# Assignment 2 - Scientific Computing SC9505

Jared Wogan

2022-11-21

All code was compiled inside an Ubunutu WSL environment on an AMD 3900X with 32GB of RAM. All timing are performance data was run with dense, randomly generated arrays and vectors.

## 1 Matrix Vector Multiplication Using MPI

Our first exercise is to determine the benefits of parallelization on a code that computes a matrix vector product. We first test the code ensuring that it is giving correct results, as can be seen in Figure 1.

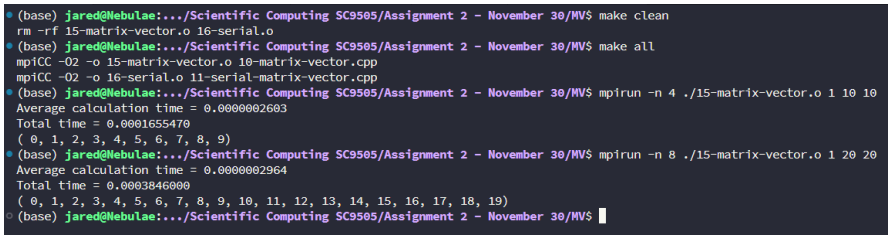



Figure 1: Demonstration of the code working. Here the matrix is taken to be the identity, and the vector is filled with consecutive integers.

The scaling when the program is run on 4 threads as a function of the number of rows/columns was determined to be $\mathcal{O}(N^{1.5})$, see Figure 2. This is less than is expected for matrix vector multiplication, which is typically an $\mathcal{O}(N^2)$ operation. The serial code I used to compare the parallel code with scaled roughly as $\mathcal{O}(N^{2.2})$.

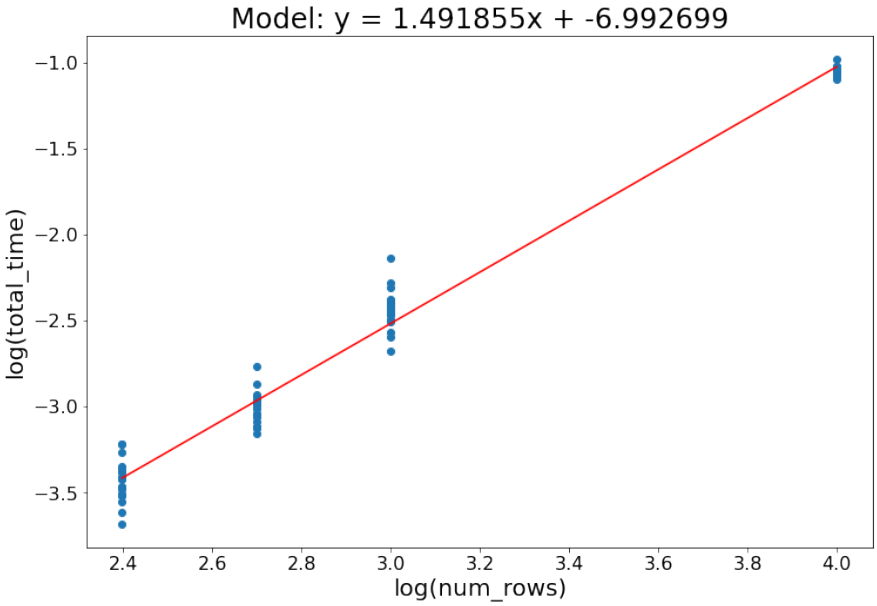



Figure 2: Plotting the runtime of the program as a function of the number of rows $N$ (on a log-log scale), we find that the total runtime scales as $\mathcal{O}(N^{1.5})$, which is less than the expected $\mathcal{O}(N^2)$. Note the plot is generated with data for a constant amount of threads, which in this case is 4.

Looking at the scaling instead for a constant number of rows (here we are using 10000), we find there is no real scaling. Ideally we would like to see the total runtime decrease as we increase the number of threads, but this is not the case for the algorithm being used here.

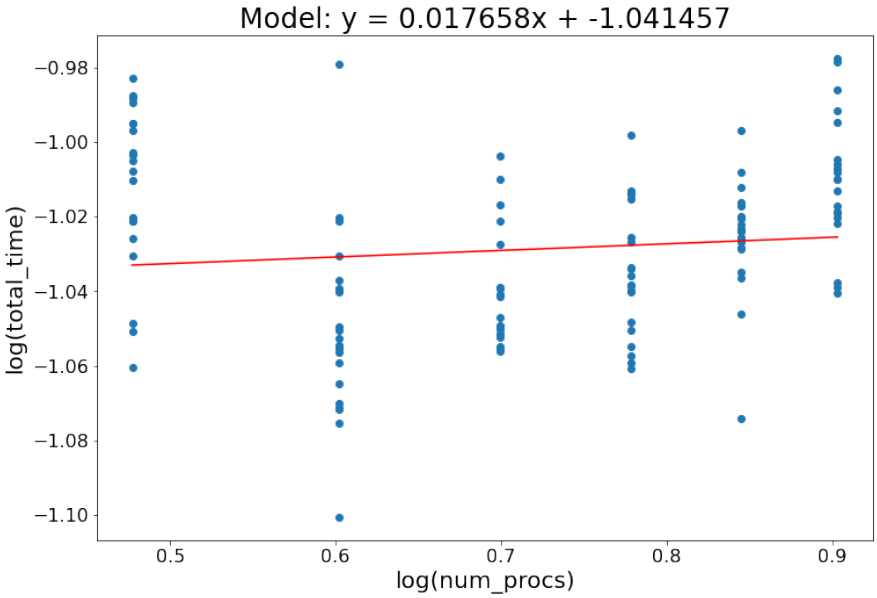



Figure 3: Plotting the runtime of the program as a function of the number of threads $P$ (on a log-log scale). The linear fit is not ideal, such that it is fair to conclude that there is no note worthy scaling with the number of processors. Again it is worth noting that the plot has been generated for a constant number of rows which was taken to be 10000.

As discussed in lecture, the efficiency of our algorithm is less than ideal, which is supported by the calculations in Table 1 (see Appendix A for details on how these values were calculated). We see that running the code across 3 threads yields superlinear scaling, and our program is more efficient than the serial code. However, as the number of threads increase, we start seeing diminishing returns in terms of the speedup, and so our efficiency starts dropping dramatically. At 8 concurrent threads, our code is less than half as efficient as the serial code, and it is actually slower than the case with only 3 threads (in reality they are approximataly equal in runtime, but the linear model predicted it would be slower).

If we wanted to write a program to do matrix vector multiplication in parallel more efficiently, it would be best to have a shared pool of memory that each process can read and write to at the same time. This could be done by adding a single line to the serial code and using OpenMP.

| Number of Threads | Speedup | Scaling Factor $a$ | Efficiency |
|---|---|---|---|
| 3 | 3.92 | 1.31 | 1.31 |
| 4 | 3.76 | 0.94 | 0.94 |
| 5 | 3.61 | 0.72 | 0.72 |
| 6 | 3.47 | 0.58 | 0.58 |
| 7 | 3.33 | 0.48 | 0.48 |
| 8 | 3.19 | 0.40 | 0.40 |

Table 1: Summary of the speedup and efficiency of our matrix vector multiplication code. These values are taken for a constant number of rows $N = 10000$.

```
////////////////////////////////////////////
// Matrix-vector multiplication code Ab=c //
////////////////////////////////////////////

// Note that I will index arrays from 0 to n-1.
// Here workers do all the work and boss just handles collating results
// and sending info about A.

// include, definitions, globals etc here
#include <iostream>
#include <fstream>
#include <iomanip>
#include <random>
#include "boost/multi_array.hpp"
#include "mpi.h"

using namespace std;

class MPI_Obj
{
public:
    int size;
    int rank;

    MPI_Obj(int &argc, char **&argv)
    {
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    }

    ~MPI_Obj()
    {
        MPI_Finalize();
    }
};

void GetArraySize(int &output, int &nrows, int &ncols, MPI_Obj &the_mpi, int argc, char **argv)
{
    if (the_mpi.rank == 0)
    {
        if (argc == 4)
        {
            output = atoi(argv[1]);
            nrows = atoi(argv[2]);
            ncols = atoi(argv[3]);
        }
        else
        {
            cout << "Please enter the number of rows -> ";
            cin >> nrows;
            cout << "Please enter the number of columns -> ";
            cin >> ncols;
        }
    }

    // send everyone nrows, ncols
    int buf[2] = {nrows, ncols};
    MPI_Bcast(buf, 2, MPI_INT, 0, MPI_COMM_WORLD);
    if (the_mpi.rank != 0)
    {
```

```
62          nrows = buf[0];
63          ncols = buf[1];
64      }
65  }
66
67  void SetupArrays(int nrows, int ncols, boost::multi_array<double, 2> &A, vector<double> &b,
    ↪  vector<double> &c, vector<double> &Arow, MPI_Obj &the_mpi)
68  {
69      uniform_real_distribution<double> unif(-1.0, 10);
70      default_random_engine re;
71      // Boss part
72      if (the_mpi.rank == 0)
73      {
74          // Set size of A
75          A.resize(boost::extents[nrows][ncols]);
76
77          // Initialize A
78          for (int i = 0; i < nrows; ++i)
79              for (int j = 0; j < ncols; ++j)
80              {
81                  // Identity
82                  // if (i == j)
83                  //     A[i][j] = 1.0;
84                  // else
85                  //     A[i][j] = 0.0;
86                  // Reverse Identity
87                  // if (i == (ncols - j - 1))
88                  //     A[i][j] = 1.0;
89                  // else
90                  //     A[i][j] = 0.0;
91                  // Random
92                  A[i][j] = unif(re);
93              }
94
95          // Initialize b
96          for (int i = 0; i < ncols; ++i)
97          {
98              // b[i] = 1.0;
99              // b[i] = (double) i;
100             b[i] = unif(re);
101         }
102
103         // Allocate space for c, the answer
104         c.reserve(nrows);
105         c.resize(nrows);
106     }
107     // Worker part
108     else
109     {
110         // Allocate space for 1 row of A
111         Arow.reserve(ncols);
112         Arow.resize(ncols);
113     }
114
115     // send b to every worker process, note b is a vector so b and &b[0] not same
116     MPI_Bcast(&b[0], ncols, MPI_DOUBLE, 0, MPI_COMM_WORLD);
117 }
118
119 void Output(int output, vector<double> &c, MPI_Obj &the_mpi)
120 {
121     if (the_mpi.rank == 0 && output == 1)
122     {
123         cout << "( " << c[0];
124         for (int i = 1; i < c.size(); ++i)
125             cout << ", " << c[i];
126         cout << ")\n";
127     }
128 }
129
130 int main(int argc, char **argv)
131 {
132     // Data File
133     // ofstream datafile("/home/jared/Desktop/mv-timings.txt", ios_base::app);
134     ofstream datafile("mv-timings.txt", ios_base::app);
135
136     // initialize MPI
137     MPI_Obj the_mpi(argc, argv);
```

```
138     if (the_mpi.size < 2)
139         MPI_Abort(MPI_COMM_WORLD, 1);
140 
141     // determine/distribute size of arrays here
142     int output = 1, nrows = 0, ncols = 0;
143     GetArraySize(output, nrows, ncols, the_mpi, argc, argv);
144     if (the_mpi.size - 1 > nrows)
145     {
146         MPI_Abort(MPI_COMM_WORLD, -1);
147     }
148 
149     boost::multi_array<double, 2> A;
150     vector<double> b(ncols);
151     vector<double> c;
152     vector<double> Arow;
153     SetupArrays(nrows, ncols, A, b, c, Arow, the_mpi);
154 
155     MPI_Status status;
156 
157     // Timing variables
158     double calc_time = 0, avg_time, total_time;
159 
160     // Boss part
161     if (the_mpi.rank == 0)
162     {
163         total_time = MPI_Wtime();
164         // send one row to each worker tagged with row number, assume size<nrows
165         int rowsent = 0;
166         for (int i = 1; i < the_mpi.size; i++)
167         {
168             MPI_Send(&A[rowsent][0], ncols, MPI_DOUBLE, i, rowsent + 1, MPI_COMM_WORLD);
169             rowsent++;
170         }
171 
172         for (int i = 0; i < nrows; i++)
173         {
174             double ans;
175             MPI_Recv(&ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
176             int sender = status.MPI_SOURCE;
177             int anstype = status.MPI_TAG; // row number+1
178             c[anstype - 1] = ans;
179             if (rowsent < nrows)
180             { // send new row
181                 MPI_Send(&A[rowsent][0], ncols, MPI_DOUBLE, sender, rowsent + 1, MPI_COMM_WORLD);
182                 rowsent++;
183             }
184             else
185             { // tell sender no more work to do via a 0 TAG
186                 MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, sender, 0, MPI_COMM_WORLD);
187             }
188         }
189     }
190     else
191     {
192         // Get a row of A
193         MPI_Recv(&Arow[0], ncols, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
194         while (status.MPI_TAG != 0)
195         {
196             // work out Arow.b
197             double ans = 0.0;
198 
199             calc_time -= MPI_Wtime();
200             for (int i = 0; i < ncols; i++)
201                 ans += Arow[i] * b[i];
202             calc_time += MPI_Wtime();
203 
204             // Send answer of Arow.b back to boss and get another row to work on
205             MPI_Send(&ans, 1, MPI_DOUBLE, 0, status.MPI_TAG, MPI_COMM_WORLD);
206             MPI_Recv(&Arow[0], ncols, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
207         }
208     }
209 
210     if (the_mpi.rank == 0)
211         total_time = MPI_Wtime() - total_time;
212 
213     MPI_Reduce(&calc_time, &avg_time, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
214 
```

```
    if (the_mpi.rank == 0)
    {
        avg_time /= (the_mpi.size - 1); // Boss node doesn't do any of the calculations
        printf("Average calculation time = %.10f\n", avg_time);
        printf("Total time = %.10f\n", total_time);

        datafile << fixed << setprecision(10);
        datafile << nrows << " " << ncols << " " << the_mpi.size << " " << avg_time << " " << total_time
    << endl;
    }

    // output c here on Boss node
    Output(output, c, the_mpi);
}
```

## 2 Matrix Matrix Multiplication Using MPI

Our second task is to generalize the matrix vector product code to perform matrix matrix products. The modified code can be seen below, while a demonstration that the code works correctly can be seen in Figure 4. The serial code that will be used for comparison was written using the DGEMM_ function from the BLAS library.

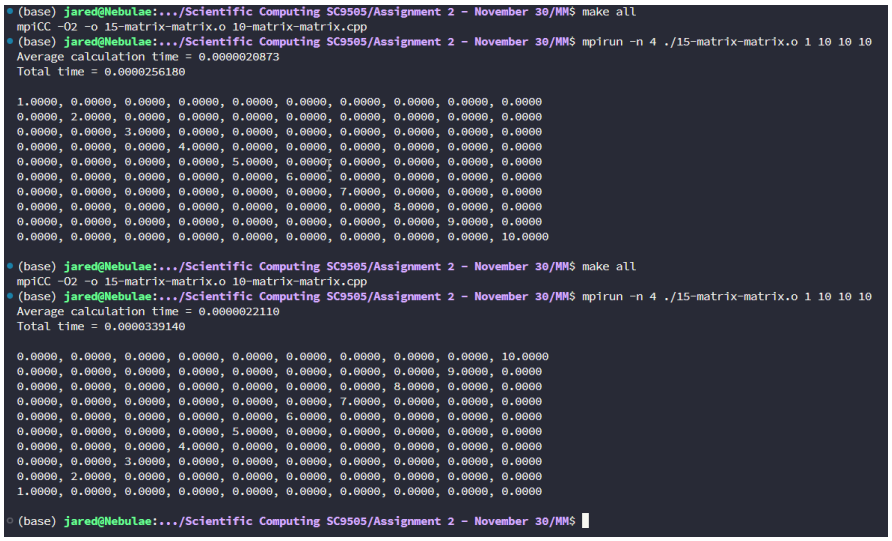



Figure 4: Demonstration of the code working. In the first example, the first matrix is taken to be the identity and the second has consecutive integers along the main diagonal. In the second example, the first matrix is changed to the reverse identity (1s along the antidiagonal). In both cases, we get the expected result.

The modified code for matrix matrix multiplication scales as $\mathcal{O}(N^{3.3})$,where $N$ is the number of rows as can be seen in Figure 5. This is slightly more than the expected$\mathcal{O}(N^3)$.

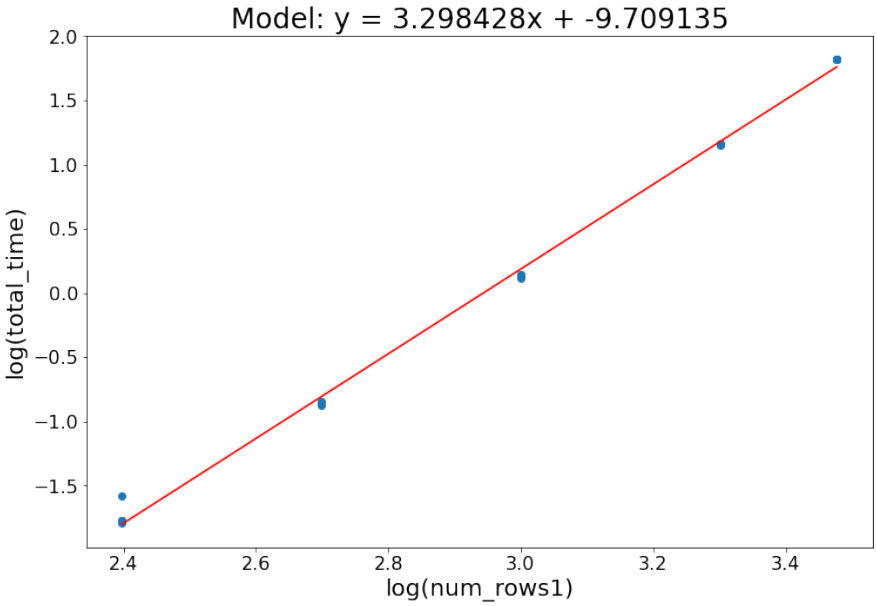



Figure 5: Plotting the runtime of the program as a function of the number of rows $N$ (on a log-log scale), we find that the total runtime scales as $\mathcal{O}(N^{3.3})$, which is slightly more than the expected$\mathcal{O}(N^3)$. It should be noted that the serial code scaled as $\mathcal{O}(N^{2.9})$, which is slightly less than expected. It is also interesting that the current world record (as of 2020) for the asympotitc scaling of matrix matrix multiplication has been shown to be of the order$\mathcal{O}(N^{2.3728596})$ by Josh Alman and Virginia Vassilevska Williams (arXiv:2010.05846).

More importantly for our purposes, we find that the total runtime scales as $\mathcal{O}(P^{-1})$,where $P$ is the total number of threads (for a fixed number of rows $N$). This is the expected result as we discussed in lecture: for large $N$,we expect the speedup to approach $P$, which is supported by Figure 6 and Table 2.

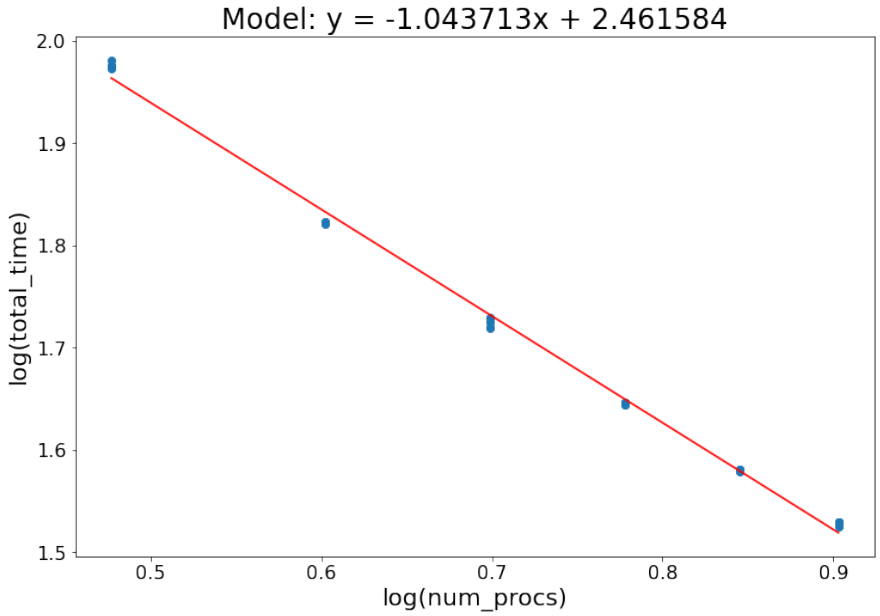



Figure 6: Plotting the runtime of the program as a function of the number of threads $P$, we find the scaling for a fixed number of rows $N$ (here $N = 3000$) to be of the order $\mathcal{O}(P^{-1})$

| Number of Threads | Speedup | Scaling Factor $a$ | Efficiency |
|---|---|---|---|
| 3 | 1.66 | 0.55 | 0.55 |
| 4 | 2.24 | 0.56 | 0.56 |
| 5 | 2.83 | 0.57 | 0.57 |
| 6 | 3.42 | 0.57 | 0.57 |
| 7 | 4.02 | 0.57 | 0.57 |
| 8 | 4.62 | 0.58 | 0.58 |

Table 2: Summary of the speedup and efficiency of our matrix matrix multiplication code. These values are taken for a constant number of rows $N = 3000$.

```
////////////////////////////////////////////
// Matrix-Matrix multiplication code AB=C //
////////////////////////////////////////////

// Note that I will index arrays from 0 to n-1.
// Here workers do all the work and boss just handles collating results
// and sending info about A.

// include, definitions, globals etc here
#include <iostream>
#include <fstream>
#include <iomanip>
#include <random>
#include "boost/multi_array.hpp"
#include "mpi.h"

using namespace std;

class MPI_Obj
{
public:
    int size;
    int rank;

```

```
25      MPI_Obj(int &argc, char **&argv)
26      {
27          MPI_Init(&argc, &argv);
28          MPI_Comm_size(MPI_COMM_WORLD, &size);
29          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
30      }
31
32      ~MPI_Obj()
33      {
34          MPI_Finalize();
35      }
36  };
37
38  void GetArraySize(
39      int &output,
40      int &nrows1,
41      int &nrowscols12,
42      int &ncols2,
43      MPI_Obj &the_mpi,
44      int argc,
45      char **argv
46  ) {
47      if (the_mpi.rank == 0)
48      {
49          if (argc  == 5) {
50              output = atoi(argv[1]);
51              nrows1 = atoi(argv[2]);
52              nrowscols12 = atoi(argv[3]);
53              ncols2 = atoi(argv[4]);
54          } else {
55              cout << "Please enter the number of rows for the first matrix -> ";
56              cin >> nrows1;
57              cout << "Please enter the number of columns/rows for the first/second matrix-> ";
58              cin >> nrowscols12;
59              cout << "Please enter the number of columns for the second matrix-> ";
60              cin >> ncols2;
61          }
62      }
63
64      // send everyone nrows, ncols
65      int buf[3] = {nrows1, nrowscols12, ncols2};
66      MPI_Bcast(buf, 3, MPI_INT, 0, MPI_COMM_WORLD);
67      if (the_mpi.rank != 0)
68      {
69          nrows1 = buf[0];
70          nrowscols12 = buf[1];
71          ncols2 = buf[2];
72      }
73  }
74
75  void SetupArrays(
76      int nrows1,
77      int nrowscols12,
78      int ncols2,
79      boost::multi_array<double, 2> &A,
80      boost::multi_array<double, 2> &B,
81      boost::multi_array<double, 2> &C,
82      vector<double> &Arow,
83      vector<double> &Crow,
84      MPI_Obj &the_mpi
85  ) {
86      uniform_real_distribution<double> unif(-1.0, 10);
87      default_random_engine re;
88
89      B.resize(boost::extents[nrowscols12][ncols2]);
90      Crow.reserve(ncols2); Crow.resize(ncols2); // Main process will need to store the values temporarily
91
92      // Boss part
93      if (the_mpi.rank == 0)
94      {
95          // Set size of A
96          A.resize(boost::extents[nrows1][nrowscols12]);
97          C.resize(boost::extents[nrows1][ncols2]);
98
99          // Initialize A
100         for (int i = 0; i < nrows1; ++i) {
101             for (int j = 0; j < nrowscols12; ++j) {
```

```
                    // Identity
                    // if (i == j)
                    //     A[i][j] = 1.0;
                    // else
                    //     A[i][j] = 0.0;
                    // Reverse Identity
                    // if (i == (nrowscols12 - j - 1))
                    //     A[i][j] = 1.0;
                    // else
                    //     A[i][j] = 0.0;
                    // Random
                    A[i][j] = unif(re);
                }
            }

            // Initialize B
            for (int i = 0; i < nrowscols12; ++i) {
                for (int j = 0; j < ncols2; ++j) {
                    // Identity
                    // if (i == j)
                    //     B[i][j] = 1.0;
                    // else
                    //     B[i][j] = 0.0;
                    // Reverse Identity
                    // if (i == (ncols2 - j - 1))
                    //     B[i][j] = 1.0;
                    // else
                    //     B[i][j] = 0.0;
                    // Random
                    B[i][j] = unif(re);
                    // Other
                    // if (i == j)
                    //     B[i][j] = (double) i + 1;
                    // else
                    //     B[i][j] = 0;
                }
            }
        } else {
            // Worker part
            // Allocate space for 1 row of A and 1 row of the answer C
            Arow.reserve(nrowscols12); Arow.resize(nrowscols12);
        }
        MPI_Bcast(&B[0][0], nrowscols12*ncols2, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }

    void Output(int output, boost::multi_array<double, 2> &array, MPI_Obj &the_mpi)
    {
        if (the_mpi.rank == 0 && output) {
            cout << endl << fixed << setprecision(4);

            for (int i = 0; i < array.shape()[0]; i++) {
                for (int j = 0; j < array.shape()[1]; j++) {
                    cout << array[i][j];
                    if (j < array.shape()[1] - 1) cout << ", ";
                }
                cout << endl;
            }
            cout << endl;

            cout << scientific;
        }
    }

    int main(int argc, char **argv)
    {
        // Data File
        // ofstream datafile("/home/jared/Desktop/mmblas-timings.txt", ios_base::app);
        ofstream datafile("mm-timings.txt", ios_base::app);

        // initialize MPI
        MPI_Obj the_mpi(argc, argv);
        if (the_mpi.size < 2) MPI_Abort(MPI_COMM_WORLD, 1);

        // determine/distribute size of arrays here
        int output = 1, nrows1 = 0, nrowscols12 = 0, ncols2 = 0;
        GetArraySize(output, nrows1, nrowscols12, ncols2, the_mpi, argc, argv);
        if (the_mpi.size - 1 > nrows1) {
```

```
179             MPI_Abort(MPI_COMM_WORLD, -1);
180         }
181
182         boost::multi_array<double, 2> A;
183         boost::multi_array<double, 2> B;
184         boost::multi_array<double, 2> C;
185         vector<double> Arow;
186         vector<double> Crow;
187         SetupArrays(nrows1, nrowscols12, ncols2, A, B, C, Arow, Crow, the_mpi);
188
189         MPI_Status status;
190
191         // Timing variables
192         double calc_time = 0, avg_time, total_time;
193
194         // Boss part
195         if (the_mpi.rank == 0)
196         {
197             total_time = MPI_Wtime();
198             // send one row to each worker tagged with row number, assume size<nrows
199             int rowsent = 1;
200             for (int i = 1; i < the_mpi.size; i++)
201             {
202                 MPI_Send(&A[rowsent - 1][0], nrowscols12, MPI_DOUBLE, i, rowsent, MPI_COMM_WORLD);
203                 rowsent++;
204             }
205
206             for (int i = 0; i < nrows1; i++)
207             {
208                 MPI_Recv(&Crow[0], ncols2, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
209                 int sender = status.MPI_SOURCE;
210                 int row = status.MPI_TAG - 1;
211                 memcpy(&C[row][0], &Crow[0], ncols2 * sizeof(double));
212
213                 if (rowsent - 1 < nrows1) {
214                     // send new row
215                     MPI_Send(&A[rowsent - 1][0], nrowscols12, MPI_DOUBLE, sender, rowsent, MPI_COMM_WORLD);
216                     rowsent++;
217                 } else {
218                     // tell sender no more work to do via a 0 TAG
219                     MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, sender, 0, MPI_COMM_WORLD);
220                 }
221             }
222         }
223         // Worker part: compute dot products of Arow.b until done message recieved
224         else
225         {
226             // Get a row of A
227             MPI_Recv(&Arow[0], nrowscols12, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
228             while (status.MPI_TAG != 0)
229             {
230                 for (int i = 0; i < ncols2; i++) {
231                     // work out Crow = Arow.B
232                     double c = 0;
233                     calc_time -= MPI_Wtime();
234                     for (int j = 0; j < nrowscols12; j++) {
235                         c += Arow[j] * B[j][i];
236                     }
237                     calc_time += MPI_Wtime();
238                     Crow[i] = c;
239
240                 }
241                 // Send answer of Arow.B back to boss and get another row to work on
242                 MPI_Send(&Crow[0], ncols2, MPI_DOUBLE, 0, status.MPI_TAG, MPI_COMM_WORLD);
243                 MPI_Recv(&Arow[0], nrowscols12, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
244             }
245             // cout << "Worker " << the_mpi.rank << " got kill tag\n";
246         }
247
248         if (the_mpi.rank == 0)
249             total_time = MPI_Wtime() - total_time;
250
251         MPI_Reduce(&calc_time, &avg_time, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
252         avg_time /= (the_mpi.size - 1); // Boss node doesn't do any of the calculations
253
254         if (the_mpi.rank == 0)
255         {
```

```
256             printf("Average calculation time = %.10f\n", avg_time);
257             printf("Total time = %.10f\n", total_time);
258 
259             datafile << fixed << setprecision(10);
260             datafile << nrows1 << " " << nrowscols12 << " " << ncols2 << " " <<
261                 the_mpi.size << " " << avg_time << " " << total_time << endl;
262         }
263 
264         // output c here on Boss node
265         Output(output, C, the_mpi);
266     }
```

# 3 Matrix Matrix Multiplication Using MPI and BLAS

In this section, we modify the code to compute the matrix matrix porduct from above. Instead of computing the product of a row from the first matrix and a column of the second matrix manually, we substitute the DDOT_ function from the BLAS library. As we did for the previous code, we test the code still works as can be seen in Figure 7.

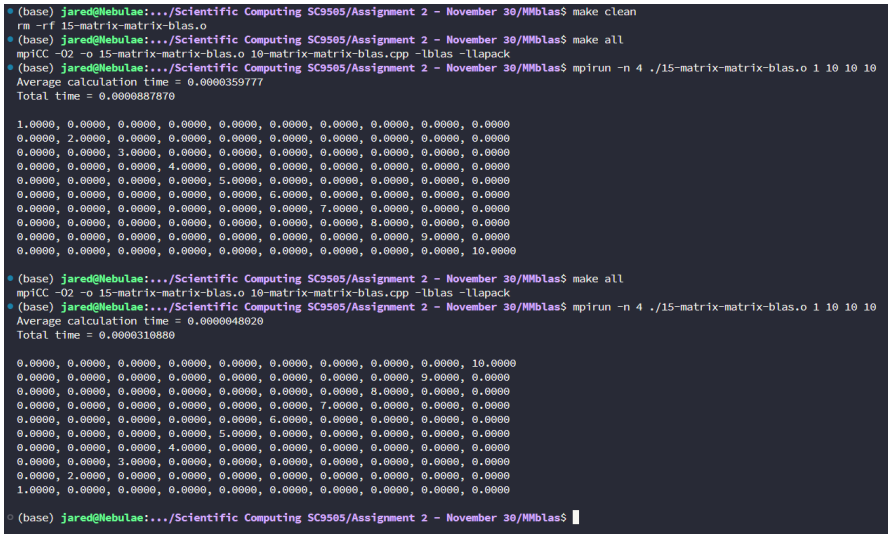



Figure 7: Demonstration of the code working. Just as we did for the non-BLAS implementation, the first example takes the first matrix to be the identity while the second has consecutive integers along the main diagonal. In the second example, the first matrix is changed to the reverse identity (1s along the antidiagonal). In both cases, we get the expected result.

The new code using BLAS shows that the total runtime as a function of the number of rows $N$ scales as $\mathcal{O}(N^{3.6})$ as supported by Figure 8. This is peculiar since this means the modified code using BLAS actually scales worse than the non-BLAS code.

Perhaps this is because the BLAS code doesn't actually scale in polynomial time and is instead scaling like $\mathcal{O}(N^{\alpha} \log N)$ for some $\alpha$, but I am not confident enough to say for certain. I would expect the

BLAS code to scale better than the non-BLAS code, and given that the linear regression does not fit as nicely as it did for the non-BLAS case, I suspect this may be the case.

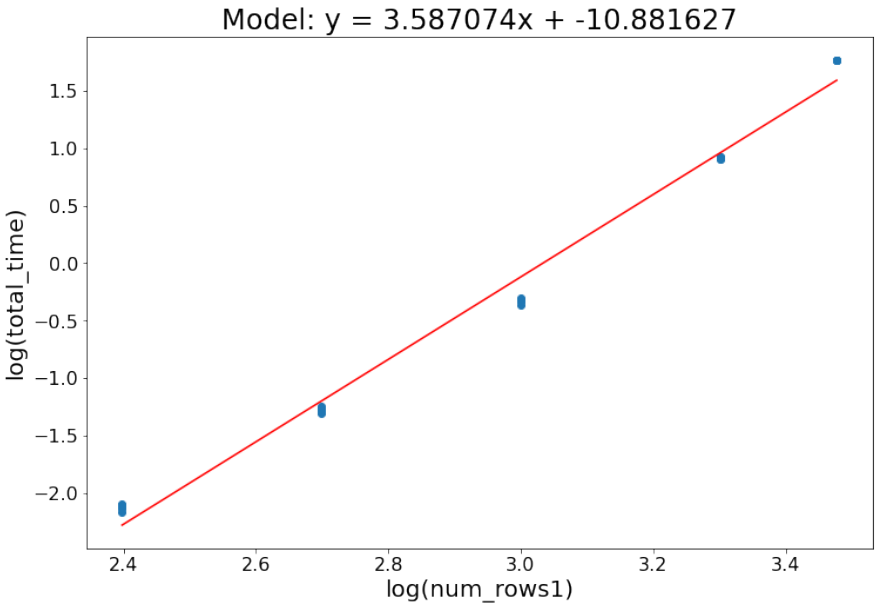



Figure 8: Plotting the runtime of the program as a function of the number of rows $N$ (on a log-log scale), we find that the total runtime scales as $\mathcal{O}(N^{3.6})$, which again is more than the expected$\mathcal{O}(N^3)$, and even worse than the non-BLAS scaling.

Considering now the scaling as a function of the number of threads $P$, we again find a similar result: that for a fixed number of rows $N$, the scaling goes as $\mathcal{O}(P^{-1})$, as can be seen in Figure 9. This is once again good news and agrees with what we would expect: the scaling should still be the same as in the non-BLAS case.

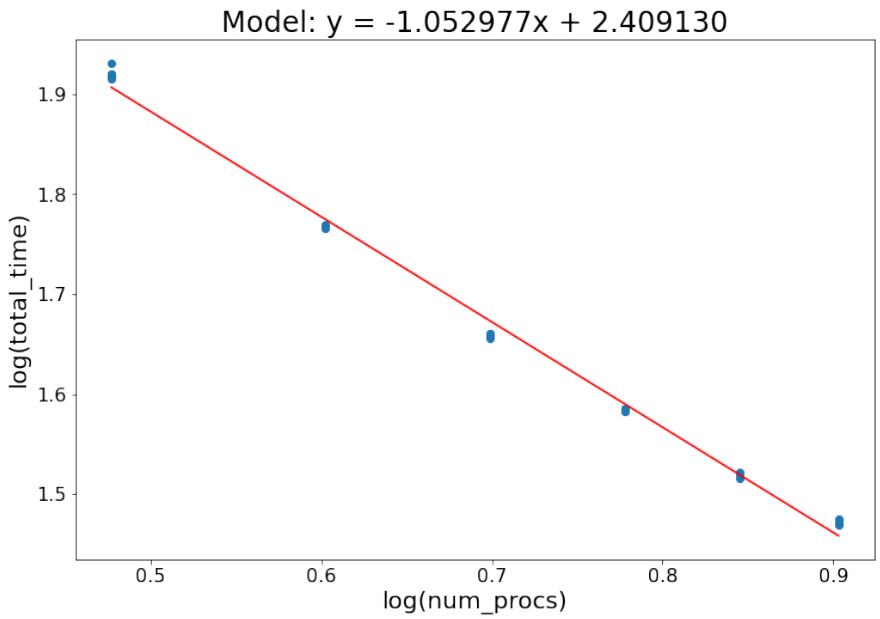



Figure 9: Plotting the runtime of the program as a function of the number of threads $P$, we find the scaling for a fixed number of rows $N$ (here $N = 3000$) to be of the order $\mathcal{O}(P^{-1})$

Looking at the speedup and efficiency as seen in Table 3, we see that our BLAS code runs faster than the non-BLAS code and is therefore more efficient as well. It also appears that that efficiency of the program increases as we allocate more and more threads. As a test, I ran the code with $N = 3000$ rows and $P = 12$ threads, and found the efficiency to drop off slightly to 0.59.

| Number of Threads | Speedup | Scaling Factor $a$ | Efficiency |
|---|---|---|---|
| 3 | 2.05 | 0.63 | 0.63 |
| 4 | 2.65 | 0.64 | 0.64 |
| 5 | 3.22 | 0.65 | 0.65 |
| 6 | 3.80 | 0.65 | 0.65 |
| 7 | 4.35 | 0.66 | 0.66 |
| 8 | 4.90 | 0.66 | 0.66 |

Table 3: Summary of the speedup and efficiency of our matrix matrix multiplication code using the BLAS DDOT_ function. These values are taken for a constant number of rows $N = 3000$.

I would expect if we ran the code for say $N = 5000$, the efficiency would be much higher, since if we look at the efficiency for $N = 2000$, as in Table 4, we see that the efficiency of the program nearly doubles going from $N = 2000$ to $N = 3000$. As a test, I ran the serial code for $N = 5000$, which took 1083.20 seconds. The parallel code ran with $P = 12$ took 128.45 seconds, corresponding to a speedup of 8.43 and an efficiency of 0.70. When run with $P = 8$ instead, it took 188.57 seconds, giving a speedup of 5.74 and an efficiency of 0.72.

| Number of Threads | Speedup | Scaling Factor $a$ | Efficiency |
|---|---|---|---|
| 3 | 1.16 | 0.39 | 0.39 |
| 4 | 1.52 | 0.38 | 0.38 |
| 5 | 1.87 | 0.37 | 0.37 |
| 6 | 2.22 | 0.37 | 0.37 |
| 7 | 2.56 | 0.37 | 0.37 |
| 8 | 2.90 | 0.36 | 0.36 |

Table 4: Summary of the speedup and efficiency of our matrix matrix multiplication code using the BLAS DDOT_ function. These values are taken for a constant number of rows $N = 2000$.

```
////////////////////////////////////////////
// Matrix-Matrix multiplication code AB=C //
////////////////////////////////////////////

// Note that I will index arrays from 0 to n-1.
// Here workers do all the work and boss just handles collating results
// and sending infor about A.

// include, definitions, globals etc here
#include <iostream>
#include <fstream>
#include <iomanip>
#include <random>
#include "boost/multi_array.hpp"
#include "mpi.h"

extern "C" {
    extern double ddot_(int *, double *, int *, double *, int *);
}

using namespace std;

class MPI_Obj
{
public:
    int size;
    int rank;

    MPI_Obj(int &argc, char **&argv)
    {
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    }

    ~MPI_Obj()
    {
        MPI_Finalize();
    }
};

void GetArraySize(
    int &output,
    int &nrows1,
    int &nrowscols12,
    int &ncols2,
    MPI_Obj &the_mpi,
    int argc,
    char **argv
) {
    if (the_mpi.rank == 0)
    {
        if (argc  == 5) {
            output = atoi(argv[1]);
            nrows1 = atoi(argv[2]);
            nrowscols12 = atoi(argv[3]);
            ncols2 = atoi(argv[4]);
        } else {
            cout << "Please enter the number of rows for the first matrix -> ";
```

```
60              cin >> nrows1;
61              cout << "Please enter the number of columns/rows for the first/second matrix-> ";
62              cin >> nrowscols12;
63              cout << "Please enter the number of columns for the second matrix-> ";
64              cin >> ncols2;
65          }
66      }
67
68      // send everyone nrows, ncols
69      int buf[3] = {nrows1, nrowscols12, ncols2};
70      MPI_Bcast(buf, 3, MPI_INT, 0, MPI_COMM_WORLD);
71      if (the_mpi.rank != 0)
72      {
73          nrows1 = buf[0];
74          nrowscols12 = buf[1];
75          ncols2 = buf[2];
76      }
77  }
78
79  void SetupArrays(
80      int nrows1,
81      int nrowscols12,
82      int ncols2,
83      boost::multi_array<double, 2> &A,
84      boost::multi_array<double, 2> &B,
85      boost::multi_array<double, 2> &C,
86      vector<double> &Arow,
87      vector<double> &Crow,
88      MPI_Obj &the_mpi
89  ) {
90      uniform_real_distribution<double> unif(-1.0, 10);
91      default_random_engine re;
92
93      B.resize(boost::extents[nrowscols12][ncols2]);
94      Crow.reserve(ncols2); Crow.resize(ncols2); // Main process will need to store the values temporarily
95
96      // Boss part
97      if (the_mpi.rank == 0)
98      {
99          // Set size of A
100         A.resize(boost::extents[nrows1][nrowscols12]);
101         C.resize(boost::extents[nrows1][ncols2]);
102
103         // Initialize A
104         for (int i = 0; i < nrows1; ++i) {
105             for (int j = 0; j < nrowscols12; ++j) {
106                 // Identity
107                 // if (i == j)
108                 //     A[i][j] = 1.0;
109                 // else
110                 //     A[i][j] = 0.0;
111                 // Reverse Identity
112                 // if (i == (nrowscols12 - j - 1))
113                 //     A[i][j] = 1.0;
114                 // else
115                 //     A[i][j] = 0.0;
116                 // Random
117                 A[i][j] = unif(re);
118             }
119         }
120
121         // Initialize B
122         for (int i = 0; i < nrowscols12; ++i) {
123             for (int j = 0; j < ncols2; ++j) {
124                 // Identity
125                 // if (i == j)
126                 //     B[i][j] = 1.0;
127                 // else
128                 //     B[i][j] = 0.0;
129                 // Reverse Identity
130                 // if (i == (ncols2 - j - 1))
131                 //     B[i][j] = 1.0;
132                 // else
133                 //     B[i][j] = 0.0;
134                 // Random
135                 // B[i][j] = unif(re);
136                 // Other
```

```
                    if (i == j)
                        B[i][j] = (double) i + 1;
                    else
                        B[i][j] = 0;
                }
            }
        } else {
            // Worker part
            // Allocate space for 1 row of A and 1 row of the answer C
            Arow.reserve(nrowscols12); Arow.resize(nrowscols12);
        }
        MPI_Bcast(&B[0][0], nrowscols12*ncols2, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }

    void Output(int output, boost::multi_array<double, 2> &array, MPI_Obj &the_mpi)
    {
        if (the_mpi.rank == 0 && output) {
            cout << endl << fixed << setprecision(4);

            for (int i = 0; i < array.shape()[0]; i++) {
                for (int j = 0; j < array.shape()[1]; j++) {
                    cout << array[i][j];
                    if (j < array.shape()[1] - 1) cout << ", ";
                }
                cout << endl;
            }
            cout << endl;

            cout << scientific;
        }
    }

    int main(int argc, char **argv)
    {
        // Data File
        // ofstream datafile("/home/jared/Desktop/mm-timings.txt", ios_base::app);
        ofstream datafile("mmblas-timings.txt", ios_base::app);

        // initialize MPI
        MPI_Obj the_mpi(argc, argv);
        if (the_mpi.size < 2) MPI_Abort(MPI_COMM_WORLD, 1);

        // determine/distribute size of arrays here
        int output = 1, nrows1 = 0, nrowscols12 = 0, ncols2 = 0;
        GetArraySize(output, nrows1, nrowscols12, ncols2, the_mpi, argc, argv);
        if (the_mpi.size - 1 > nrows1) {
            MPI_Abort(MPI_COMM_WORLD, -1);
        }

        boost::multi_array<double, 2> A;
        boost::multi_array<double, 2> B;
        boost::multi_array<double, 2> C;
        vector<double> Arow;
        vector<double> Crow;
        SetupArrays(nrows1, nrowscols12, ncols2, A, B, C, Arow, Crow, the_mpi);

        MPI_Status status;

        // Timing variables
        double calc_time = 0, avg_time, total_time;

        // Boss part
        if (the_mpi.rank == 0)
        {
            total_time = MPI_Wtime();
            // send one row to each worker tagged with row number, assume size<nrows
            int rowsent = 1;
            for (int i = 1; i < the_mpi.size; i++)
            {
                MPI_Send(&A[rowsent - 1][0], nrowscols12, MPI_DOUBLE, i, rowsent, MPI_COMM_WORLD);
                rowsent++;
            }

            for (int i = 0; i < nrows1; i++)
            {
                MPI_Recv(&Crow[0], ncols2, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                int sender = status.MPI_SOURCE;
```

```
            int row = status.MPI_TAG - 1;
            memcpy(&C[row][0], &Crow[0], ncols2 * sizeof(double));

            if (rowsent - 1 < nrows1) {
                // send new row
                MPI_Send(&A[rowsent - 1][0], nrowscols12, MPI_DOUBLE, sender, rowsent, MPI_COMM_WORLD);
                rowsent++;
            } else {
                // tell sender no more work to do via a 0 TAG
                MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, sender, 0, MPI_COMM_WORLD);
            }
        }
    }
    // Worker part: compute dot products of Arow.b until done message recieved
    else
    {
        // Get a row of A
        MPI_Recv(&Arow[0], nrowscols12, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        while (status.MPI_TAG != 0)
        {
            for (int i = 0; i < ncols2; i++) {
                // work out Crow = Arow.B
                calc_time -= MPI_Wtime();
                Crow[i] = ddot_(&nrowscols12, &Arow[0], new int(1), &B[0][i], &ncols2);
                calc_time += MPI_Wtime();

            }
            // Send answer of Arow.B back to boss and get another row to work on
            MPI_Send(&Crow[0], ncols2, MPI_DOUBLE, 0, status.MPI_TAG, MPI_COMM_WORLD);
            MPI_Recv(&Arow[0], nrowscols12, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        }
    }

    if (the_mpi.rank == 0)
        total_time = MPI_Wtime() - total_time;

    MPI_Reduce(&calc_time, &avg_time, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    avg_time /= (the_mpi.size - 1); // Boss node doesn't do any of the calculations

    if (the_mpi.rank == 0)
    {
        printf("Average calculation time = %.10f\n", avg_time);
        printf("Total time = %.10f\n", total_time);

        datafile << fixed << setprecision(10);
        datafile << nrows1 << " " << nrowscols12 << " " << ncols2 << " " <<
            the_mpi.size << " " << avg_time << " " << total_time << endl;
    }

    // output c here on Boss node
    Output(output, C, the_mpi);
}
```

# 4 Solving Poisson's Equation Using LAPACK

The last part of the assignment is a brief introduction to the LAPACK library. The code being tested solves the Poisson equation $\nabla^2 U(\vec{x}) = f(\vec{x})$. In Figure 10, I have run the provided code and plotted the solution using plotly in Python.

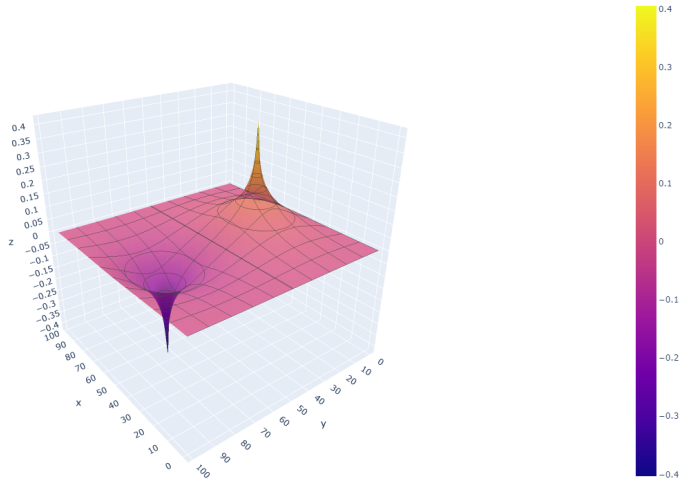



Figure 10: Plot of the solution using $N = 100$ sites.

Testing the run time of the code as a function of the number of sites $N$, the scaling is found to be of the order $\mathcal{O}(N^{3.8})$, as can be seen in Figure 11. The scaling can be expected to be $\mathcal{O}(N^4)$, as performing a PLU factoraization scales as $\mathcal{O}(N^3)$, and back-substitution should contribute a factor proportional to $\mathcal{O}(N)$. Given we are using BLAS routines which are optimized for banded and positive definite matrices, it is convincing our scaling should be slightly less than the worst case scenario of $\mathcal{O}(N^4)$.

Using the linear regression model (red line) in the plot, we can estimate that the largest system that can be solved (on my hardware) in 5 minutes to be roughly $N = 975$. This seems reasonable, since the runtime for a system with $N = 1000$ was roughly 385 seconds.

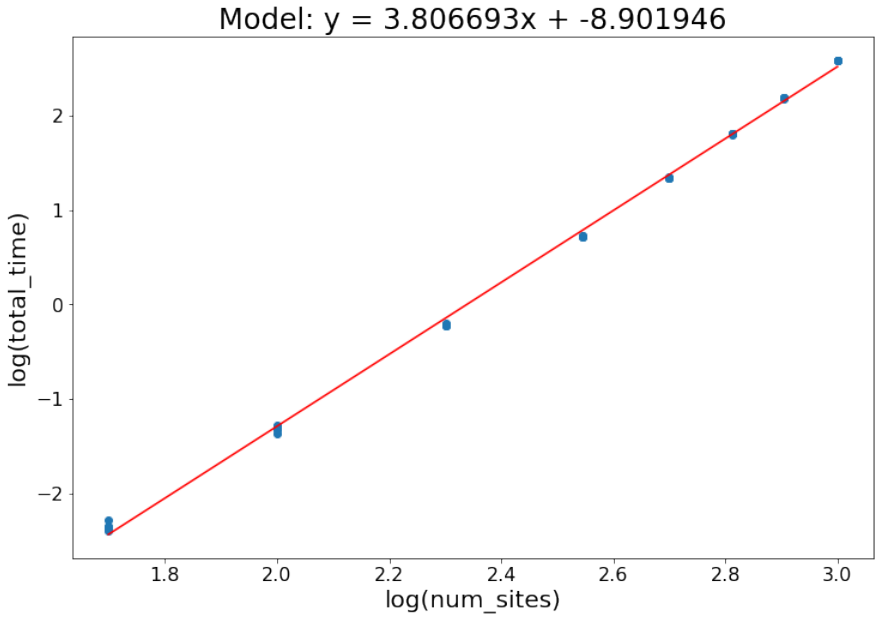



Figure 11: Plotting the runtime of the program as a function of the number of sites $N$ (on a log-log scale), we find that the total runtime scales as $\mathcal{O}(N^{3.8})$.

```
// Program to compute solution to -laplacian u = -F
// compile with
// run with mpirun -n 1 <executable> output_cout output_file N

#include <iostream>
#include <iomanip>
#include <fstream>
#include <mpi.h>
#include "boost/multi_array.hpp"

// LAPACK library files
extern "C"
{
    // general band factorize routine
    extern int dpbtrf_(char *, int *, int *, double *, int *, int *);
    // general band solve using factorization routine
    extern int dpbtrs_(char *, int *, int *, int *, double *, int *, double *, int *, int *);
}

void AbInit(int N, boost::multi_array<double, 2> &Ab)
{
    // Coefficient Matrix: initialize
    for (int i = 0; i < N * N; i++)
        for (int j = 0; j < N + 1; j++)
        {
            Ab[j][i] = 0.0;
            if (j == 0)
                Ab[j][i] = -1.0;
            if (j == N - 1 && i % N)
                Ab[j][i] = -1.0;
            if (j == N)
                Ab[j][i] = 4.0;
        }
    // std::cout << "Ab initialized \n";
    //  Printout Ab for testing, small N only
    // for (int i=0; i < N+1; i++) {
    //   std::cout << Ab[i][0];
    //   for (int j=1; j < N*N; j++) {
    //     std::cout << " " << Ab[i][j];
    //   }
    //   std::cout << "\n";
    // }
}
```

```
44
45  void RHSInitialize(int N, std::vector<double> &F, double bcx, double bcy)
46  {
47      // RHS: fill in boundary condition values
48      for (int i = 0; i < N; i++)
49      {
50          F[i] += bcx;             // bottom boundary
51          F[N * N - i - 1] += bcx; // top boundary
52          F[i * N] += bcy;         // left boundary
53          F[i * N + N - 1] += bcy; // right boundary
54      }
55      // std::cout << "RHS initialized\n";
56
57      // RHS: fill in actual right-hand side, some "charges", actually h^2*charge
58      F[N / 4 * N + N / 2] += 0.5;
59      F[3 * N / 4 * N + N / 2] += -0.5;
60  }
61
62  int main(int argc, char** argv)
63  {
64      double time = MPI_Wtime();
65      // ofstream timingfile("/home/jared/Desktop/lp-timings.txt", ios_base::app);
66      std::ofstream timingfile("lp-timings.txt", std::ios_base::app);
67
68      // ofstream datafile("/home/jared/Desktop/lp-data.txt");
69      std::ofstream datafile("lp-data.txt");
70
71      // Coefficient Matrix: declare
72      int output_cout = 1, output_file = 0;
73      int N = 10;
74      if (argc == 4) {
75          output_cout = atoi(argv[1]);
76          output_file = atoi(argv[2]);
77          N = atoi(argv[3]);
78      }
79
80      int M = N + 1;
81      int ABcols = N * N;
82      boost::multi_array<double, 2> Ab(boost::extents[M][ABcols], boost::fortran_storage_order());
83
84      AbInit(N, Ab); // Initialize coefficient matrix
85
86      // Coefficient Matrix: factorize
87      char uplo = 'U';
88      int KD = N;
89      int info;
90      dpbtrf_(&uplo, &ABcols, &KD, &Ab[0][0], &M, &info);
91      if (info)
92      {
93          std::cout << "Ab failed to factorize, info = " << info << "\n";
94          exit(1);
95      }
96
97      // RHS: declare
98      const double bcx = 0.0, bcy = 0.0; // boundary conditions along x and y assume same on both sides
99      std::vector<double> F(N * N, 0.0);
100
101     RHSInitialize(N, F, bcx, bcy); // set up boundary conditions and right hand side
102
103     // Solve system
104     int Bcols = 1;
105     dpbtrs_(&uplo, &ABcols, &KD, &Bcols, &Ab[0][0], &M, &F[0], &ABcols, &info);
106     if (info)
107     {
108         std::cout << "System solve failed, info = " << info << "\n";
109         exit(1);
110     }
111
112     time = MPI_Wtime() - time;
113     timingfile << N << " " << time << std::endl;
114
115     // Output solution
116     if (output_cout) {
117         for (int i = 0; i < N; i++)
118         {
119             std::cout << F[i * N];
120             for (int j = 1; j < N; j++)
```

```
121                 {
122                     std::cout << " " << F[i * N + j];
123                 }
124                 std::cout << "\n";
125             }
126         }
127         if (output_file) {
128             datafile << std::fixed << std::setprecision(10);
129             for (int i = 0; i < N; i++)
130             {
131                 datafile << F[i * N];
132                 for (int j = 1; j < N; j++)
133                 {
134                     datafile << " " << F[i * N + j];
135                 }
136                 datafile << "\n";
137             }
138         }
139 
140         return 0;
141     }
```

# A Jupyter Notebook

Below is an example notebook I have used in order to generate and analyze my timing data. The notebook will compile and run the given program a number of times for various numbers of threads and matrix dimensions. It then creates linear regression models from the data which can be used to determine the speedup and efficiency of parallelization.

# plots

November 21, 2022

[ ]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import pandas as pd
import subprocess as sp
import itertools as it
```

[ ]:
```
def genData(procs, sizes, data_file, cycles=20, clean=False):
    sp.run(
        "/usr/bin/mpiCC -O2 -o 15-matrix-vector.o ./10-matrix-vector.cpp",
        shell=True,
        stdout=sp.DEVNULL,
        stderr=sp.DEVNULL
    )
    if clean:
        sp.run(
            f'echo "num_rows num_cols num_procs avg_time total_time" >␣
 ↪{data_file}',
            shell=True,
            stdout=sp.DEVNULL,
            stderr=sp.DEVNULL
        )

    for size, proc in it.product(sizes, procs):
        for _ in range(cycles):
            sp.run(
                f"/usr/bin/mpirun -n {proc} ./15-matrix-vector.o 0 {size}␣
 ↪{size}",
                shell=True,
                stdout=sp.DEVNULL,
                stderr=sp.DEVNULL
            )
        print(f"Finished {size}x{size} with {proc} processes")

def plotModel(
    data: pd.DataFrame,
    xdata: str,
```

```
    ydata: str,
    plot: bool,
    loglog: bool
):
    x = data[xdata].values.reshape(-1, 1)
    y = data[ydata].values.reshape(-1, 1)
    xname = xdata
    yname = ydata
    if loglog:
        x = np.log10(x)
        y = np.log10(y)
        xname = "log(" + xdata + ")"
        yname = "log(" + ydata + ")"

    linear_model = LinearRegression()
    linear_model.fit(x, y)
    model = linear_model.predict(x)

    if plot:
        fig, ax = plt.subplots(figsize=(12,8))
        ax.set_title("Model: y = {:.6f}x + {:.6f}".format(linear_model.
 ↪coef_[0][0], linear_model.intercept_[0]), size=24)
        ax.set_xlabel(xname, size=20)
        ax.tick_params(axis='x', labelsize=16)
        ax.set_ylabel(yname, size=20)
        ax.tick_params(axis='y', labelsize=16)
        ax.scatter(x, y)
        ax.plot(x, model, color='red')
        plt.show()

    return linear_model
```

[ ]:
```
# Variables
# data_file = "/home/jared/Desktop/mv-timings.txt"
data_file = "./mv-timings.txt"
procs = np.array([3, 4, 5, 6, 7, 8])
sizes = np.array([10, 50, 100, 250, 500, 1000, 10000])

# Generate data if needed
# genData(procs=procs, sizes=sizes, data_file=data_file, clean=True)

# Load the data
data = pd.read_csv(data_file, delimiter=" ")
```

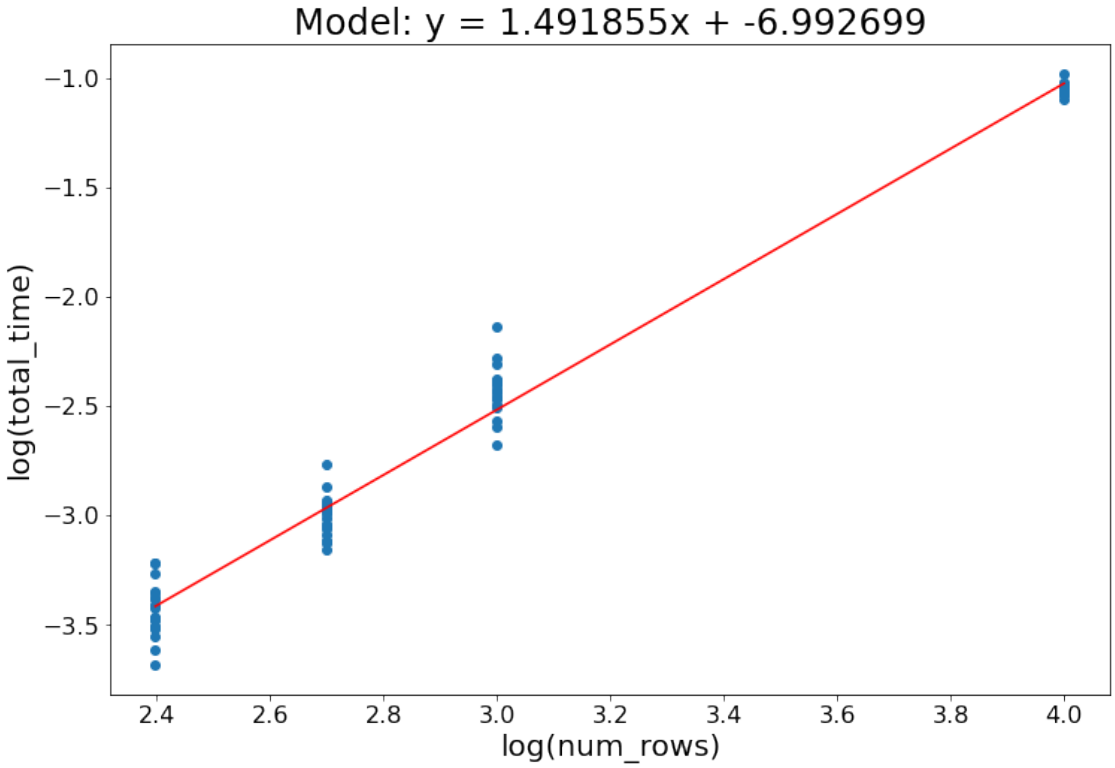
[ ]:
```
num_procs = 4
num_rows = 10000
min_procs = 2
```

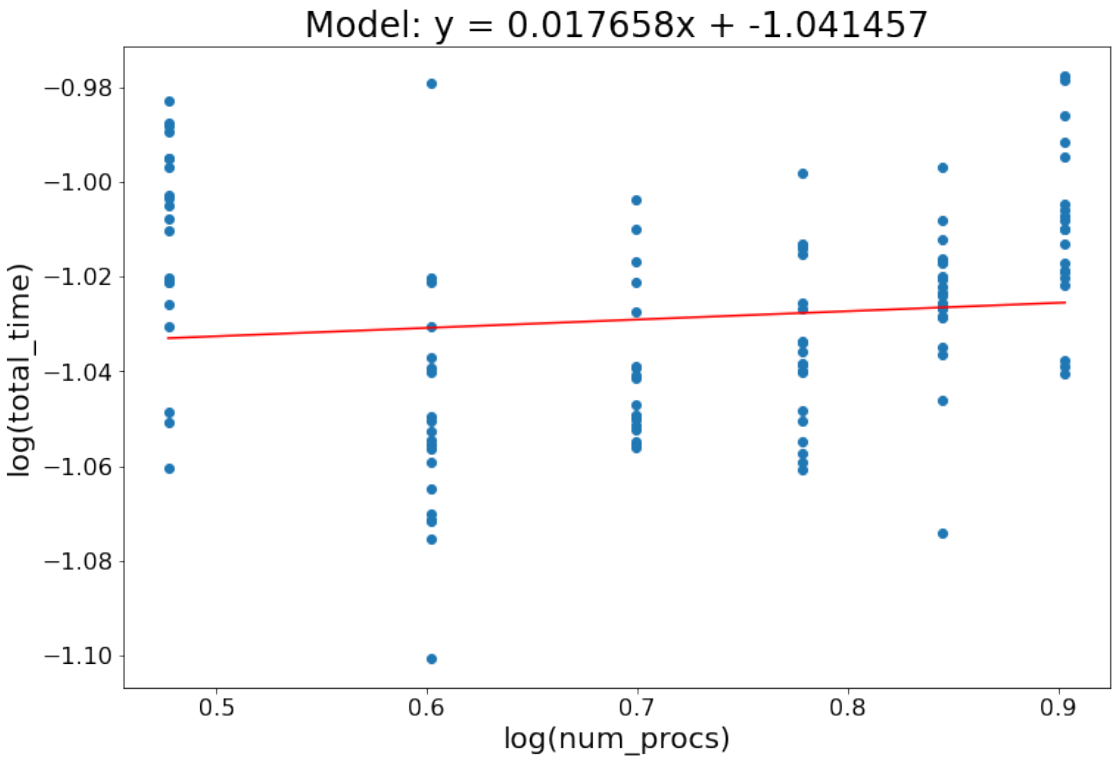
```
min_rows = 100

data_reduced_row_scaling = data[data["num_procs"] == num_procs]
data_reduced_row_scaling =␣
 ↪data_reduced_row_scaling[data_reduced_row_scaling["num_rows"] > min_rows]

data_reduced_proc_scaling = data[data["num_rows"] == num_rows]
data_reduced_proc_scaling =␣
 ↪data_reduced_proc_scaling[data_reduced_proc_scaling["num_procs"] > min_procs]
```

[ ]:
```
reduced_row_scaling_model = plotModel(data_reduced_row_scaling, "num_rows",␣
 ↪"total_time", plot=True, loglog=True)
reduced_proc_scaling_model = plotModel(data_reduced_proc_scaling, "num_procs",␣
 ↪"total_time", plot=True, loglog=True)
```






[ ]:
```
num_rows = 10000

serial_times = {
    10: 0.0000047090,
    50: 0.0000092870,
    100: 0.0000360770,
    250: 0.0001662810,
    500: 0.0008027130,
    1000: 0.0025987250,
    10000: 0.4020480850,
}

prediction = np.array([
    10**val[0] for val in
    reduced_proc_scaling_model.predict(procs.reshape(-1, 1))
])

speedup = serial_times[num_rows] / prediction
print("speedup = \n", np.array_str(speedup))

a = speedup / procs
print("a = \n", np.array_str(a))
```

```
efficiency = speedup / procs
print("efficiency = \n", np.array_str(efficiency))
```

```
speedup =
 [3.91524996 3.75924844 3.60946274 3.46564519 3.32755799 3.19497282]
a =
 [1.30508332 0.93981211 0.72189255 0.57760753 0.47536543 0.3993716 ]
efficiency =
 [1.30508332 0.93981211 0.72189255 0.57760753 0.47536543 0.3993716 ]
```