Linear Algebra Optimization in C using OpenMPI

Aidan Levy
Dr. Spickler
May 12th, 2025

1 Introduction

In today's data driven world, there is a constant demand for faster and most efficient computing methods. Traditional sequential computing, where tasks are executed one after another by a singular processor, often falls short when tackling complex and large scale problems. While this model remains useful for certain tasks and small scale applications, it fails to meet the performance needs of large scale computations in modern research, analytics, and simulation based fields.

Parallel computing addresses this limitation by distributing workload across multiple processors or cores, enabling simultaneous execution of sub tasks. This approach significantly reduces execution time and increases efficiency, especially for complex mathematical operations and data heavy computations. Parallelism has become integral in many different cases– from national defense and physics simulations to artificial intelligence and high frequency trading– where performance constraints dictate the outcomes of solutions (Barney & Frederick). The Open Message Passing Interface (OpenMPI) implementation has become a core element of solving this problem by facilitating communication and coordination among distributed processes. OpenMPI is widely used in both academic and industry settings due to its scalability, ease of use, and support for high performance clusters.

This research project investigates the use of OpenMPI to optimize linear algebra algorithms written in C, which is a very low level language. Linear algebra remains a central pillar of computer science, with applications ranging from computer graphics and machine learning to economic modeling and engineering. One historic example includes Professor Leontief's use of linear equations in 1949 to model the structure of the United States economy, which demonstrated the importance of these systems (Lay et al., 2020). In this study, emphasis is placed on the efficient execution of operations such as matrix multiplication, row echelon form (REF), reduced row echelon form (RREF), and LU decomposition, which serve as critical tools in solving systems of linear equations. These algorithms are implemented in both sequential and parallel forms to allow for direct performance comparisons. Timing benchmarks collected during testing reveal the extent to which parallelism accelerates execution and under what conditions those gains are most significant. These performance results will be discussed in detail in later sections.

In addition to algorithm optimization, the project involved constructing a custom parallel computing environment. The process of configuring and deploying a multi node OpenMPI cluster provided valuable learning experiences into the logistics and technical requirement of distributed computing. This included configuration of operating systems, secure shell (SSH) communication setup between nodes, implementation of shared storage, and fine tuning of the scripts. The resulting cluster not only enabled controlled benchmarking of parallel code but also served as a replicable model for lightweight high performance computing environments.

2 Terminology and MPI Overview

To aid in understanding the parallel implementations discussed in this study, this section defines key terms and MPI (message passing interface) functions that appear throughout the

algorithms discussed. MPI, message passing interface, is a standardized and portable message passing system that allows processes to run on distributed memory systems. The CPUs are physically separated from each other, but still communicate and transmit data between themselves. According to Spickler (2025), the following diagram displays how distributed memory systems look:

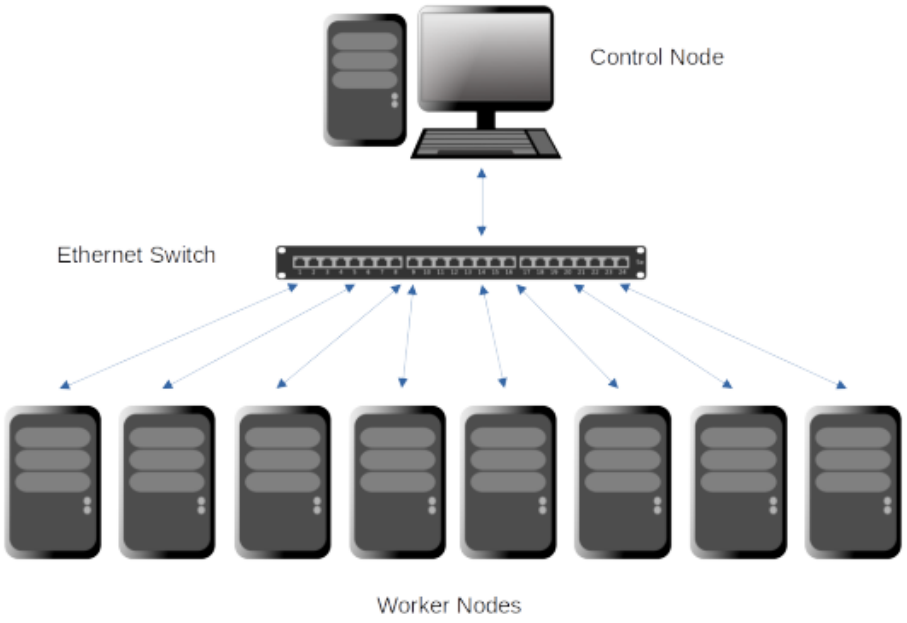

Figure 1: Data Flow in OpenMPI Cluster

An MPI process is a single instance of a running program in an MPI application. Each process has a unique rank, which is an integer associated with that process, separating it from the other processes. Rank 0 is typically the root process, the one that initializes the program. MPI_Bcast is a broadcast function that is used to send data from one process, usually the root, to all other processes in a communicator. This ensures that all processes have the same updated data before continuing. Without the usage of MPI_Bcast, the data would now be shared across all nodes. The following diagram displays how MPI_Bcast distributes memory:

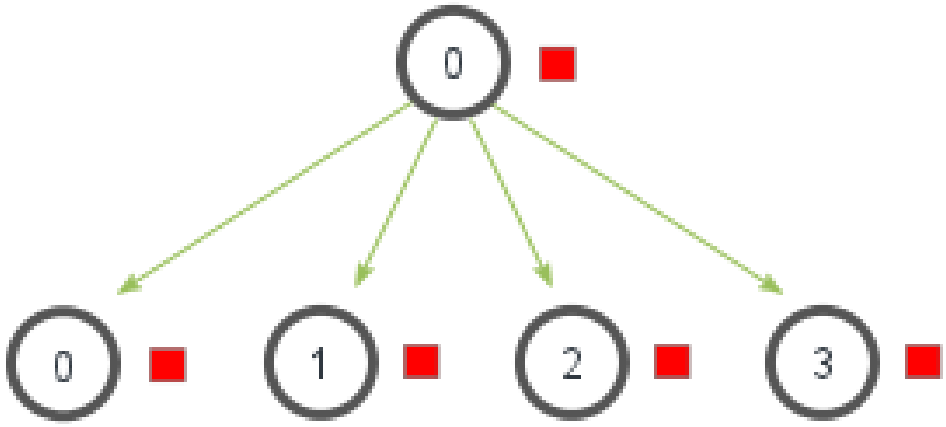

Figure 2: Data Flow using MPI_Bcast

Additionally, MPI_Barrier is a synchronization function that blocks all processes until every process in the communicator has reached the barrier. This makes sure that all previous operations are complete across all processes before proceeding. While this tool is necessary, large overhead can occur because it pauses all processes until every one is complete. In large matrices, this build up can accumulate over time. The following diagram demonstrates how MPI_Barrier waits for all processes to complete:

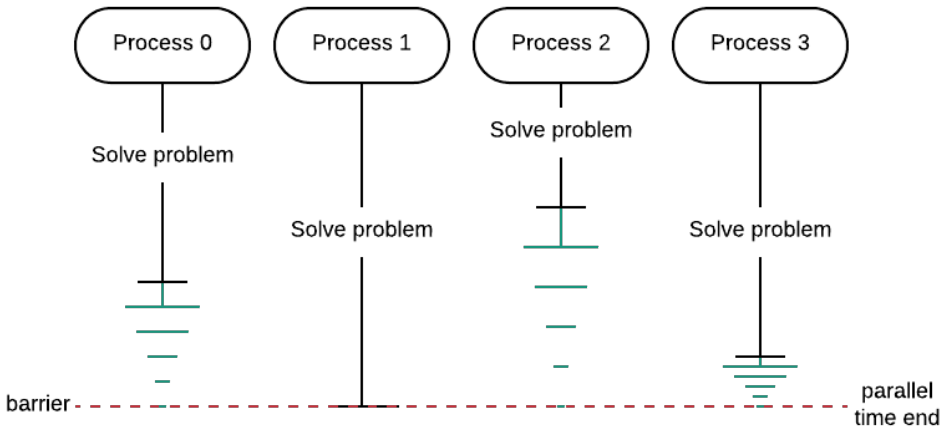

Figure 3: Data Flow using MPI_Barrier

MPI_Allreduce is a collective communication operation that sums values from all running processed using a specified operation, and distributes the result back to all processes. This command is used to pull all results together and re-send data back out to all processes. Here is a diagram displaying how MPI_Allreduce works:

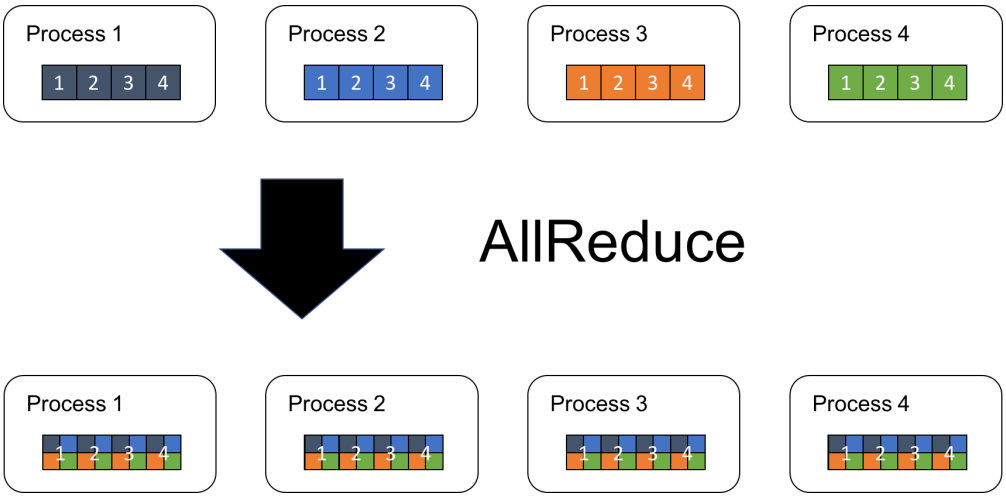

Figure 4: Data Flow using MPI_Allreduce

3 Mathematics and Algorithms

The implementation of parallelized linear algebra routines requires not only a theoretical understanding of the algorithms but also practical insight into how these operations work under distributed computation. This section outlines the core mathematical algorithms used in the project: matrix multiplication, REF, RREF, and LU decomposition, as well as comparing the differences between their sequential and OpenMPI forms. Matrix multiplication is a foundational operation in linear algebra with applications ranging from linear transformations to machine learning. Given two matrices, A of size m x n and B of size n x p, the produce C = A*B is defined as:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

In the sequential implementation, three nested loops are used to iterate through each row i of A, each column j of B, and each inner dimension k, summing the partial products in Ci,j. The parallel implementation distributes the row range of matrix A across all MPI processes. Each process computes a subset of the output matrix C, then gets called by MPI_Allreduce to consolidate the partial results across all nodes. Load balancing is managed by evenly dividing rows, and the implementations accounts for division with remainders by assigning an extra row to the first few ranks. This design allows each core to perform independent multiplication on the assigned rows, significantly reducing computation time while leveraging all available nodes. The correctness of results is guaranteed by the summation of MPI_Allreduce.

Additionally, the REF form of a matrix is another fundamental transformation in linear algebra, and is commonly used for solving systems of linear equations and preparing a matrix for further operations such as back substitution, RREF conversion, and LU decomposition. A matrix is said to be in row echelon form when all nonzero rows are above any rows of all zeroes; the leading coefficient (pivot) of a nonzero row is strictly to the right of the pivot in the row above it; and all entries below a pivot are zero. The sequential algorithm follows the

Gaussian elimination process. The function iteratively scans for nonzero pivots, performs row swaps if needed, normalizes the pivot row so the leading entry becomes 1, and eliminates all entries below the pivot by subtracting a multiple of the pivot row from each target row. In the parallel implementation, the primary logic is preserved, but the elimination step is distributed across multiple MPI processes. Each process is responsible for updating a subset of rows. More specifically, rows are assigned based on their index modulus the total number of MPI processes:

$$\text{If } i \bmod p = \text{rank}, \text{ then process } \texttt{rank} \text{ handles row } i.$$

Each process waits until the pivot row is broadcast from rank 0, then applies the elimination step locally to its assigned rows. After completing a round of elimination, processes synchronize using MPI_Barrier and then broadcast the updated matrix using MPI_Bcast to ensure all nodes have a consistent view before moving to the next pivot row. This approach preserves correctness while achieving partial parallelism. Although some overhead exists due to the global communication at each iteration, the performance gains from dividing the row work outweigh the costs, especially on large matrices. Building upon the REF, RREF represents a stricter standard that fully solves a system of linear equations in a matrix form. In addition to the requirements for REF, RREF imposes two additional conditions: Each leading '1' is the only nonzero entry in its column; and each leading '1' is the first nonzero entry in its row. The RREF algorithm is typically used as a second phase after REF. Once the matrix is in upper triangular form, the algorithm proceeds in reverse, starting from the bottom row and moving upwards, eliminating entries above each pivot. In sequential form, the implementation iterations over the rows in reverse order. For each row, it locates the pivot, then eliminates all entries above that pivot by subtracting multiples of the pivot row from each row above. This back substitution stage ensures that each pivot is isolated. In the parallel implementation using OpenMPI, the reverse pass is distributed similarly to REF. After converting the matrix to REF form, the backward elimination steps assigns work to MPI processes based on row indices. Each process eliminates entries above pivots in rows it is responsible for, using the modulus rank pattern seen before. Once a set of updates is applied, all processes synchronize with MPI_Barrier, and the updated matrix is broadcasted again using MPI_Bcast to maintain consistency. While this introduces communication overhead, particularly in the upper triangular back propagation phase, the division of labor across processors still yields speedup in large matrices.

Furthermore, LU decomposition is another rudimentary matrix operation used to repeatedly solve systems of linear equations, especially when the coefficient matrix remains constant. The process factors a square matrix, A, into two triangular matrices– a lower triangular matrix L and an upper triangular matrix U such that A = L * U. In the sequential algorithm, Gaussian elimination is used to progressively zero out entries below the pivots, simultaneously creating the L matrix with the multipliers used during the elimination process and assigning the resulting upper triangle to U. This involves searching through each pivot position, calculating scalars for the rows below, and updating both matrices in place. In the parallel OpenMPI version, the decomposition is distributed across multiple processes by partitioning the matrix rows and assigning each subset to a different rank. During each iteration, the pivot row is broadcast from the root process to all other using MPI_Bcast,

and each process computes updates for its assigned rows independently. As with REF and RREF, synchronization is required after each iteration using MPI_Barrier to ensure consistent progress. The challenge in the parallel LU decomposition lies in how the managing data dependencies between the L and U matrices are handled.

4 Results

To evaluate the performance of sequential versus parallel implementations, a series of timing benchmarks were conducted on square matrices of varying sizes. Each algorithm– matrix multiplication, REF, RREF, and LU decomposition– was tested using both a single process mode and a multi processes OpenMPI mode on the constructed cluster. Matrix sized ranged from 10 x 10 to 2,000 x 2,000 and execution times were recorded for each operation. The results revealed that parallel matrix multiplication, REF, and RREF achieved significant speedups compared to their sequential counterparts, particularly as matrix size increased. For example, on a REF test of a 2,000 x 2,000 matrix, the sequential algorithm took 9.422 seconds, whereas the parallel process with 8 processes took 1.971 seconds to compute, which is roughly a 4.75x speedup. Similarly, the parallel RREF and matrix multiplication implementations showed substantial time reductions on larger matrices, although some communication overhead was observed during pivot synchronization. However, the parallel LU decomposition algorithm was almost consistently slower than the sequential version across all tested matrix sizes. For example, on a 2,000 x 2,000 matrix, the sequential time for LU Decomposition took 8.153 seconds, whereas the parallel time with 8 processes took 19.306 seconds. As discussed earlier, the method in which the algorithm has to be built and the frequent use of MPI_Bcast and MPI_Barrier caused the communication and synchronization overhead to dominate execution time, negating the benefits of multiprocessing.

Below are the timing results of all four operations being tested on the cluster in the HPCL. Variables that could affect the speeds were minimized— for example, there were no background processes running during any of the runtime tests. As seen, the sequential algorithms run slower in matrix multiplication, REF, and RREF compared to the parallel versions, whereas the sequential algorithm for LU Decomposition runs faster in comparison to the parallel version.

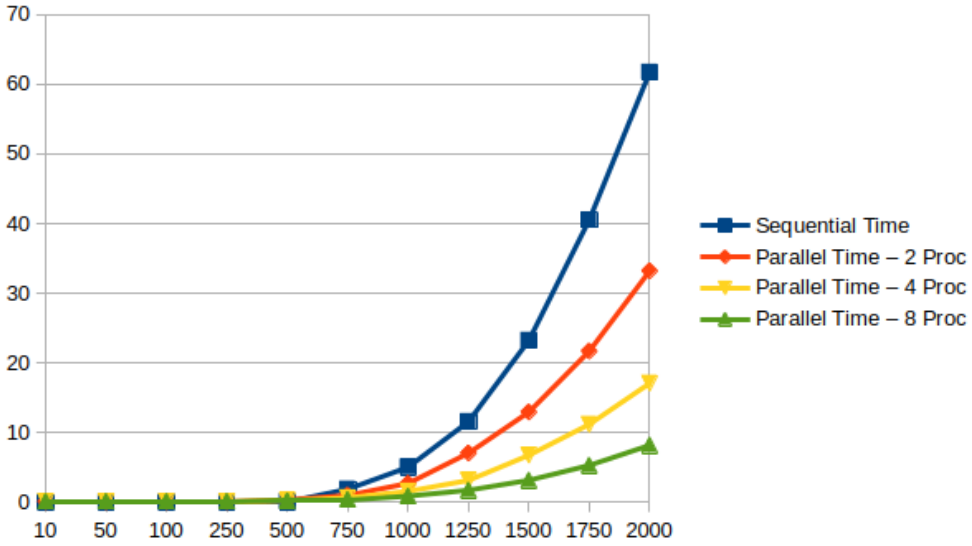

Figure 5: Matrix Multiplication Timings

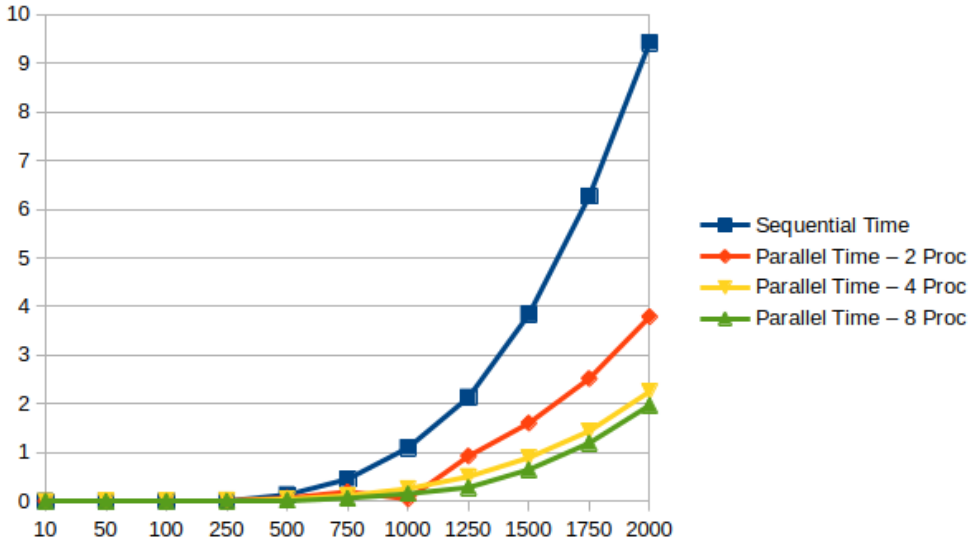

Figure 6: REF Timings

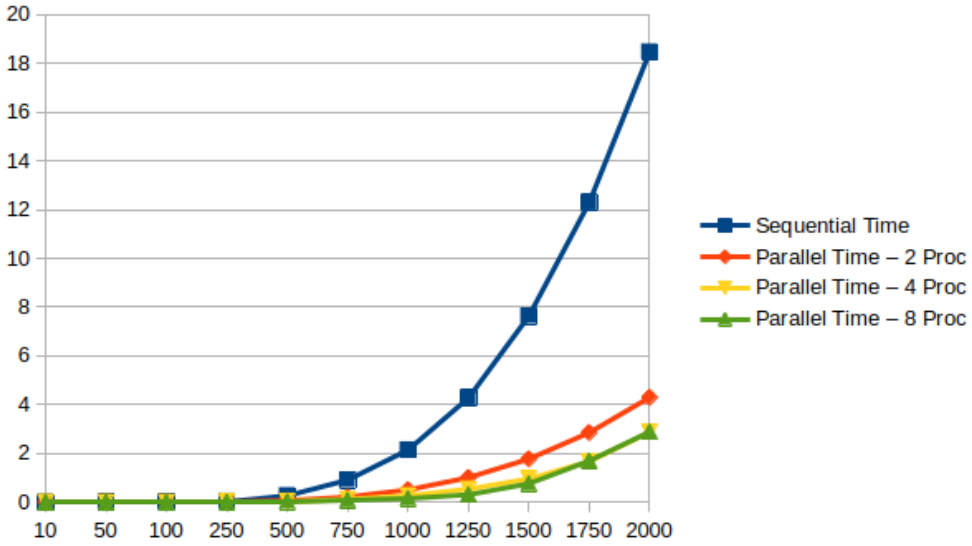

Figure 7: RREF Timings

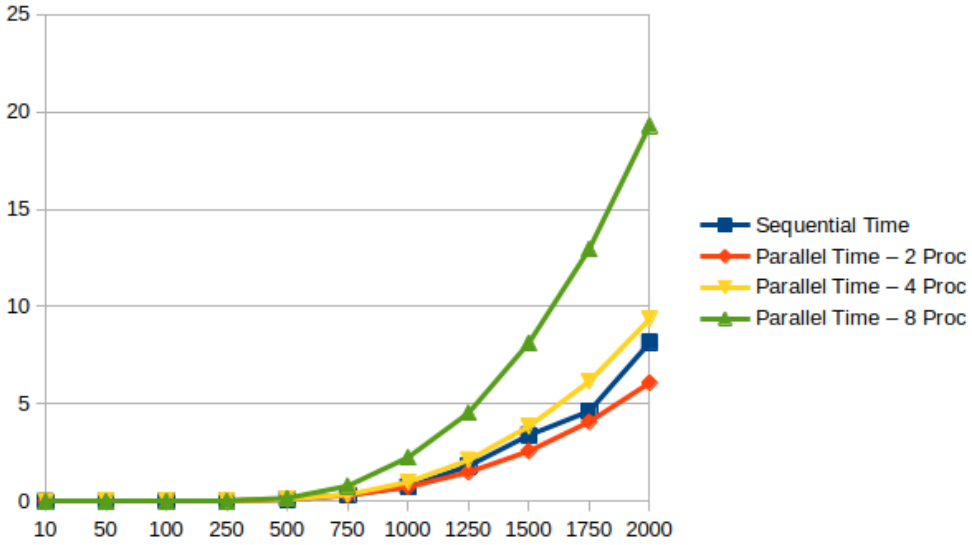

Figure 8: LU Decomposition Timings

5 Conclusion

The study demonstrated that parallel computing using OpenMPI can significantly accelerate linear algebra operations that naturally allow independent work distribution, such as matrix multiplication, REF, and RREF. Timing benchmarks confirmed that parallel implementations of these algorithms achieved substantial speedup compared to sequential versions, particularly as matrix sizes increased. However, not all algorithms benefited equally from parallelization. LU decomposition, which has strong data dependencies between elimination

steps, experienced performance degradation in the parallel implementation due to communication and synchronization overhead. This highlighted a key limitation in naive parallelization strategies– operations that frequently require inter process communication may not scale well without more sophisticated techniques or algorithms. Overall, this project provided valuable insights into both the potential and challenges of parallel computing. Future work could explore more advanced decomposition methods, such as block LU decomposition, or hybrid models that combine MPI with multithreading to better balance computation and communication.

References

[1] Barney, B., & Frederick, D. (n.d.). *Introduction to parallel computing tutorial*. Lawrence Livermore National Laboratory. Retrieved April 21, 2025, from `https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial`

[2] Lay, D. C., Lay, S. R., & McDonald, J. J. (2020). *Linear algebra and its applications* (6th ed.). Pearson.

[3] Spickler, D. (2025). *COSC 420: High-Performance Computing*. Salisbury University.

MatrixC.c

=1

1

```c
#include "MatrixC.h"

/*
Name: error()
Parameters: const char*
Return: void
Description: Prints error message and exits program
*/
void error(const char *message)
{
  fprintf(stderr, "Error: %s\n", message);
  exit(EXIT_FAILURE);
}

/*
Name: isValid()
Parameters: const Matrix*
Return: bool
Description: Returns true if cols and rows are greater than 0
*/
bool isValid(const Matrix *m)
{
  return (m->rows > 0 && m->cols > 0);
}

/*
Name: getRows()
Parameters: const Matrix*
Return: int
Description: Returns the number of rows in a matrix
*/
int getRows(const Matrix *m) { return m->rows; }

/*
Name: getCols()
Parameters: const Matrix*
Return: int
Description: Returns the number of cols in a matrix
*/
int getCols(const Matrix *m) { return m->cols; }

```

```
42 /*
43 Name: init()
44 Parameters: Matrix*
45 Return: void
46 Description: Default constructor
47 */
48 void init(Matrix *m)
49 {
50   m->rows = 0;
51   m->cols = 0;
52 }
53
54 /*
55 Name: initSize()
56 Parameters: Matrix*, const int r, const int c
57 Return: void
58 Description: Constructor with set size parameters
59 */
60 void initSize(Matrix *m, const int r, const int c)
61 {
62   m->rows = r;
63   m->cols = c;
64
65   for (int i = 0; i < r; i++)
66   {
67     for (int j = 0; j < c; j++)
68     {
69       m->matrix[i][j] = 0;
70     }
71   }
72 }
73
74 /*
75 Name: initValue();
76 Parameters: Matrix*, const int r, const int c, const double
      defval
77 Return: void
78 Description: Constructor with set size parameters and default
      value
79 */
80 void initValue(Matrix *m, const int r, const int c, const double
      defval)
81 {
82   m->rows = r;
83   m->cols = c;
```

```
84
85    for (int i = 0; i < r; i++)
86    {
87      for (int j = 0; j < c; j++)
88      {
89        m->matrix[i][j] = defval;
90      }
91    }
92  }
93
94  /*
95  Name: copyMatrix()
96  Parameters: Matrix *dest, const Matrix *src
97  Return: void
98  Description: Copies src matrix into dest matrix
99  */
100 void copyMatrix(Matrix *dest, const Matrix *src)
101 {
102   if (dest == src)
103     return;
104
105   if (!isValid(dest) || !isValid(src))
106   {
107     error("Cannot copy invalid matrix");
108   }
109
110   dest->rows = src->rows;
111   dest->cols = src->cols;
112
113   for (int i = 0; i < src->rows; i++)
114   {
115     for (int j = 0; j < src->cols; j++)
116     {
117       dest->matrix[i][j] = src->matrix[i][j];
118     }
119   }
120 }
121
122 /*
123 Name: display()
124 Parameters: Matrix
125 Return: void
126 Description: Displays the matrix
127 */
128 void display(const Matrix *m)
```

```
129 {
130   int maxRows = 10, maxCols = 10;
131   int showAllRows = m->rows <= maxRows;
132   int showAllCols = m->cols <= maxCols;
133 
134   for (int i = 0; i < (showAllRows ? m->rows : maxRows); i++)
135   {
136     printf("[");
137     for (int j = 0; j < (showAllCols ? m->cols : maxCols); j++)
138     {
139       printf(" %7.2f", m->matrix[i][j]);
140     }
141     if (!showAllCols)
142       printf("   ...");
143     printf(" ]\n");
144   }
145   if (!showAllRows)
146   {
147     int width = (showAllCols ? m->cols : maxCols) * 8 + (
          showAllCols ? 2 : 6);
148     for (int i = 0; i < width; i++)
149       printf(" ");
150     printf("...\n");
151   }
152 }
153 
154 /*
155 Name: rowScale()
156 Parameters: Matrix*, int row, double scalar
157 Return: void
158 Description: Multiplies a row by a non-zero scalar
159 */
160 void rowScale(Matrix *m, const int row, const double scalar)
161 {
162   if (!isValid(m) || row >= m->rows || row < 0)
163   {
164     error("Invalid matrix size for row scaling");
165   }
166 
167   for (int i = 0; i < m->cols; i++)
168   {
169     m->matrix[row][i] *= scalar;
170   }
171 }
172 
```

```
173 /*
174 Name: rowSwap()
175 Parameters: Matrix*, int row1, int row2
176 Return: void
177 Description: Swaps two valid rows of the matrix
178 */
179 void rowSwap(Matrix *m, const int row1, const int row2)
180 {
181   if (!isValid(m) || row1 >= m->rows || row2 >= m->rows || row1 <
        0 || row2 < 0)
182   {
183     error("Invalid row access for row swap");
184   }
185
186   for (int i = 0; i < m->cols; i++)
187   {
188     double temp = m->matrix[row1][i];
189     m->matrix[row1][i] = m->matrix[row2][i];
190     m->matrix[row2][i] = temp;
191   }
192 }
193
194 /*
195 Name: rowReplace()
196 Parameters: Matrix*, int targetRow, int sourceRow, double scalar
197 Return: void
198 Description: Replaces target row with itself plus scalar
      multiplied by source row
199 */
200 void rowReplace(Matrix *m, const int targetRow, const int
      sourceRow, const double scalar)
201 {
202   if (!isValid(m) || targetRow >= m->rows || sourceRow >= m->rows
        || targetRow < 0 || sourceRow < 0)
203   {
204     error("Invalid row access for row replace");
205   }
206
207   for (int i = 0; i < m->cols; i++)
208   {
209     m->matrix[targetRow][i] += scalar * m->matrix[sourceRow][i];
210   }
211 }
212
213 /*
```

```
214 Name: setValue()
215 Parameters: Matrix*, int r, int c, double value
216 Return: void
217 Description: Sets an element of the row to a value
218 */
219 void setValue(Matrix *m, const int r, const int c, const double
       value)
220 {
221   if (!isValid(m) || r >= m->rows || c >= m->cols || r < 0 || c <
          0)
222   {
223     error("Invalid row or column access for set value");
224   }
225
226   m->matrix[r][c] = value;
227 }
228
229 /*
230 Name: setRandom()
231 Parameters: Matrix*, const int maxRand
232 Return: void
233 Description: Sets matrix to random values
234 */
235 void setRandom(Matrix *m, const int maxRand)
236 {
237   if (!isValid(m))
238   {
239     error("Invalid matrix in set random");
240   }
241
242   for (int i = 0; i < m->rows; i++)
243   {
244     for (int j = 0; j < m->cols; j++)
245     {
246       m->matrix[i][j] = rand() % maxRand;
247     }
248   }
249 }
250
251 /*
252 Name: getValue()
253 Parameters: Matrix*, const int r, const int c
254 Return: double
255 Description: Returns matrix element at a slot
256 */
```

```
257 double getValue(Matrix *m, const int r, const int c)
258 {
259   if (!isValid(m) || r >= m->rows || c >= m->cols || r < 0 || c <
        0)
260   {
261     error("Invalid row or column access in get value");
262   }
263
264   return m->matrix[r][c];
265 }
266
267 /*
268 Name: displayDimensions()
269 Parameters: Matrix*
270 Return: void
271 Description: Displays dimensions of the matrix
272 */
273 void displayDimensions(const Matrix *m)
274 {
275   if (!isValid(m))
276   {
277     error("Invalid matrix");
278   }
279
280   printf("[%d rows, %d cols]\n", m->rows, m->cols);
281 }
282
283 /*
284 Name: isSquare()
285 Parameters: const Matrix* A
286 Return: bool
287 Description: Returns true if the matrix is square
288 */
289 bool isSquare(const Matrix *A)
290 {
291   return A->rows == A->cols;
292 }
293
294 /*
295 Name: transpose()
296 Parameters: const Matrix *input, Matrix *output
297 Return: void
298 Description: Transposes the input matrix, stores result in output
       matrix
299 */
```

```
300 void transpose(const Matrix *input, Matrix *output)
301 {
302
303   if (!isValid(input))
304   {
305     error("Invalid matrix in transpose");
306   }
307
308   int rank, size;
309   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
310   MPI_Comm_size(MPI_COMM_WORLD, &size);
311   MPI_Bcast((void *)input, sizeof(Matrix), MPI_BYTE, 0,
        MPI_COMM_WORLD);
312
313   output->rows = input->cols;
314   output->cols = input->rows;
315
316   int rows_per_proc = input->rows / size;
317   int remaining_rows = input->rows % size;
318   int start = rank * rows_per_proc + (rank < remaining_rows ?
        rank : remaining_rows);
319   int end = start + rows_per_proc + (rank < remaining_rows ? 1 :
        0);
320
321   for (int i = start; i < end; i++)
322   {
323     for (int j = 0; j < input->cols; j++)
324     {
325       output->matrix[j][i] = input->matrix[i][j];
326     }
327   }
328
329   // back to all processes
330   for (int proc = 0; proc < size; proc++)
331   {
332     int proc_start = proc * rows_per_proc + (proc <
          remaining_rows ? proc : remaining_rows);
333     int proc_end = proc_start + rows_per_proc + (proc <
          remaining_rows ? 1 : 0);
334
335     for (int row = 0; row < input->cols; row++)
336     {
337       for (int col = proc_start; col < proc_end; col++)
338       {
339         MPI_Bcast(&output->matrix[row][col], 1, MPI_DOUBLE, proc,
```

```
                MPI_COMM_WORLD);
340         }
341       }
342     }
343 }
344
345 /*
346 Name: writeToFile()
347 Parameters: const Matrix*, const char* filename
348 Return: void
349 Description: Writes a matrix to a file
350 */
351 void writeToFile(const Matrix *m, const char *filename)
352 {
353   if (!isValid(m))
354   {
355     error("Invalid matrix");
356   }
357
358   FILE *file = fopen(filename, "w");
359   if (file == NULL)
360   {
361     error("Unable to open file for writing");
362   }
363
364   fprintf(file, "Rows: %d, Cols: %d\n", m->rows, m->cols);
365   for (int i = 0; i < 25; i++)
366     fprintf(file, "-");
367   fprintf(file, "\n");
368
369   for (int i = 0; i < m->rows; i++)
370   {
371     for (int j = 0; j < m->cols; j++)
372     {
373       fprintf(file, "%lf ", m->matrix[i][j]);
374     }
375     fprintf(file, "\n");
376   }
377
378   fclose(file);
379 }
380
381 /*
382 Name: solve()
383 Parameters: const Matrix*, const double[], const int size
```

```
384 Return: bool
385 Description: Returns true if the solution set works with the
       matrix
386 */
387 bool solve(const Matrix *m, const double solutions[], const int
       size)
388 {
389   if (!isValid(m) || size < 1)
390   {
391     error("Cannot check solution set on empty matrix or empty
          solution set");
392   }
393
394   const double EPSILON = 1e-5;
395   double value = 0;
396
397   for (int i = 0; i < m->rows; i++)
398   {
399     value = 0;
400     for (int j = 0; j < m->cols - 1; j++)
401     {
402       value += m->matrix[i][j] * solutions[j];
403     }
404     if (fabs(value - m->matrix[i][m->cols - 1] > EPSILON))
405       return false;
406   }
407   return true;
408 }
409
410 /*
411 Name: multiplyMatrix()
412 Parameters: const Matrix *A, const Matrix *B, Matrix *C
413 Return: void
414 Description: Multiplies matrix A and B using OpenMPI and stores
       result in matrix C.
415 */
416 void multiplyMatrix(const Matrix *A, const Matrix *B, Matrix *C)
417 {
418   int rank, size;
419   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
420   MPI_Comm_size(MPI_COMM_WORLD, &size);
421
422   int a_rows = A->rows;
423   int a_cols = A->cols;
424   int b_cols = B->cols;
```

```
425
426   if (rank == 0)
427   {
428     if (!isValid(A) || !isValid(B))
429       error("Invalid input matrices");
430     if (a_cols != B->rows)
431       error("Incompatible dimensions for multiplication");
432   }
433
434   // Broadcast matrix dimensions
435   MPI_Bcast(&a_rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
436   MPI_Bcast(&a_cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
437   MPI_Bcast(&b_cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
438
439   // Broadcast B (whole matrix)
440   MPI_Bcast((void *)B->matrix, MAX_ROWS * MAX_COLS, MPI_DOUBLE,
        0, MPI_COMM_WORLD);
441
442   // Calculate local rows
443   int rows_per_proc = a_rows / size;
444   int remainder = a_rows % size;
445   int local_rows = (rank < remainder) ? rows_per_proc + 1 :
        rows_per_proc;
446   printf("Rank %d received %d rows, computing...\n", rank,
        local_rows);
447
448   // Allocate only as much as needed
449   double local_A[local_rows][a_cols];
450   double local_C[local_rows][b_cols];
451   memset(local_C, 0, sizeof(local_C));
452
453   // Prepare scatter metadata
454   int sendcounts[size];
455   int displs[size];
456   int offset = 0;
457   for (int i = 0; i < size; i++)
458   {
459     int count = ((i < remainder) ? rows_per_proc + 1 :
          rows_per_proc) * a_cols;
460     sendcounts[i] = count;
461     displs[i] = offset;
462     offset += count;
463   }
464
465   // Scatter A rows to processes
```

```
466   MPI_Scatterv(&(A->matrix[0][0]), sendcounts, displs, MPI_DOUBLE
          ,
467                &(local_A[0][0]), local_rows * a_cols, MPI_DOUBLE,
468                0, MPI_COMM_WORLD);
469
470   // Compute local result
471   for (int i = 0; i < local_rows; i++)
472   {
473     for (int j = 0; j < b_cols; j++)
474     {
475       for (int k = 0; k < a_cols; k++)
476       {
477         local_C[i][j] += local_A[i][k] * B->matrix[k][j];
478       }
479     }
480   }
481
482   // Prepare gather metadata
483   int recvcounts[size];
484   int recvdispls[size];
485   offset = 0;
486   for (int i = 0; i < size; i++)
487   {
488     int count = ((i < remainder) ? rows_per_proc + 1 :
          rows_per_proc) * b_cols;
489     recvcounts[i] = count;
490     recvdispls[i] = offset;
491     offset += count;
492   }
493
494   // Gather results into final matrix C
495   MPI_Gatherv(&(local_C[0][0]), local_rows * b_cols, MPI_DOUBLE,
496               &(C->matrix[0][0]), recvcounts, recvdispls,
                    MPI_DOUBLE,
497               0, MPI_COMM_WORLD);
498
499   // Set dimensions on root
500   if (rank == 0)
501   {
502     C->rows = a_rows;
503     C->cols = b_cols;
504   }
505 }
506
507 /*
```

```
508 Name: ref()
509 Parameters: Matrix *m
510 Return: void
511 Description: Computes the Row Echelon Form (REF) of matrix m
       using OpenMPI.
512 */
513 void ref(Matrix *m)
514 {
515   if (!isValid(m))
516     error("Invalid matrix");
517
518   int rank, size;
519   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
520   MPI_Comm_size(MPI_COMM_WORLD, &size);
521
522   int rows = m->rows;
523   int cols = m->cols;
524
525   int rows_per_proc = rows / size;
526   int remainder = rows % size;
527   int local_rows = (rank < remainder) ? rows_per_proc + 1 :
        rows_per_proc;
528   printf("Rank %d preparing to allocate %d rows    %d cols = %.2f
        MB\n", rank, local_rows, cols, (local_rows * cols * sizeof(
       double)) / (1024.0 * 1024.0));
529
530   int start_row = (rank < remainder)
531                       ? rank * (rows_per_proc + 1)
532                       : rank * rows_per_proc + remainder;
533
534   double local_matrix[local_rows][cols];
535   printf("Rank %d received %d rows, computing...\n", rank,
       local_rows);
536
537   // Scatter rows of m into local_matrix
538   int sendcounts[size], displs[size];
539   int offset = 0;
540   for (int i = 0; i < size; i++)
541   {
542     int count = ((i < remainder) ? rows_per_proc + 1 :
          rows_per_proc) * cols;
543     sendcounts[i] = count;
544     displs[i] = offset;
545     offset += count;
546   }
```

```
547
548   MPI_Scatterv(&(m->matrix[0][0]), sendcounts, displs, MPI_DOUBLE
          ,
549                &(local_matrix[0][0]), local_rows * cols,
                     MPI_DOUBLE,
550                0, MPI_COMM_WORLD);
551
552   // Temporary pivot row buffer
553   double pivot_row[cols];
554
555   for (int r = 0; r < rows; r++)
556   {
557     int owner = -1, local_r = -1;
558     for (int i = 0; i < size; i++)
559     {
560       int start = (i < remainder) ? i * (rows_per_proc + 1) : i *
              rows_per_proc + remainder;
561       int end = start + ((i < remainder) ? rows_per_proc + 1 :
             rows_per_proc);
562       if (r >= start && r < end)
563       {
564         owner = i;
565         local_r = r - start;
566         break;
567       }
568     }
569
570     // Broadcast pivot row
571     if (rank == owner)
572       memcpy(pivot_row, local_matrix[local_r], sizeof(double) *
             cols);
573
574     MPI_Bcast(pivot_row, cols, MPI_DOUBLE, owner, MPI_COMM_WORLD)
          ;
575
576     // Eliminate below pivot
577     for (int i = 0; i < local_rows; i++)
578     {
579       int global_row = start_row + i;
580       if (global_row <= r)
581         continue;
582
583       double factor = local_matrix[i][r] / pivot_row[r];
584       for (int j = r; j < cols; j++)
585       {
```

```
586           local_matrix[i][j] -= factor * pivot_row[j];
587         }
588       }
589     }
590
591     // Gather results back to m
592     MPI_Gatherv(&(local_matrix[0][0]), local_rows * cols,
          MPI_DOUBLE,
593                 &(m->matrix[0][0]), sendcounts, displs, MPI_DOUBLE,
594                 0, MPI_COMM_WORLD);
595 }
596
597 /*
598 Name: rref()
599 Parameters: Matrix *m
600 Return: void
601 Description: Computes Reduced Row Echelon Form (RREF) using
       OpenMPI.
602 */
603 void rref(Matrix *m)
604 {
605     if (!isValid(m))
606       error("Invalid matrix");
607
608     int rank, size;
609     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
610     MPI_Comm_size(MPI_COMM_WORLD, &size);
611
612     int rows = m->rows;
613     int cols = m->cols;
614
615     int rows_per_proc = rows / size;
616     int remainder = rows % size;
617     int local_rows = (rank < remainder) ? rows_per_proc + 1 :
          rows_per_proc;
618
619     int start_row = (rank < remainder)
620                         ? rank * (rows_per_proc + 1)
621                         : rank * rows_per_proc + remainder;
622
623     double local_matrix[local_rows][cols];
624
625     printf("Rank %d received %d rows, computing...\n", rank,
          local_rows);
626
```

```
627   // Setup for scatter/gather
628   int sendcounts[size], displs[size];
629   int offset = 0;
630   for (int i = 0; i < size; i++)
631   {
632     int count = ((i < remainder) ? rows_per_proc + 1 :
          rows_per_proc) * cols;
633     sendcounts[i] = count;
634     displs[i] = offset;
635     offset += count;
636   }
637
638   // Scatter matrix
639   MPI_Scatterv(&(m->matrix[0][0]), sendcounts, displs, MPI_DOUBLE
       ,
640                &(local_matrix[0][0]), local_rows * cols,
                     MPI_DOUBLE,
641                0, MPI_COMM_WORLD);
642
643   double pivot_row[cols];
644
645   for (int r = 0; r < rows; r++)
646   {
647     int owner = -1, local_r = -1;
648     for (int i = 0; i < size; i++)
649     {
650       int s = (i < remainder) ? i * (rows_per_proc + 1) : i *
            rows_per_proc + remainder;
651       int e = s + ((i < remainder) ? rows_per_proc + 1 :
            rows_per_proc);
652       if (r >= s && r < e)
653       {
654         owner = i;
655         local_r = r - s;
656         break;
657       }
658     }
659
660     // Normalize pivot row on owner
661     if (rank == owner)
662     {
663       double pivot = local_matrix[local_r][r];
664       if (pivot != 0)
665       {
666         for (int j = r; j < cols; j++)
```

```
667             local_matrix[local_r][j] /= pivot;
668         }
669         memcpy(pivot_row, local_matrix[local_r], sizeof(double) *
               cols);
670       }
671
672       // Broadcast normalized pivot row
673       MPI_Bcast(pivot_row, cols, MPI_DOUBLE, owner, MPI_COMM_WORLD)
             ;
674
675       // Eliminate all other rows (above and below)
676       for (int i = 0; i < local_rows; i++)
677       {
678         int global_i = start_row + i;
679         if (global_i == r)
680           continue;
681
682         double factor = local_matrix[i][r];
683         for (int j = r; j < cols; j++)
684           local_matrix[i][j] -= factor * pivot_row[j];
685       }
686     }
687
688     // Gather result to root
689     MPI_Gatherv(&(local_matrix[0][0]), local_rows * cols,
           MPI_DOUBLE,
690                 &(m->matrix[0][0]), sendcounts, displs, MPI_DOUBLE,
691                 0, MPI_COMM_WORLD);
692 }
693
694 /*
695 Name: addMatrix()
696 Parameters: const Matrix *A, const Matrix *B, Matrix *result
697 Return: Void
698 Description: Stores result of addition of A and B
699 */
700 void addMatrix(const Matrix *A, const Matrix *B, Matrix *result)
701 {
702   if (!isValid(A) || !isValid(B) || A->cols != B->cols || A->rows
          != B->rows)
703   {
704     error("Invalid matrix size");
705   }
706
707   int rank, size;
```

```
708   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
709   MPI_Comm_size(MPI_COMM_WORLD, &size);
710
711   int rows_per_proc = A->rows / size;
712   int remaining_rows = A->rows % size;
713   int start = rank * rows_per_proc + (rank < remaining_rows ?
         rank : remaining_rows);
714   int end = start + rows_per_proc + (rank < remaining_rows ? 1 :
         0);
715
716   for (int i = start; i < end; i++)
717   {
718     for (int j = 0; j < A->cols; j++)
719     {
720       result->matrix[i][j] = A->matrix[i][j] + B->matrix[i][j];
721     }
722   }
723
724   MPI_Barrier(MPI_COMM_WORLD);
725   MPI_Allreduce(MPI_IN_PLACE, result->matrix, MAX_ROWS * MAX_COLS
         , MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
726 }
727
728 /*
729 Name: subtractMatrix()
730 Parameters: const Matrix *A, const Matrix *B, Matrix *result
731 Return: Void
732 Description: Stores result of subtraction of A and B
733 */
734 void subtractMatrix(const Matrix *A, const Matrix *B, Matrix *
      result)
735 {
736   if (!isValid(A) || !isValid(B) || A->cols != B->cols || A->rows
        != B->rows)
737   {
738     error("Invalid matrix size");
739   }
740
741   if (!isValid(A) || !isValid(B) || A->cols != B->cols || A->rows
        != B->rows)
742   {
743     error("Invalid matrix size");
744   }
745
746   int rank, size;
```

```
747   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
748   MPI_Comm_size(MPI_COMM_WORLD, &size);
749
750   int rows_per_proc = A->rows / size;
751   int remaining_rows = A->rows % size;
752   int start = rank * rows_per_proc + (rank < remaining_rows ?
         rank : remaining_rows);
753   int end = start + rows_per_proc + (rank < remaining_rows ? 1 :
         0);
754
755   for (int i = start; i < end; i++)
756   {
757     for (int j = 0; j < A->cols; j++)
758     {
759       result->matrix[i][j] = A->matrix[i][j] - B->matrix[i][j];
760     }
761   }
762
763   MPI_Barrier(MPI_COMM_WORLD);
764   MPI_Allreduce(MPI_IN_PLACE, result->matrix, MAX_ROWS * MAX_COLS
       , MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
765 }
766
767 /*
768 Name: LU
769 Parameters: Matrix* A, Matrix* L, Matrix* U
770 Return: void
771 Description: Performs LU decomposition
772 */
773 void LU(Matrix *A, Matrix *L, Matrix *U)
774 {
775   int n = A->rows;
776
777   // Initialize L and U matrices
778   for (int i = 0; i < n; ++i)
779   {
780     for (int j = 0; j < n; ++j)
781     {
782       L->matrix[i][j] = 0.0;
783       U->matrix[i][j] = A->matrix[i][j];
784     }
785   }
786
787   for (int k = 0; k < n; ++k)
788   {
```

```
789       for (int i = k + 1; i < n; ++i)
790       {
791         double multiplier = U->matrix[i][k] / U->matrix[k][k];
792         L->matrix[i][k] = multiplier;
793
794         for (int j = k; j < n; ++j)
795         {
796           U->matrix[i][j] -= multiplier * U->matrix[k][j];
797         }
798       }
799     }
800
801     // Set the diagonal of L to 1
802     for (int i = 0; i < n; ++i)
803     {
804       L->matrix[i][i] = 1.0;
805     }
806   }
807
808   /*
809
810
811
812
813
814   Sequentual functions below
815
816
817
818
819
820   */
821
822   /*
823   Name: Seq_multiplyMatrix()
824   Parameters: const Matrix *A, const Matrix *B, Matrix *C
825   Return: void
826   Description: Multiplies matrix A and matrix B sequentially and
          stores result in matrix C.
827   */
828   void Seq_multiplyMatrix(const Matrix *A, const Matrix *B, Matrix
          *C)
829   {
830     if (!isValid(A) || !isValid(B))
831       error("Invalid input matrices");
```

```
832   if (A->cols != B->rows)
833     error("Incompatible dimensions for multiplication");
834 
835   for (int i = 0; i < A->rows; i++)
836   {
837     for (int j = 0; j < B->cols; j++)
838     {
839       C->matrix[i][j] = 0;
840       for (int k = 0; k < A->cols; k++)
841       {
842         C->matrix[i][j] += A->matrix[i][k] * B->matrix[k][j];
843       }
844     }
845   }
846 }
847 
848 /*
849 Name: Seq_ref()
850 Parameters: Matrix *m
851 Return: void
852 Description: Puts the matrix into row echelon form sequentially.
853 */
854 void Seq_ref(Matrix *m)
855 {
856   if (!isValid(m))
857     error("Invalid matrix");
858 
859   int lead = 0;
860   int rowCount = m->rows;
861   int colCount = m->cols;
862 
863   for (int r = 0; r < rowCount; r++)
864   {
865     if (lead >= colCount)
866       return;
867 
868     int i = r;
869     while (m->matrix[i][lead] == 0)
870     {
871       i++;
872       if (i == rowCount)
873       {
874         i = r;
875         lead++;
876         if (lead == colCount)
```

```
877             return;
878         }
879       }
880
881       if (i != r)
882       {
883         for (int j = 0; j < colCount; j++)
884         {
885           double temp = m->matrix[r][j];
886           m->matrix[r][j] = m->matrix[i][j];
887           m->matrix[i][j] = temp;
888         }
889       }
890
891       double lv = m->matrix[r][lead];
892       if (lv != 0)
893       {
894         for (int j = 0; j < colCount; j++)
895           m->matrix[r][j] /= lv;
896       }
897
898       for (int i = r + 1; i < rowCount; i++)
899       {
900         double lv2 = m->matrix[i][lead];
901         for (int j = 0; j < colCount; j++)
902           m->matrix[i][j] -= lv2 * m->matrix[r][j];
903       }
904
905       lead++;
906     }
907 }
908
909 /*
910 Name: Seq_rref()
911 Parameters: Matrix *m
912 Return: void
913 Description: Converts matrix to reduced row echelon form without
       MPI.
914 */
915 void Seq_rref(Matrix *m)
916 {
917   if (!isValid(m))
918     error("Invalid matrix");
919
920   Seq_ref(m);
```

```
921
922   int rowCount = m->rows;
923   int colCount = m->cols;
924
925   for (int r = rowCount - 1; r >= 0; r--)
926   {
927     int leadCol = -1;
928     for (int j = 0; j < colCount; j++)
929     {
930       if (fabs(m->matrix[r][j] - 1.0) < 1e-6)
931       {
932         leadCol = j;
933         break;
934       }
935     }
936
937     if (leadCol == -1)
938       continue;
939
940     for (int i = 0; i < r; i++)
941     {
942       double factor = m->matrix[i][leadCol];
943       for (int j = 0; j < colCount; j++)
944         m->matrix[i][j] -= factor * m->matrix[r][j];
945     }
946   }
947 }
948
949 /*
950 Name: Seq_LU()
951 Parameters: const Matrix *A, Matrix *L, Matrix *U
952 Return: void
953 Description: Performs LU decomposition sequentially.
954 */
955 void Seq_LU(const Matrix *A, Matrix *L, Matrix *U)
956 {
957   if (!isValid(A))
958     error("Invalid matrix");
959
960   int n = A->rows;
961
962   for (int i = 0; i < n; ++i)
963   {
964     for (int j = 0; j < n; ++j)
965     {
```

```
966         L->matrix[i][j] = 0;
967         U->matrix[i][j] = 0;
968       }
969     }
970
971     for (int k = 0; k < n; ++k)
972     {
973       for (int j = k; j < n; ++j)
974       {
975         double sum = 0;
976         for (int s = 0; s < k; ++s)
977           sum += L->matrix[k][s] * U->matrix[s][j];
978         U->matrix[k][j] = A->matrix[k][j] - sum;
979       }
980
981       L->matrix[k][k] = 1.0;
982
983       for (int i = k + 1; i < n; ++i)
984       {
985         double sum = 0;
986         for (int s = 0; s < k; ++s)
987           sum += L->matrix[i][s] * U->matrix[s][k];
988         L->matrix[i][k] = (A->matrix[i][k] - sum) / U->matrix[k][k
              ];
989       }
990     }
991 }
```

multiplyTest.c

=1

1

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
#include "MatrixC.h"

void divisor(const char *msg)
{
  printf("\n---- %s ----\n\n", msg);
}

Matrix A, B, C_parallel, C_seq;

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  srand(time(NULL) + rank);

  int N = 100, use_parallel = 0;

  if (rank == 0)
  {
    printf("Enter matrix size: ");
    scanf("%d", &N);

    printf("Run sequential too? (1 = yes, 0 = no): ");
    scanf("%d", &use_parallel);
  }

  // Broadcast settings
  MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&use_parallel, 1, MPI_INT, 0, MPI_COMM_WORLD);

  initSize(&A, N, N);
  initSize(&B, N, N);
  initSize(&C_parallel, N, N);
```

```
42  initSize(&C_seq, N, N);
43
44  // Rank 0 generates the random data
45  if (rank == 0)
46  {
47    setRandom(&A, 100);
48    setRandom(&B, 100);
49  }
50
51  // Broadcast A and B
52  MPI_Bcast(&(A.matrix[0][0]), N * N, MPI_DOUBLE, 0,
       MPI_COMM_WORLD);
53  MPI_Bcast(&(B.matrix[0][0]), N * N, MPI_DOUBLE, 0,
       MPI_COMM_WORLD);
54
55  // ------------------ Parallel ------------------
56  if (rank == 0)
57    printf("Running parallel multiplyMatrix()...\n");
58
59  double start = MPI_Wtime();
60  multiplyMatrix(&A, &B, &C_parallel);
61  double end = MPI_Wtime();
62
63  if (rank == 0)
64  {
65    printf("multiplyMatrix() done in %.3f seconds\n", end - start
         );
66    divisor("multiplyMatrix");
67  }
68
69  // ------------------ Sequential ------------------
70  if (rank == 0 && use_parallel == 1)
71  {
72    printf("Running sequential Seq_multiplyMatrix()...\n");
73
74    start = MPI_Wtime();
75    Seq_multiplyMatrix(&A, &B, &C_seq);
76    end = MPI_Wtime();
77
78    printf("Seq_multiplyMatrix() done in %.3f seconds\n", end -
         start);
79    divisor("Seq_multiplyMatrix");
80  }
81
82  MPI_Finalize();
```

```
83    return 0;
84 }
```

refTest.c

=1

1

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
#include "MatrixC.h"

void divisor(const char *msg)
{
  printf("\n---- %s ----\n\n", msg);
}

// Global matrices
Matrix A_parallel, A_seq;

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  srand(time(NULL) + rank);

  int N = 10, use_parallel = 1;

  if (rank == 0)
  {
    printf("Enter matrix size: ");
    scanf("%d", &N);

    printf("Run sequential too? (1 = yes, 0 = no): ");
    scanf("%d", &use_parallel);
  }

  // Broadcast settings
  MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&use_parallel, 1, MPI_INT, 0, MPI_COMM_WORLD);

  initSize(&A_parallel, N, N);
  initSize(&A_seq, N, N);
```

```
42
43    // Rank 0 generates the random matrix
44    if (rank == 0)
45    {
46      setRandom(&A_parallel, 100);
47      copyMatrix(&A_seq, &A_parallel);
48    }
49
50    // Broadcast A_parallel to all ranks
51    MPI_Bcast(&(A_parallel.matrix[0][0]), N * N, MPI_DOUBLE, 0,
         MPI_COMM_WORLD);
52
53    // ----------- Parallel ref() -----------
54    if (rank == 0)
55      printf("Running parallel ref()...\n");
56
57    double start = MPI_Wtime();
58    ref(&A_parallel);
59    double end = MPI_Wtime();
60
61    if (rank == 0)
62    {
63      printf("ref() done in %.3f seconds\n", end - start);
64      divisor("ref");
65    }
66
67    // ----------- Sequential Seq_ref() -----------
68    if (rank == 0 && use_parallel == 1)
69    {
70      printf("Running sequential Seq_ref()...\n");
71
72      start = MPI_Wtime();
73      Seq_ref(&A_seq);
74      end = MPI_Wtime();
75
76      printf("Seq_ref() done in %.3f seconds\n", end - start);
77      divisor("Seq_ref");
78    }
79
80    MPI_Finalize();
81    return 0;
82  }
```

rrefTest.c

=1

1

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
#include "MatrixC.h"

void divisor(const char *msg)
{
  printf("\n---- %s ----\n\n", msg);
}

// Global matrices
Matrix A_parallel, A_seq;

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  srand(time(NULL) + rank);

  int N = 10, use_parallel = 1;

  if (rank == 0)
  {
    printf("Enter matrix size: ");
    scanf("%d", &N);

    printf("Run sequential too? (1 = yes, 0 = no): ");
    scanf("%d", &use_parallel);
  }

  // Broadcast settings
  MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&use_parallel, 1, MPI_INT, 0, MPI_COMM_WORLD);

  initSize(&A_parallel, N, N);
  initSize(&A_seq, N, N);
```

```
42
43    // Rank 0 sets data
44    if (rank == 0)
45    {
46      setRandom(&A_parallel, 100);
47      copyMatrix(&A_seq, &A_parallel);
48    }
49
50    // Broadcast to all processes
51    MPI_Bcast(&(A_parallel.matrix[0][0]), N * N, MPI_DOUBLE, 0,
          MPI_COMM_WORLD);
52
53    // ----------- Parallel rref() -----------
54    if (rank == 0)
55      printf("Running parallel rref()...\n");
56
57    double start = MPI_Wtime();
58    rref(&A_parallel);
59    double end = MPI_Wtime();
60
61    if (rank == 0)
62    {
63      printf("rref() done in %.3f seconds\n", end - start);
64      divisor("rref");
65    }
66
67    // ----------- Sequential Seq_rref() -----------
68    if (rank == 0 && use_parallel == 1)
69    {
70      printf("Running sequential Seq_rref()...\n");
71
72      start = MPI_Wtime();
73      Seq_rref(&A_seq);
74      end = MPI_Wtime();
75
76      printf("Seq_rref() done in %.3f seconds\n", end - start);
77      divisor("Seq_rref");
78    }
79
80    MPI_Finalize();
81    return 0;
82  }
```

luTest.c

=1

1

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
#include "MatrixC.h"

void divisor(const char *msg);

Matrix A_original, L_parallel, U_parallel;
Matrix L_seq, U_seq;

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  srand(time(NULL) + rank);

  int N = 10, use_sequential = 0;

  if (rank == 0)
  {
    printf("Enter matrix size (N for NxN): ");
    scanf("%d", &N);

    printf("Run sequential also? (1 = yes, 0 = no): ");
    scanf("%d", &use_sequential);
  }

  MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&use_sequential, 1, MPI_INT, 0, MPI_COMM_WORLD);

  initSize(&A_original, N, N);
  initSize(&L_parallel, N, N);
  initSize(&U_parallel, N, N);

  if (rank == 0)
  {
```

```
42      initSize(&L_seq, N, N);
43      initSize(&U_seq, N, N);
44      setRandom(&A_original, 100);
45    }
46
47    MPI_Bcast(&(A_original.matrix[0][0]), N * N, MPI_DOUBLE, 0,
          MPI_COMM_WORLD);
48
49    // ------------------ Parallel LU -----------------
50    if (rank == 0)
51      printf("Running parallel LU()...\n");
52
53    double start = MPI_Wtime();
54    LU(&A_original, &L_parallel, &U_parallel);
55    double end = MPI_Wtime();
56
57    if (rank == 0)
58    {
59      printf("Parallel LU() done in %.3f seconds\n", end - start);
60      divisor("Parallel LU");
61
62      if (use_sequential == 1)
63      {
64        Matrix A_copy;
65        initSize(&A_copy, N, N);
66        copyMatrix(&A_copy, &A_original);
67
68        printf("Running sequential Seq_LU()...\n");
69
70        start = MPI_Wtime();
71        Seq_LU(&A_copy, &L_seq, &U_seq);
72        end = MPI_Wtime();
73
74        printf("Sequential Seq_LU() done in %.3f seconds\n", end -
              start);
75        divisor("Sequential LU");
76      }
77    }
78
79    MPI_Finalize();
80    return 0;
81  }
82
83  void divisor(const char *msg)
84  {
```

```
85   printf("\n---- %s ----\n\n", msg);
86 }
```

runTest.sh

=1

1

```
#!/bin/bash

echo "Enter matrix size (N for NxN): "
read N

echo "Run sequential also? (1 = yes, 0 = no): "
read USE_SEQ

echo "Enter number of MPI processes to use: "
read PROCS

echo "Which test to run?"
echo "1. multiplyTest"
echo "2. refTest"
echo "3. rrefTest"
echo "4. luTest"
read CHOICE

# Pick executable
case $CHOICE in
  1) EXEC="./multiplyTest" ;;
  2) EXEC="./refTest" ;;
  3) EXEC="./rrefTest" ;;
  4) EXEC="./luTest" ;;
  *) echo "Invalid choice"; exit 1 ;;
esac

echo
echo "Running $EXEC with:"
echo "- Matrix size: $N"
echo "- Sequential: $USE_SEQ"
echo "- Processes: $PROCS"
echo

mpirun --oversubscribe -np $PROCS $EXEC <<< "$N
$USE_SEQ"
```