MECHTRON 2MP3: Programming for Mechatronics

Solving System of Linear Equations in C

Ethan Otteson, 400433686

1 Introduction

The following documentation is for a C program created for McMaster University MECHTRON 2MP3 Assignment 3 which solves the linear algebra equation Ax = b for x. This is done by reading a file in Matrix Market (MM) format and storing it in Compressed Sparse Row (CSR) format. CSR format uses three one-dimensional arrays to store sparse matrices in order to save memory. After the matrix has been stored the equation Ax = b, where A is the discussed matrix, b is a known column vector, and x is the unknown column vector, can be solved using the Gauss-Seidel Method. Once x has been calculated its accuracy is checked by computing the residual and the norm of this residual vector, which is the metric used to determine success. Below, each one of these steps is discussed in further detail, the results of several test cases are examined, and the code is critically analyzed. Additionally, the `Makefile` used is shown and discussed at the end.

2 Matrix Market Format to CSR Format

To begin each program the user will specify which Matrix Market format file is to be used after the program call in the terminal `./main filename.mtx`. The function `ReadMMtoCSR` will take the filename as an input and open the file in read mode. First, it will skip any comments at the top of the file before reading the first line with numeric data. This first line is important as it contains the total number of rows, columns, and non-zero entries. Once these parameters are known, memory can be dynamically allocated to the exact size required for each of the one-dimensional arrays. The rest of the file is then read as the row, column, and value of each entry are stored in temporary arrays. These temporary arrays are used to properly order each entry before storing them in the A matrix structure. Once all the file data has been read the file is closed and the temporary arrays are sorted based on ascending row data, this is done using the `qsort()` part of the C standard library. Sorting each array by row is required so that the row pointer can properly point to the start of each row within the matrix.

After the matrix data has been correctly stored in CSR format, the program will check if the matrix is either upper or lower triangular with the `check_symmetry` function. This is done as it is standard practice for MM format to save symmetric matrices as triangular matrices in order to save storage without losing data. As all the matrices given were symmetric matrices it was always

assumed that a triangular matrix represents its symmetric cousin.

If a matrix was determined to be triangular and therefore symmetric the empty triangle of the matrix was filled with the appropriate data by the `fill_symmetry` function. This was done by understanding that `A[i][j]` was also equal to `A[j][i]`. The new required size of each array was determined by doubling the number of non-zero entries then subtracting the number of entries on the diagonal before reallocating the memory based on this new number of total non-zero entries. After this once again temporary arrays were used to store and sort the data with `qsort()` before the data was permanently stored in the A matrix structure.

While it is not strictly necessary to fill the other half of the matrix and is possible to continue working with just the triangular part of the matrix, filling the other half has two key benefits. It makes it simpler to access entire rows from the matrix reducing the chance of logical mistakes within the code. Additionally, it allows the sparse matrix multiplication and solver functions to work with both symmetric and non-symmetric matrices without having to include conditional statements which perform the functions differently dependent on symmetry. However, it is important to point out that this technique of copying the data to both triangles of the matrix requires significantly more memory, although this was not an issue I came across it is still important to mention.

Below is an example of what the row pointer, column index, and data index arrays look like after reading the `LFAT5.mtx` file. It is important to mention that these results are after having duplicated the matrix across the diagonal. This would result in the row pointer, column index, and data index arrays appearing differently than if they were displayed before the duplicating process. This is the reason for `Number of File Read Non-Zeros` and `Number of Total Non-Zeros` being different for symmetric matrices. The former is the number of non-zeros shown in the file, while the latter is the actual number of non-zeros within the whole array. If a non-symmetric matrix such as `b1_ss.mtx` is passed into the program instead it would be observed that both values are the same as there is no duplicating of values required for a non-symmetric matrix.

```
Finished reading LFAT5.mtx file.
This matrix is lower triangular assuming this means it is symmetric
   , program will continue.

Matrix data from LFAT5.mtx file. Note that if this matrix was given
    as triangular it was assumed symmetric and the relevant data
   was copied to the other triangle.

Matrix Dimensions: 14 by 14
Number of File Read Non-Zeros: 30
Number of Total Non-Zeros: 46
Row Pointer: 0 3 5 7 11 15 18 21 26 31 33 35 39 43 46
```

```
Column Index: 0 3 4 1 5 2 6 0 3 7 8 0 4 7 8 1 5 9 2 6 10 3 4 7 11
   12 3 4 8 11 12 5 9 6 10 7 8 11 13 7 8 12 13 11 12 13
Value: 1.5709 -94.2528 0.7854 12566400.0000 -6283200.0000 0.6088
   -0.3044 -94.2528 15080.4480 -7540.2240 94.2528 0.7854 3.1418
   -94.2528 0.7854 -6283200.0000 12566400.0000 -6283200.0000
   -0.3044 0.6088 -0.3044 -7540.2240 -94.2528 15080.4480 -7540.2240
    94.2528 94.2528 0.7854 3.1418 -94.2528 0.7854 -6283200.0000
   12566400.0000 -0.3044 0.6088 -7540.2240 -94.2528 15080.4480
   94.2528 94.2528 0.7854 3.1418 0.7854 94.2528 0.7854 1.5709

Gauss-Seidel solver terminated by reaching the stop criteria of 1.0
   e-16 in 1101 iterations.
CPU time taken to solve Ax=b: 0.000231 seconds

Residual Norm: 1.6298e-14
```

3 Sparse Matrix Multiplication

The sparse matrix multiplication is actually quite simple especially because of the duplicating across the diagonal for symmetric matrices which was previously discussed. All that needs to be done is to go through each row one by one and multiply the values by the proper values in the column vector.

The assignment instructions ask the question, we know that CSR uses less memory but why is it also faster when doing an operation like this? The reason is because we only do the multiplication for the non-zero values. For example, in `LFAT5.mtx` there are three non-zero values in the first row so with a sparse matrix only three multiplications are done to calculate the first row of the output. On the other hand, a dense matrix would do this multiplication 14 times as there are 14 entries in each row, even though 11 of these multiplications are by zero and do not yield anything useful. This drastically decreased number of total multiplications is why sparse matrix multiplication for a given matrix is faster than the same multiplication done by representing the matrix in dense format.

4 Solving $Ax = b$

In order to solve the system of linear equations the Gauss-Seidel method is used. This method is almost identical to the Jacobi method but with one improvement. The Gauss-Seidel method always

uses the most up-to-date values for the variables rather than the Jacobi method which uses the same set of variables to solve for the whole set of variables each iteration. This small improvement allows the Gauss-Seidel method to converge to a correct answer much faster. Additionally, this iterative method allows the user to decide how precise they would like their answer as they can choose the maximum number of iterations and the stop criteria. In the current implementation, these values are hard coded in the main as values which gave sufficient results for each file I tested. They were hard coded as it was not part of the assignment to give the user the choice for these parameters but since they are defined in the main before being passed into the function it would be very easy to adapt the code to allow for user input to set these variables. There are some downsides with this method though, which go for both the Gauss-Seidel and Jacobi as they are both so similar. These methods require the coefficient matrix to be diagonally dominant. This not only means that each row must have a value on the diagonal but that diagonal value must also be larger than or equal to the sum of all other values in the row to guarantee solution convergence.

As the matrix `b1_ss.mtx` does not have non-zero values all the way down the diagonal my solver is unable to solve it. It would be possible to solve with some row swaps however my program does not have an implementation to do this. It is key to point out that the reason this matrix is unsolvable using this program is not because it is non-symmetric but because it is not diagonally dominant. A non-symmetric matrix that is diagonally dominant would be solvable using this program.

Additionally, the solver seems unable to solve `ACTIVSg70K.mtx` but can solve the other seven matrices to varying degrees of accuracy. I will go into the reasons why I think this is true in the next section.

5 Final Results and Analysis

Table 1: Results with `max_iteration = 25000` and `stop_criteria = 1e-16`

Problem	Size		Non-Zeros	Non-Zeros	CPU time	Norm of
	row	column	File Read	Total	(Seconds)	Residual
LFAT5.mtx	14	14	30	46	0.000152	1.6298e-14
LF10.mtx	18	18	50	82	0.003909	5.9306e-13
ex3.mtx	1821	1821	27253	52685	1.605369	3.6002e+01
jnlbrng1.mtx	40000	40000	119600	199200	0.844569	1.2662e-12
ACTIVSg70K.mtx	69999	69999	154313	238627	0.123565	-nan
2cubes_sphere.mtx	101492	101492	874378	1647264	107.6328	8.6670e-14
tmt_sym.mtx	726713	726713	2903837	5080961	492.3450	8.5272e+02
StocF-1465.mtx	1465137	1465137	11235263	21005389	1213.518	3.5499e+03

5.1 Discussion of Results

As can be seen in the table `LFAT5.mtx`, `LF10.mtx`, `jnlbrng1.mtx`, and `2cubes_sphere.mtx` all run very well and have a very good norm of the residual. This leads me to believe that all four of these matrices are diagonally dominant and work well with this algorithm.

However, `ex3.mtx` does not run as well as the others which completed. If I were to guess, I would say this matrix is likely ill-conditioned for being solved with the Gauss-Seidel method as my friends who also used an iterative method similar to the Gauss-Seidel were also having trouble with this matrix. My one friend who did not use an iterative method was able to solve this matrix to a much higher degree of accuracy, further strengthening my argument that these poor results are a result of using the Gauss-Seidel method or any iterative method. While relative to the other solutions this one is inaccurate that is not to say it is that bad relative to itself. Consider that between 1821 variables there is only a combined magnitude of error of about plus or minus 36, which works out to a very small degree of error for each individual variable.

Interestingly, `ACTIVSg70K.mtx` did run however it did not converge to a number and instead had an error outputting the final norm of the residual value. This leads me to believe that this matrix is not diagonally dominant as this unexpected behavior is like that of a problem that does not follow our solution constraints.

Testing the `tmt_sys.mtx` and `StocF-1465.mtx` matrices have similar results to the `ex3.mtx` matrix, however, I believe this is for different reasons. Instead of the relatively poor accuracy compared to the other test being from an ill-conditioned matrix, I believe it has more to do with the shear scale of these matrices. If more iterations were used I am sure that the norm of the residual could be reduced dramatically, however, this was not done as it already took a very long time to run the program with the current number of iterations. However, considering the shear scale of these matrices with over 700 000 and 1.4 million variables to calculate respectively, a total summed error in only the hundreds or thousands respectively is still very good for each individual variable.

5.2 Sparsity Visualizations

Below are several of the matrices from the table with their sparsity patterns visualized. The two largest matrices not shown below can be created using the command `./main filename.mtx plot` when executing the program. They were not included because each entry in the matrix is represented by a single pixel and these matrices are so sparse when the image is compressed to fix onto this page it appears as a black image with no points. To see the sparsity pattern, create the png using the command above, and once the program is done executing the sparsity can be observed by zooming in and out on specific areas of the matrix. All of these visualizations were

Figure 1: LFAT5.mtx

created using functions made by ChatGPT. To use the function and create a visualization the user must give any input after the filename, for example, `./main LFAT5.mtx plot` will create a sparsity pattern for `LFAT5.mtx`. There are two visualization functions one is for small matrices such as `LFAT5.mtx` while the other is for larger matrices such as `ex3.mtx`. The decision of which visualization function to use is made automatically by the program based on the number of rows and columns.

Figure 2: LF10.mtx

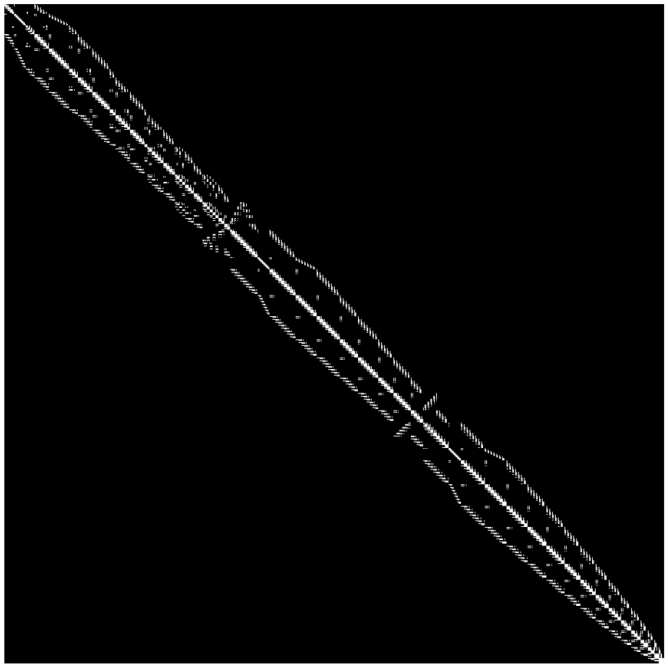

Figure 3: ex3.mtx

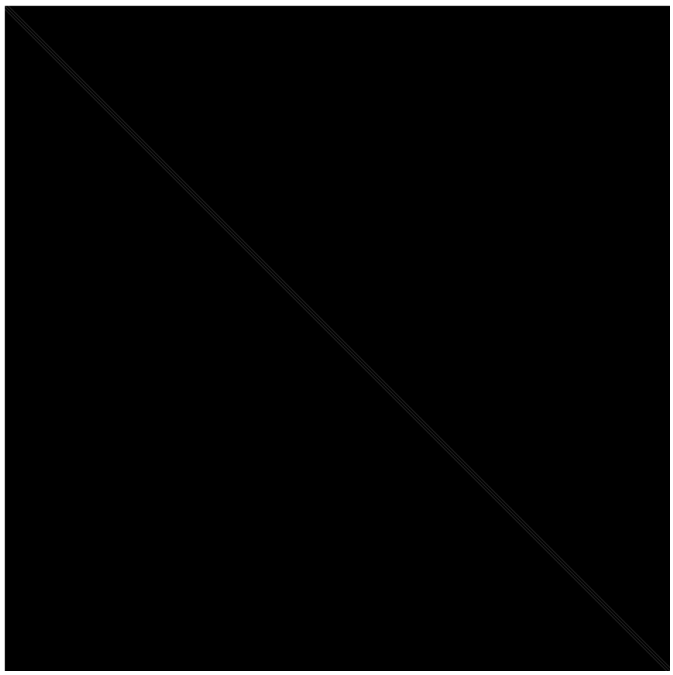

Figure 4: jnlbrng1.mtx

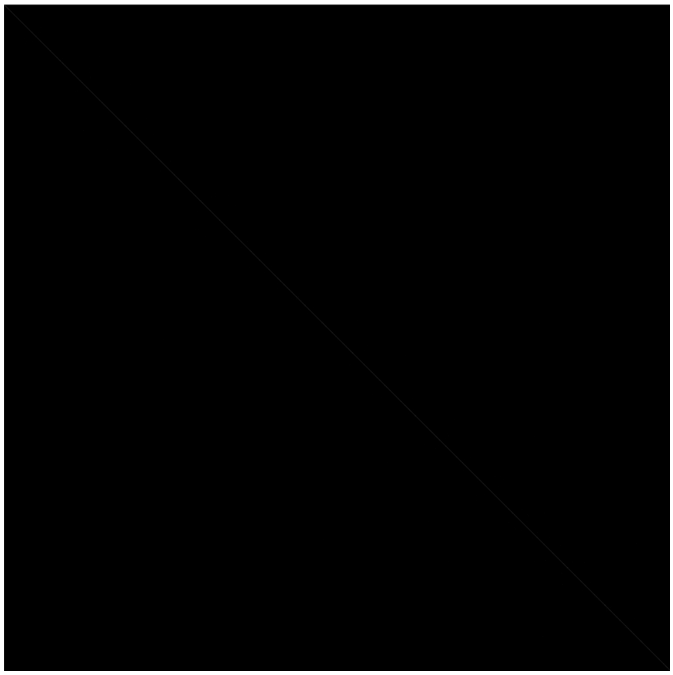

Figure 5: ACTIVSg70K.mtx

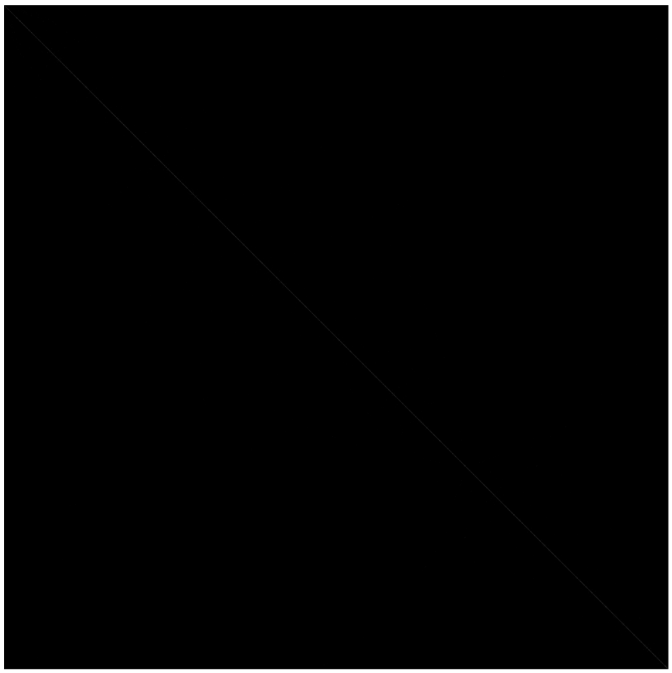

Figure 6: 2cubes_sphere.mtx

5.3 `gcov` and VTune

All testing done in this section used the `jnlbrng1.mtx` file as it was large enough to produce meaningful results while still being quick to run and do multiple tests quickly. Initially, when I ran VTune I noticed that the `ReadMMtoCSR` and `fill`$_s ymmetry$ functions were taking most of the CPU time for the program as seen in Figure 7. This was because I was using two for loops to sort all of the data based on the row data and this was very inefficient which is why I switched to using the `qsort()` function. This reduced the run time dramatically as seen in Figure 8. In both cases, the CPU utilization is poor likely due to the number of conditional statements and lack of consideration for parallel computing when writing the program. The terminal output from `gcov` is provided in the appendix but I decided to exclude the `gcov` file outputs from this report so as to not add 20 pages to the appendix. Looking at these results we are executing about 87% of the code in `main.c` which seems accurate as all of the ERROR messages were not executed as the program ran successfully. Looking at `functions.c` we see a much lower execution of about 54%. However, this can be explained as about half of the `functions.c` code is used for plotting the sparsity pattern of the matrix, and in this case, I did not include a third input and therefore did not call the if statement that would execute this code.

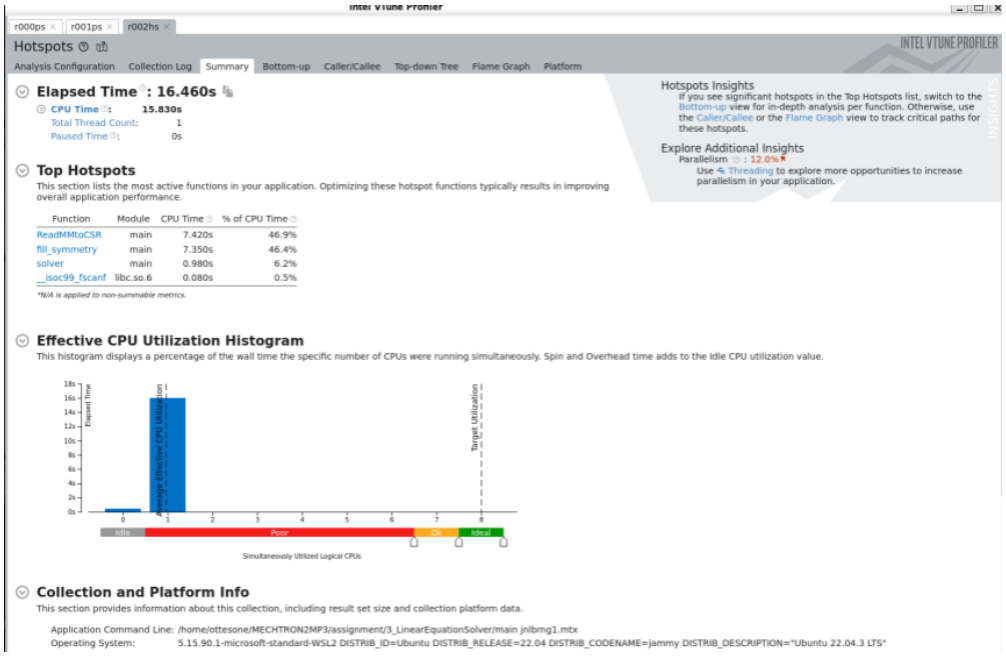

Figure 7: VTune results before adding the qsort function for the `jnlbrng1.mtx` matrix

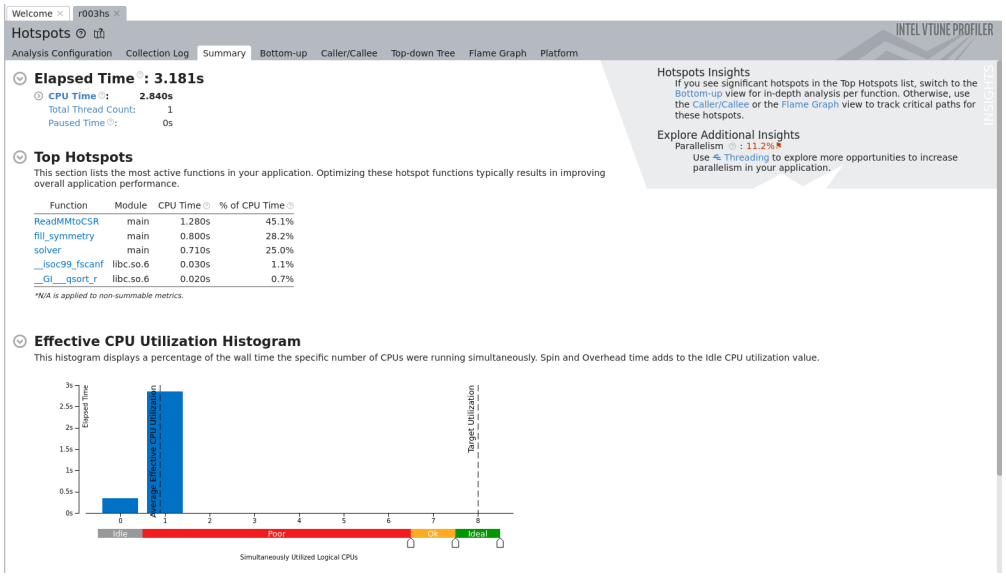

Figure 8: VTune results after to adding the qsort function for the `jnlbrng1.mtx` matrix

6 Makefile

The makefile is used in order to simply compile the `main.c` with one command as without it one would have to compile each file separately because of the multi-file dependency. The makefile I created and used for this assignment is shown and explained below, it is the same makefile as Assignment 2 with some modifications.

```
CC = gcc
CFLAGS = -Wall -Wextra -g -O3 -lm -lpng -fprofile-arcs -ftest-
   coverage
TARGET = main
SRC = main.c functions.c
OBJ = $(SRC:.c=.o)

$(TARGET): $(OBJ)
 $(CC) $(CFLAGS) -o $(TARGET) $(OBJ) -lm -lpng

%.o: %.c
 $(CC) $(CFLAGS) -c $<

clean:
 rm -f $(OBJ) $(TARGET)
```

- The first line defines 'gcc' to be used as the C compiler.
- The second line sets the flags which will be used to compile each file. `-Wall` and `-Wextra` enable most warning messages. While `-g` generates debugging and extra information so that the gdb tool can be used. `-O3` enables a high level of optimization to improve run time. `-lm` and `-lpng` are the library flags for the math and libpng libraries. The last two flags are used for `gcov` profiling.
- The third line specifies the name of the generated executable.
- The fourth line specifies all of the source files.
- The fifth line creates a list of object files replacing the .c extension with a .o extension for each of the files defined in the previous line
- The sixth line gives rules as to how to build the target executable from the object files.

- The seventh line is the exact rules that need to be followed when building the executable. Using the specified compiler and compiler flags create the target output file using the list of object files. Additionally, I added the flag for the math and libpng libraries here as when I only included it in the list of flags defined above, I would receive an error that certain elementary math functions and png functions were not defined.
- The eighth line is the generic rule for building the object files from their source file.
- The ninth line is the exact generic rule for the line above. Use the compiler and flags defined above, and "$ < " allows the use of automatic variables for each source file.
- The final couple of lines are used to remove all the files created by the makefile, if the command 'make clean' is run in the terminal it will automatically remove all of the object and target files created.

7 Appendix

```
File 'main.c'
Lines executed:86.67% of 60
Creating 'main.c.gcov'

File '/usr/include/x86_64-linux-gnu/bits/stdio2.h'
Lines executed:75.00% of 8
Creating 'stdio2.h.gcov'

File 'functions.c'
Lines executed:54.13% of 242
Creating 'functions.c.gcov'

Lines executed:60.97% of 310
```