
477           cout << "Bcast time is: " << bcast_finish - bcast_start << endl;
478
479           vector<double> solutions(N);
480
481           //Record serial time
482           start_time = MPI::Wtime();
483           solutions = thomas_algorithm(a,b,c,sol,N);
484           finish_time = MPI::Wtime();
485
486           cout << "Serial time is " << finish_time-start_time << endl;
487         }
488
489
490         MPI::Finalize();
491
492 }
```