MECHTRON 2MP3 Assignment 3

Developing a Biconjugate Gradient Stabilized Algorithm in C

Andrew De Rango, 400455362

1 Introduction

1.1 Assignment Overview

The goal of this assignment was to develop a program in C that is capable of solving first-order linear systems of equations. These systems are of the form Ax=b, where A is a known matrix, b is a known vector, and x is a vector whose elements must be solved for. The algorithm should be capable to solving very large and sparse matrices, up to millions of rows. In developing a solution, two parameters must be optimized/minimized: The runtime and the norm of the residual vector. The residual norm, or rather the magnitude of the residual vector, was the measure used to illustrate the accuracy of the solution given and can be defined by $||Ax - b||$.

It should be noted that very large matrices can impose significant memory issues when running the program. As the matrices being dealt with are sparse, it is better to store the matrices in CSR format, rather than storing every element of the 2D array in memory. This not only minimizes the memory allocated and reduces the incidence of segmentation fault, but it also improves runtime.

Furthermore, it would take a very long time to read the given matrices if each of their elements were given in a file. This would entail reading trillions of values, which is simply infeasible. To overcome this issue, files were provided and read from MTX format. For further information regarding MTX formatting, visit the Given Files folder of this repo.

1.2 Approach

The general process that the program developed for this assignment undergoes is described in this section.

First, it reads information from the MTX file and converts it directly to the CSR format. Then, it checks if the matrix is triangular. If the matrix is triangular, then it asks the user if they intended to solve the linear system where A is a simply the triangular matrix as provided, or if A was intended to be the corresponding symmetrical matrix. This occurs because a triangular matrix and its symmetrical counterpart, obtained by reflecting one triangular obliquely onto the other triangle, would have the same MTX file. In MTX format, there is no way to discern the

triangular matrix from the symmetrical. If the user decides to symmetrise the matrix, or reflect the triangle obliquely to the other triangle, then the program completes the rest of the operations using the symmetrical matrix.

Next, the program uses the `png.h` library to generate a PNG file that shows the sparsity of the matrix. This is an image that shows the sparsity pattern of the matrix, represented by black and white pixels showing where non-zero values exist in the matrix. White represents non-zero elements, while black pixels represent zeros. It creates a new directory if one has not already been created to add the PNG file to. The PNG file is always added to a folder in the directory that the user is currently in called *Sparsity Pattern Images*. The program notifies the user of this and specifies the file name and directory.

Following this, the program undergoes the iterative Biconjugate Gradient Stabilized (BiCGSTAB) algorithm and returns a solution vector x. It then computes and stores the residual and its norm. The program then executes the Conjugate Gradient algorithm and stores its residual norm. The program then compares the residual norms and takes the better (lower) one. The program calculates the program runtime, which accounts for both algorithms combined.

1.3 Biconjugate Gradient Stabilized Algorithm Overview

BiCGSTAB is a robust algorithm used to solve first-order linear systems of equations. It is an extension of the standard conjugate graduate method, in that it supports solutions for matrices that are both nonsymmetrical and indefinite. That is, the eigenvalues of the A matrix are not required to be positive in order to derive an accurate solution to the system. As an iterative method that uses the residual to define two new search directions (biconjugate gradient vectors), BiCGSTAB also has functionality that enforces stabilization which renders faster convergence.

2 Implementation in C

2.1 Structure Overview

The code can be broken down into four files:

- `Makefile`: Defines rules to be employed upon compilation.
- `functions.h`: Contains function prototypes that are formally defined in `functions.c`
- `main.c`: Contains the main function that calls functions from `functions.c`

- `functions.c`: Defines all functions other than `int main` such as `spvm_csr`, `bicgstab`, `conjugate_gradient`, and more.

2.2 Parameter Adjustment

There exists four arbitrary parameters within the program that may significantly change the resultant solution for x, and thus the residual vector and its norm. They are defined in the following lines:

```
bicgstab(csrMatrix, b, x, 1e-7, 10000);
conjugate_gradient(csrMatrix, b, x, 1e-7, 10000);
```

Both the BiCGSTAB and Conjugate Gradient algorithms take in the parameters `tolerance` and `max_iterations`. Above, they are defined as 0.0000001 and 10,000 respectively.

- `tolerance`: The iterations stop once the residual norm converges to below this specified tolerance. This should depend on the accuracy that the user is looking for in their specific circumstance.
- `max_iterations`: If the algorithms can't converge, then the iterations will stop after `max_iterations` iterations. This helps deal with cases in which BiCGSTAB will take a long time to converge to the true solution.

2.3 Makefile Summary

The Makefile supplied with the BiCGSTAB repository can be seen below:

```
CC = gcc
CFLAGS = -Wall -Wextra -g -lm -O2 -fprofile-arcs -ftest-coverage
    $(shell pkg-config --cflags libpng)
LIBS = $(shell pkg-config --libs libpng)

all: bicgstab

bicgstab: main.c functions.c functions.h
$(CC) $(CFLAGS) -o bicgstab main.c functions.c $(LIBS)

clean:
rm -f bicgstab
```

```
rm -rf bicgstab.dSYM
rm -f *.gcda
rm -f *.gcno
rm -f *.gcov
```

This Makefile is a set of instructions instigated from the command line that aids in the compilation and linking processes of potentially multiple source code files. An example is shown in the code block above. Here are the roles of the individual components within the Makefile:

`CC = gcc`: `CC` sets the compiler that will be used to compile the program. In this case, we are using gcc.

`CFLAGS = -Wall -Wextra -g -lm -O2 $(shell pkg-config --cflags libpng)`: `CFLAGS` lists the flags that will be used by the compiler defined above. Each dash represents a precursor for another flag. `-Wall` enables the compiler to display warning messages upon compilation, such as declared but unused variables within the program. `-Wextra` provides more potential warnings. `-g` enables the compiler to provide debugging information, and `-lm` helps connect the `math.h` header used for other files. `-O2` enables possible optimizations within the program to further reduce the computation time. `-fprofile-arcs` and `-ftest-coverage` support gcov profiling, which is shown at the bottom of this README document. The elements within the brackets configure the `libpng` package, which is needed for the sparse patterns.

`LIBS=$(shell pkg-config --libs libpng)`: Necessary for ligpng, helps configure libpng via the linker flag.

`all: bicgstab`: Defines bicgstab as the default target when running `make` without any following arguments.

`bicgstab`: `main.c`, `functions.c`, `functions.h`: This is the rule for building the target `bicgstab`. The target is built depending on `main.c`, `functions.c`, and `functions.h`. This is the part of the Makefile responsible for creating just one object file despite multiple C files. The line below, `$(CC) $(CFLAGS) -o bicgstab main.c functions.c $(LIBS)`, is responsible for building the target if it needs to be rebuilt.

`clean`: This is an independent rule. It does not depend on any of the parameters defined above. If the user runs the command `make clean` in the command line, then the listed files will be removed from the system. In the Makefile shown above, the `bicgstab`, `bicgstab.dSYM`, and gcov files, which are created upon compilation, will be deleted if the user executes the command `make clean`.

2.4 CSR Formatting

The CSR format of LFAT5.mtx can be seen below:

```
Number of 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
CSR Data: 1.570880 -94.252800 0.785440 12566400.000000 -6283200.000000
0.608806 -0.304403 -94.252800 15080.448000 -7540.224000 94.252800 0.785440
3.141760 -94.252800 0.785440 -6283200.000000 12566400.000000 -6283200.000000
-0.304403 0.608806 -0.304403 -7540.224000 -94.252800 15080.448000 -7540.224000
94.252800 94.252800 0.785440 3.141760  -94.252800 0.785440 -6283200.000000
12566400.000000 -0.304403 0.608806 -7540.224000 -94.252800 15080.448000
94.252800 94.252800 0.785440 3.141760 0.785440 94.252800 0.785440 1.570880
```

2.5 Dependencies

This program requires `png.h` in order to execute.

To install the dependency on Debian-based Linux, run the following:

```
sudo apt-get update
```

```
sudo apt-get install libpng-dev
```

On MacOS, use Homebrew:

```
brew update
```

```
brew install libpng
```

2.6 Running the Program

To run the program, the following commands must be run in the program's directory:

```
make
```

```
./bicgstab <filename.mtx>
```

For example,

`make`

`./bicgstab LFAT5.mtx`

3 Results

3.1 Performance

Table 1: BiCGSTAB Result for Different Matrices

Matrix	Dimensions	Non-Zeros	CPU Time (s)	Residual Norm
b1_ss.mtx	7x7	15	0.000010	0.000000
LFAT5.mtx	14x14	46	0.000038	0.000000
LF10.mtx	18x18	82	0.000141	0.000000
ex3.mtx	1821x1821	52685	1.011970	0.000112
jnlbrng1.mtx	40000x40000	199200	0.041605	0.000000
ACTIVSg70K.mtx	69999x69999	238627	14.121449	7595.457651
2cubes_sphere.mtx	101492x101492	1647264	66.212776	87.843514
tmt_sym.mtx	726713x726713	2903837	91.981226	0.182784
StocF-1465.mtx	1465137x1465137	11235263	84.903552	8179.010486

3.2 Gcov Reporting

Gcov reports were retrieved to further investigate the profiling of the program. The gcov reports for both `main.c` and `functions.c` can be seen in this section. These were retrieved from running the program with the input matrix A as LFAT5.mtx.

3.2.1 Code Coverage for `main.c`

```
        -:    0:Source:main.c
        -:    0:Graph:main.gcno
        -:    0:Data:main.gcda
        -:    0:Runs:2
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:#include <stdlib.h>
        -:    3:#include <math.h>
        -:    4:#include <time.h>
```

```
        -:    5:#include "functions.h"
        -:    6:
        1:    7:int main(int argc, char *argv[]) {
        -:    8:
        -:    9:    // Ensuring 2 arguments
        1:   10:    if (argc != 2) {
    #####:   11:        fprintf(stderr, "Sorry, I expected 2 arguments. Please
    use the following format: %s <filename.mtx>\n", argv[0]);
    #####:   12:        return 1;
        -:   13:    }
        -:   14:
        -:   15:    // Convert mtx file to CSRMatrix
        1:   16:    CSRMatrix *csrMatrix =
        (CSRMatrix*)malloc(sizeof(CSRMatrix));
        1:   17:    if (csrMatrix == NULL) {
    #####:   18:        fprintf(stderr, "Error allocating memory for
    CSRMatrix.\n");
    #####:   19:        exit(EXIT_FAILURE);
        -:   20:    }
        1:   21:    ReadMMtoCSR(argv[1], csrMatrix);
        -:   22:
        -:   23:    // Create a PNG image representing the sparsity pattern of
        the matrix
        1:   24:    createSparsePatternImage(csrMatrix->col_ind,
        csrMatrix->row_ptr, csrMatrix->num_rows, csrMatrix->num_cols, argv[1]);
        -:   25:
        -:   26:    // Initialize b and x vectors
        1:   27:    double* b = (double*)malloc(csrMatrix->num_rows *
        sizeof(double)); // RHS vector initialization
        1:   28:    double* x = (double*)malloc(csrMatrix->num_rows *
        sizeof(double)); // Solution vector initialization
        1:   29:    if (b == NULL) {
    #####:   30:        fprintf(stderr, "Error allocating memory for
    b vector.\n");
    #####:   31:        exit(EXIT_FAILURE);
        1:   32:    } else if (x == NULL) {
    #####:   33:        fprintf(stderr, "Error allocating memory for
    x vector.\n");
```

```
    #####:   34:        exit(EXIT_FAILURE);
        -:   35:    }
       15:   36:    for (int i = 0; i < csrMatrix->num_rows; i++) {
       14:   37:        b[i] = 1.0; // Assume b = [1, 1, ...]
       14:   38:        x[i] = 0.0;
       14:   39:    }
        -:   40:
        -:   41:    // Solve via BiCGSTAB and time it
        1:   42:    clock_t start = clock();
        1:   43:    bicgstab(csrMatrix, b, x, 5e-7, 10000);
        1:   44:    double* Ax = (double*)malloc(csrMatrix->num_rows *
     sizeof(double));
        1:   45:    spmv_csr(csrMatrix, x, Ax);
        1:   46:    double bicgstab_residual = 0.0;
       15:   47:    for (int i = 0; i < csrMatrix->num_rows; i++) {
       14:   48:        bicgstab_residual += (Ax[i] - b[i]) * (Ax[i] - b[i]);
       // Residual = Ax - b
        -:   49:        // printf("%lf ", x[i]); // Use this to print x
       14:   50:    }
        1:   51:    bicgstab_residual = sqrt(bicgstab_residual);
        -:   52:
        -:   53:    // Solve via Conjugate Gradient
        1:   54:    conjugate_gradient(csrMatrix, b, x, 5e-7, 10000);
        1:   55:    Ax = (double*)realloc(Ax, csrMatrix->num_rows *
     sizeof(double));
        1:   56:    spmv_csr(csrMatrix, x, Ax);
        1:   57:    double conj_grad_residual = 0.0;
       15:   58:    for (int i = 0; i < csrMatrix->num_rows; i++) {
       14:   59:        conj_grad_residual += (Ax[i] - b[i]) * (Ax[i] - b[i]);
       // Residual = Ax - b
        -:   60:        // printf("%lf ", x[i]); // Use this to print x
       14:   61:    }
        1:   62:    conj_grad_residual = sqrt(conj_grad_residual);
        -:   63:
        1:   64:    double optimal_residual = (bicgstab_residual <
     conj_grad_residual) ? bicgstab_residual : conj_grad_residual;
        -:   65:
        1:   66:    clock_t end = clock();
```

```
        1:   67:    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
        -:   68:
        1:   69:    printf("\nMatrix Name: %s\n", argv[1]);
        1:   70:    printf("Matrix Dimensions: %d x %d\n", csrMatrix->num_rows,
       csrMatrix->num_cols);
        1:   71:    printf("Number of Non-Zero Values: %d\n",
       csrMatrix->num_non_zeros);
        1:   72:    printf("Program Runtime: %f seconds\n", time_spent);
        -:   73:    // printf("Biconjugate Gradient Stabilzed Residual Norm:
       %f\n", bicgstab_residual);
        -:   74:    // printf("Conjugate Gradient Residual Norm: %f\n",
       conj_grad_residual);
        1:   75:    printf("Residual Norm: %f\n", optimal_residual);
        -:   76:
        1:   77:    free(b);
        1:   78:    free(x);
        1:   79:    free(Ax);
        1:   80:    free(csrMatrix->csr_data);
        1:   81:    free(csrMatrix->col_ind);
        1:   82:    free(csrMatrix->row_ptr);
        1:   83:    free(csrMatrix);
        -:   84:
        1:   85:    return 0;
        1:   86:}
```

3.2.2 Code Coverage for `functions.c`

```
        -:    0:Source:functions.c
        -:    0:Graph:functions.gcno
        -:    0:Data:functions.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:#include <stdlib.h>
        -:    3:#include <math.h>
        -:    4:#include <time.h>
        -:    5:#include <png.h>
        -:    6:#include <string.h>
```

```
        -:    7:#include <sys/stat.h>
        -:    8:#include <sys/types.h>
        -:    9:#include "functions.h"
        -:   10:
        2:   11:void ReadMMtoCSR(const char *filename, CSRMatrix *matrix) {
        -:   12:    // Try to open file
        2:   13:    FILE *file = NULL;
        2:   14:    file = fopen(filename, "r"); // Returns pointer to file
        if successful or NULL if unsuccessful
        -:   15:
        -:   16:    // If the file doesn't exist or can't open it, print error
        message and quit the program
        2:   17:    if (file == NULL) {
        1:   18:        perror("Error opening file");
        1:   19:        exit(EXIT_FAILURE);
        -:   20:    }
        -:   21:
        1:   22:    int rows = 0, columns = 0, nonzero_values = 0, *row_ind,
        lower_triangular = 1, upper_triangular = 1;
        1:   23:    double mtx_row, mtx_column, mtx_value;
        1:   24:    char *line = NULL;
        1:   25:    size_t len = 0;
        1:   26:    ssize_t read;
        -:   27:
        -:   28:    // Throw away all the lines that start with %
        1:   29:    do {
       18:   30:        read = getline(&line, &len, file);
       18:   31:    } while (read != -1 && line[0] == '%');
        -:   32:
        -:   33:    // If reach EOF before finding non-% line
        1:   34:    if (read == -1) {
    #####:   35:        fprintf(stderr, "This MTX format is invalid.\n");
    #####:   36:        exit(EXIT_FAILURE);
        -:   37:    }
        -:   38:
        1:   39:    sscanf(line, "%d %d %d", &rows, &columns, &nonzero_values);
        1:   40:    free(line);
        -:   41:
```

```
        1:   42:    matrix->num_rows = rows;
        1:   43:    matrix->num_cols = columns;
        1:   44:    matrix->num_non_zeros = nonzero_values;
        -:   45:
        1:   46:    matrix->csr_data = (double*)malloc(nonzero_values *
       sizeof(double));
        1:   47:    matrix->col_ind = (int*)malloc(nonzero_values *
       sizeof(int));
        1:   48:    matrix->row_ptr = (int*)malloc((rows + 1) * sizeof(int));
        1:   49:    row_ind = (int*)malloc(nonzero_values * sizeof(int));
        -:   50:
        1:   51:    if (matrix->csr_data == NULL || matrix->col_ind == NULL
       || matrix->row_ptr == NULL || row_ind == NULL) {
    #####:   52:        fprintf(stderr, "Error allocating memory for
    CSRMatrix.\n");
    #####:   53:        exit(EXIT_FAILURE);
        -:   54:    }
        -:   55:
        -:   56:    // Read the values into the CSRMatrix
        1:   57:    int max_col_seen = 0;
       31:   58:    for (int i = 0; i < matrix->num_non_zeros; i++) {
       30:   59:        fscanf(file, "%lf %lf %lf", &mtx_row, &mtx_column,
       &mtx_value);
        -:   60:
        -:   61:        // Assures MTX file is sorted by columns (major),
        then rows (minor) ascending
       30:   62:        if (mtx_column < max_col_seen) {
    #####:   63:            printf("This is improper MTX format. MTX format
    requires the column-index column to be ascending.\n");
    #####:   64:            exit(EXIT_FAILURE);
        -:   65:        } else {
       30:   66:            max_col_seen = mtx_column;
        -:   67:        }
        -:   68:
        -:   69:        // Even if 0 values are in the MTX file, they are
        not stored in the CSRMatrix
       30:   70:        if (mtx_value != 0.0) {
       30:   71:            matrix->csr_data[i] = mtx_value;
```

```
       30:   72:            matrix->col_ind[i] = mtx_column - 1; // (1, 1)
       in MTX is (0, 0) in CSR
       30:   73:            row_ind[i] = mtx_row - 1; // (1, 1) in MTX is
       (0, 0) in CSR
        -:   74:
        -:   75:            // Check if matrix is triangular
       30:   76:            if (mtx_row > mtx_column) {
       16:   77:                lower_triangular = 0;
       30:   78:            } else if (mtx_row < mtx_column) {
    #####:   79:                upper_triangular = 0;
    #####:   80:            }
       30:   81:        }
       30:   82:    }
        -:   83:
        -:   84:    // Initialize row_ptr to 0
       16:   85:    for (int i = 0; i < rows+1; i++) {
       15:   86:        matrix->row_ptr[i] = 0;
       15:   87:    }
        -:   88:
        -:   89:    // Count the number of non-zero elements in each row
       31:   90:    for (int i = 0; i < matrix->num_non_zeros; i++) {
       30:   91:        matrix->row_ptr[row_ind[i] + 1]++;
       30:   92:    }
        -:   93:
        -:   94:    // Cumulate row_ptr such that the value at each index is
        the sum of the values before it
       15:   95:    for (int i = 0; i < rows; i++) {
       14:   96:        matrix->row_ptr[i + 1] += matrix->row_ptr[i];
       14:   97:    }
        -:   98:
        -:   99:    // Assign data to MTXRow struct to sort
        -:  100:    // We need to sort because if you don't then CSR will
        be in the order of the MTX file
        -:  101:    // MTX reads up to down then left to right. CSR needs to
        read left to right then up to down.
        1:  102:    MTXRow *data = (MTXRow*)malloc(matrix->num_non_zeros
        * sizeof(MTXRow));
        1:  103:    if (data == NULL) {
```

```
#####:  104:        fprintf(stderr, "Error allocating memory for data.\n");
#####:  105:        exit(EXIT_FAILURE);
    -:  106:    }
   31:  107:    for (int i = 0; i < matrix->num_non_zeros; i++) {
   30:  108:        data[i].row = row_ind[i];
   30:  109:        data[i].col = matrix->col_ind[i];
   30:  110:        data[i].value = matrix->csr_data[i];
   30:  111:    }
    1:  112:    qsort(data, matrix->num_non_zeros, sizeof(MTXRow),
    compareMTXData);
    -:  113:
    -:  114:    // Rewrite the sorted data so that it is in the
    correct order
   31:  115:    for (int i = 0; i < matrix->num_non_zeros; i++) {
   30:  116:        row_ind[i] = data[i].row;
   30:  117:        matrix->col_ind[i] = data[i].col;
   30:  118:        matrix->csr_data[i] = data[i].value;
   30:  119:    }
    -:  120:
    1:  121:    free(data);
    1:  122:    fclose(file);
    -:  123:
    -:  124:    // Ask user if they want to symmetrise the matrix if
    it is triangular
    1:  125:    if (upper_triangular == 1 && lower_triangular == 0) {
    1:  126:        printf("I have detected that this matrix is
    upper triangular. Would you like to symmetrise it? (Y/N): ");
    1:  127:    } else if (upper_triangular == 0 &&
    lower_triangular == 1) {
#####:  128:        printf("I have detected that this matrix is lower
triangular. Would you like to symmetrise it? (Y/N): ");
#####:  129:    }
    -:  130:
    -:  131:    // Symmetrise the matrix iff user wants to and it is
    triangular
    1:  132:    if (upper_triangular == 1 || lower_triangular == 1) {
    1:  133:        char symmetrise;
    1:  134:        scanf(" %c", &symmetrise);
```

```
        1:  135:        while (symmetrise != 'y' && symmetrise != 'Y' &&
        symmetrise != '1' && symmetrise != 'n' && symmetrise != 'N'
        && symmetrise != '0') {
    #####:  136:            printf("Invalid response. Please enter Y or N: ");
    #####:  137:            scanf(" %c", &symmetrise);
        -:  138:        }
        1:  139:        if (symmetrise == 'y' || symmetrise == 'Y' ||
        symmetrise == '1') {
        1:  140:            int og_num_non_zeros = matrix->num_non_zeros;
        -:  141:            // Loop through each non-zero element.
       31:  142:            for (int i = 0; i < og_num_non_zeros; i++) {
       30:  143:                if (matrix->col_ind[i] != row_ind[i]) {
       16:  144:                    matrix->num_non_zeros++;
       16:  145:                    matrix->col_ind = realloc(
       matrix->col_ind, matrix->num_non_zeros * sizeof(int));
       16:  146:                    row_ind = realloc(row_ind,
       matrix->num_non_zeros * sizeof(int));
       16:  147:                    matrix->csr_data = realloc(
       matrix->csr_data, matrix->num_non_zeros * sizeof(double));
       16:  148:                    matrix->col_ind[matrix->num_non_zeros
       - 1] = row_ind[i];
       16:  149:                    row_ind[matrix->num_non_zeros - 1]
       = matrix->col_ind[i];
       16:  150:                    matrix->csr_data[
       matrix->num_non_zeros - 1] = matrix->csr_data[i];
       16:  151:                }
       30:  152:            }
        -:  153:
        -:  154:            // Sort the array again
        1:  155:            MTXRow* data = (MTXRow*)malloc(matrix-
        >num_non_zeros * sizeof(MTXRow));
        1:  156:            if (data == NULL) {
    #####:  157:                fprintf(stderr, "Error allocating memory
    for data.\n");
    #####:  158:                exit(EXIT_FAILURE);
        -:  159:            }
        -:  160:            // Assign data to MTXRow struct to sort
       47:  161:            for (int i = 0; i < matrix->num_non_zeros; i++) {
```

```
        46:  162:                data[i].row = row_ind[i];
        46:  163:                data[i].col = matrix->col_ind[i];
        46:  164:                data[i].value = matrix->csr_data[i];
        46:  165:            }
         1:  166:            qsort(data, matrix->num_non_zeros,
         sizeof(MTXRow), compareMTXData);
         -:  167:            // Rewrite sorted data to existing CSR matrix
        47:  168:            for (int i = 0; i < matrix->num_non_zeros; i++) {
        46:  169:                row_ind[i] = data[i].row;
        46:  170:                matrix->col_ind[i] = data[i].col;
        46:  171:                matrix->csr_data[i] = data[i].value;
        46:  172:            }
         1:  173:            free(data);
         -:  174:            // Re-compute row_ptr
         -:  175:            // Initialize row_ptr to 0
        16:  176:            for (int i = 0; i < rows+1; i++) {
        15:  177:                matrix->row_ptr[i] = 0;
        15:  178:            }
         -:  179:            // Count number of non-zero elements in each row
        47:  180:            for (int i = 0; i < matrix->num_non_zeros; i++) {
        46:  181:                matrix->row_ptr[row_ind[i] + 1]++;
        46:  182:            }
         -:  183:            // Cumulate row_ptr such that the value at each
         index is the sum of the values before it
        15:  184:            for (int i = 0; i < rows; i++) {
        14:  185:                matrix->row_ptr[i + 1] += matrix->row_ptr[i];
        14:  186:            }
         1:  187:        }
         1:  188:    }
         -:  189:
         1:  190:    free(row_ind);
         1:  191:}
         -:  192:
         -:  193:// Sorting criteria for qsort
         -:  194:// Sort by row then column
       367:  195:int compareMTXData(const void *a, const void *b) {
         -:  196:    // Cast void pointers to MTXRow pointers
         to access row and col
```

```
      367:  197:    MTXRow *rowA = (MTXRow*)a;
      367:  198:    MTXRow *rowB = (MTXRow*)b;
        -:  199:
        -:  200:    // Compare rows
      367:  201:    if (rowA->row != rowB->row) {
      297:  202:        return rowA->row - rowB->row;
        -:  203:    } else {
       70:  204:        return rowA->col - rowB->col; // Compare columns
       if rows are equal
        -:  205:    }
      367:  206:}
        -:  207:
        -:  208:// Function to create a PNG image representing the
        sparsity pattern of a CSR matrix
        -:  209:// This function was adapted from ChatGPT
        1:  210:void createSparsePatternImage(const int *columns, const
        int *row_ptr, int num_rows, int num_cols, const char
        *mtx_file_name) {
        -:  211:
        -:  212:    // Large matrices would make the image file size too
        large to view
        1:  213:    if (num_rows > 50000 || num_cols > 50000) {
    #####:  214:        fprintf(stderr, "Sorry, the matrix is too large
    to create a sparsity pattern image.\n");
    #####:  215:        return;
        -:  216:    }
        -:  217:
        -:  218:    // Remove extension of input matrix file name
        1:  219:    char *dot = strchr(mtx_file_name, '.');
        1:  220:    size_t base_name_length = dot != NULL ? dot -
        mtx_file_name : strlen(mtx_file_name);
        1:  221:    char base_name[256]; // Max file name size = 256
        1:  222:    strncpy(base_name, mtx_file_name, base_name_length);
        1:  223:    base_name[base_name_length] = '\0';
        -:  224:
        -:  225:    // Image size and scale factor
        1:  226:    int scale_factor;
        1:  227:    if (num_rows * num_cols > 5000) {
```

```
    #####:  228:        scale_factor = 1;
        1:  229:    } else if (num_rows * num_cols > 500) {
    #####:  230:        scale_factor = 10;
    #####:  231:    } else {
        1:  232:        scale_factor = 100;
        -:  233:    }
        1:  234:    uint64_t image_width = num_cols * scale_factor;
        1:  235:    uint64_t image_height = num_rows * scale_factor;
        -:  236:
        -:  237:    // Create a buffer to hold pixel data
        1:  238:    uint8_t* pixel_data = (uint8_t*)calloc(image_width *
        image_height, sizeof(uint8_t)); // Intilized all to 0, all
        pixels are black
        1:  239:    if (pixel_data == NULL) {
    #####:  240:        fprintf(stderr, "Error allocating memory for pixel data.\n");
    #####:  241:        exit(EXIT_FAILURE);
        -:  242:    }
        -:  243:    // Populate pixel_data based on the sparse matrix
       15:  244:    for (int i = 0; i < num_rows; ++i) {
       60:  245:        for (int j = row_ptr[i]; j < row_ptr[i + 1]; ++j) {
       46:  246:            int col = columns[j];
        -:  247:            // Set the pixel_data to white for each non-zero element
        -:  248:            // Need double for loop to set multiple pixels for each non-z
     4646:  249:            for (int di = 0; di < scale_factor; ++di) {
   464600:  250:                for (int dj = 0; dj < scale_factor; ++dj) {
   460000:  251:                    pixel_data[(i * scale_factor + di) *
   image_width + col * scale_factor + dj] = 255; // 255 is white
   460000:  252:                }
     4600:  253:            }
       46:  254:        }
       14:  255:    }
        -:  256:
        -:  257:    // Create an array of pointers to the rows in the image
        1:  258:    uint8_t** rows = (uint8_t**)malloc
        (image_height * sizeof(uint8_t*));
        1:  259:    if (rows == NULL) {
    #####:  260:        fprintf(stderr, "Memory allocation error.\n");
    #####:  261:        free(pixel_data);
```

```
    #####:  262:        exit(EXIT_FAILURE);
        -:  263:    }
     1401:  264:    for (uint32_t i = 0; i < image_height; ++i) {
     1400:  265:        rows[i] = &pixel_data[i * image_width];
     1400:  266:    }
        -:  267:
        -:  268:    // Check if the directory exists. If not, create it.
        1:  269:    struct stat st = {0};
        1:  270:    if (stat("Sparsity Pattern Images", &st) == -1) {
        -:  271:        // Create the directory
        -:  272:        #ifdef _WIN32
        -:  273:        _mkdir("Sparsity Pattern Images");
        -:  274:        #else
    #####:  275:        mkdir("Sparsity Pattern Images", 0700);
        -:  276:        #endif
    #####:  277:    }
        -:  278:
        -:  279:    // Write the image data to a file
        1:  280:    char file_name[256];  // Adjust the size as needed
        1:  281:    sprintf(file_name, "Sparsity Pattern
        Images/%s_sparsity_pattern.png", base_name); // Create file name
        and assign to file_name
        1:  282:    FILE* fp = fopen(file_name, "wb"); // Open file for writing
        1:  283:    if (!fp) abort(); // If file can't be opened, abort
        1:  284:    png_structp png =
        png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL,
        NULL); // Create png struct to write PNG data
        1:  285:    if (!png) abort(); // If png struct can't be
        created, abort
        1:  286:    png_infop info = png_create_info_struct(png); //
        Create png info struct to store image metadata
        1:  287:    if (!info) abort(); // If png info struct can't be
        created, abort
        -:  288:
        1:  289:    if (setjmp(png_jmpbuf(png))) abort(); // If any of
        the following libpng operations fail, abort
        1:  290:    png_init_io(png, fp); // Initialize the IO
        -:  291:
```

```
        -:  292:    // Write image header data
        1:  293:    png_set_IHDR(
        1:  294:        png,
        1:  295:        info,
        1:  296:        image_width, image_height,
        -:  297:        8,
        -:  298:        PNG_COLOR_TYPE_GRAY,
        -:  299:        PNG_INTERLACE_NONE,
        -:  300:        PNG_COMPRESSION_TYPE_DEFAULT,
        -:  301:        PNG_FILTER_TYPE_DEFAULT
        -:  302:    );
        1:  303:    png_write_info(png, info); // Write image info
        1:  304:    png_write_image(png, rows); // Write image data
        1:  305:    png_write_end(png, NULL); // Write the end of the PNG file
        1:  306:    if (png && info)
        1:  307:        png_destroy_write_struct(&png, &info); // Free allocated memory
        1:  308:    if (fp)
        1:  309:        fclose(fp); // Close file
        -:  310:
        1:  311:    free(rows);
        1:  312:    free(pixel_data);
        -:  313:
        1:  314:    printf("Sparsity pattern image saved to your current
       directory: %s\n", file_name);
        1:  315:}
        -:  316:
        -:  317:// Function to multiply matrix by vector in CSR format.
      306:  318:void spmv_csr(const CSRMatrix *A, const double *x,
      double *y) {
     4590:  319:    for (int i = 0; i < A->num_rows; i++) {
     4284:  320:        y[i] = 0.0;
    18360:  321:        for (int j = A->row_ptr[i]; j < A->row_ptr[i+1]; j++) {
    14076:  322:            y[i] += A->csr_data[j] * x[A->col_ind[j]];
    14076:  323:        }
     4284:  324:    }
      306:  325:}
        -:  326:
        -:  327:// Function to solve a linear system of equations using
```

```
        the BiCGSTAB method
        1:  328:void bicgstab(CSRMatrix *A, double *b, double *x, double
        tolerance, int max_iterations) {
        1:  329:    double* Ax = (double*)malloc(A->num_rows *
        sizeof(double)); // LHS vector
        1:  330:    double* r = (double*)malloc(A->num_rows *
        sizeof(double)); // Residual vector
        1:  331:    double* r_hat = (double*)malloc(A->num_rows *
        sizeof(double)); // Biorthogonalized residual vector
        1:  332:    double rho = 1.0; // Dot product of residual and
        biorthogonalized residual
        1:  333:    double alpha = 1.0; // Step size for search
        direction. Alpha = rho/(rhat*v)
        1:  334:    double omega = 1.0; // Step size for stabilization.
        Omega = (rhat*r)/(rhat*v)
        1:  335:    double* v = (double*)malloc(A->num_rows *
        sizeof(double)); // Temporary storage of A * p
        1:  336:    double* p = (double*)malloc(A->num_rows *
        sizeof(double)); // Search direction vector
        1:  337:    double best_residual; // Best residual so far
        1:  338:    double* best_x = (double*)malloc(A->num_rows *
        sizeof(double)); // Best solution so far
        -:  339:
        1:  340:    if (Ax == NULL || r == NULL || r_hat == NULL || v ==
        NULL || p == NULL || best_x == NULL) {
    #####:  341:        fprintf(stderr, "Error allocating memory for
    vectors in BiCGSTAB function.\n");
    #####:  342:        exit(EXIT_FAILURE);
        -:  343:    }
        -:  344:
        1:  345:    spmv_csr(A, x, Ax);
        -:  346:
       15:  347:    for (int i = 0; i < A->num_rows; i++) {
       14:  348:        r[i] = b[i] - Ax[i];
       14:  349:        r_hat[i] = r[i];
       14:  350:        v[i] = 0.0;
       14:  351:        p[i] = 0.0;
       14:  352:    }
```

```
        -:  353:
      145:  354:    for (int iteration = 0; iteration < max_iterations;
      iteration++) {
      144:  355:        double rho_new = 0.0;
     2160:  356:        for (int i = 0; i < A->num_rows; i++) {
     2016:  357:            rho_new += r_hat[i] * r[i];
     2016:  358:        }
      144:  359:        double beta = (rho_new / (rho + 1e-16)) * (alpha
      / (omega + 1e-16));
      144:  360:        rho = rho_new;
     2160:  361:        for (int i = 0; i < A->num_rows; i++) {
     2016:  362:            p[i] = r[i] + beta * (p[i] - omega * v[i]);
     2016:  363:        }
      144:  364:        spmv_csr(A, p, v);
      144:  365:        double dot_r_hat_v = 0.0;
     2160:  366:        for (int i = 0; i < A->num_rows; i++) {
     2016:  367:            dot_r_hat_v += r_hat[i] * v[i];
     2016:  368:        }
      144:  369:        alpha = rho / dot_r_hat_v;
      144:  370:        double* s = (double*)malloc(A->num_rows *
      sizeof(double)); // Temporary storage r - alpha * v
      144:  371:        double* t = (double*)malloc(A->num_rows *
      sizeof(double)); // Temporary storage A * s
      144:  372:        if (s == NULL || t == NULL) {
    #####:  373:            fprintf(stderr, "Error allocating memory for
    s or t vector.\n");
    #####:  374:            exit(EXIT_FAILURE);
        -:  375:        }
     2160:  376:        for (int i = 0; i < A->num_rows; i++) {
     2016:  377:            s[i] = r[i] - alpha * v[i];
     2016:  378:        }
      144:  379:        spmv_csr(A, s, t);
      144:  380:        double dot_t_s = 0.0;
      144:  381:        double dot_t_t = 0.0;
     2160:  382:        for (int i = 0; i < A->num_rows; i++) {
     2016:  383:            dot_t_s += t[i] * s[i];
     2016:  384:            dot_t_t += t[i] * t[i];
     2016:  385:        }
```

```
144:  386:        omega = dot_t_s / (dot_t_t + 1e-16);
2160:  387:        for (int i = 0; i < A->num_rows; i++) {
2016:  388:            x[i] += alpha * p[i] + omega * s[i];
2016:  389:            r[i] = s[i] - omega * t[i];
2016:  390:        }
   -:  391:
 144:  392:        double residual = 0.0;
2160:  393:        for (int i = 0; i < A->num_rows; i++) {
2016:  394:            residual += r[i] * r[i];
2016:  395:        }
 144:  396:        residual = sqrt(residual);
 144:  397:        if (iteration == 0 || residual < best_residual) {
  50:  398:            best_residual = residual;
 750:  399:            for (int i = 0; i < A->num_rows; i++) {
 700:  400:                best_x[i] = x[i];
 700:  401:            }
  50:  402:        }
   -:  403:
   -:  404:        // printf("Iteration: %d\tResidual: %lf\tBest
  Residual: %lf\n", iteration, residual, best_residual);
   -:  405:
 144:  406:        if (residual < tolerance) {
   1:  407:            break;
   -:  408:        }
   -:  409:
 143:  410:        free(s);
 143:  411:        free(t);
 144:  412:    }
   -:  413:
   1:  414:    free(Ax);
   1:  415:    free(r);
   1:  416:    free(r_hat);
   1:  417:    free(v);
   1:  418:    free(p);
   -:  419:
  15:  420:    for (int i = 0; i < A->num_rows; i++) {
  14:  421:        x[i] = best_x[i];
  14:  422:    }
```

```
        -:  423:    // printf("Residual: %f\n", best_residual);
        1:  424:}
        -:  425:
        -:  426:// Function to solve a linear system of equations using
        the Conjugate Gradient method
        1:  427:void conjugate_gradient(CSRMatrix* A, double* b, double*
        x, double tolerance, int max_iterations) {
        1:  428:    double* r = (double*)malloc(A->num_rows *
        sizeof(double)); // Residual vector
        1:  429:    double* p = (double*)malloc(A->num_rows *
        sizeof(double)); // Search direction vector
        1:  430:    double* Ap = (double*)malloc(A->num_rows *
        sizeof(double)); // Temporary storage of A * p
        1:  431:    double best_residual;
        1:  432:    double* best_x = (double*)malloc(A->num_rows *
        sizeof(double));
        -:  433:
        1:  434:    if (r == NULL || p == NULL || Ap == NULL || best_x
        == NULL) {
    #####:  435:        fprintf(stderr, "Error allocating memory for
    vectors in conjugate gradient function.\n");
    #####:  436:        exit(EXIT_FAILURE);
        -:  437:    }
        -:  438:
        1:  439:    spmv_csr(A, x, Ap);
        -:  440:
       15:  441:    for (int i = 0; i < A->num_rows; i++) {
       14:  442:        r[i] = b[i] - Ap[i];
       14:  443:        p[i] = r[i];
       14:  444:    }
        1:  445:    double r_dot_r = 0;
       15:  446:    for (int i = 0; i < A->num_rows; i++) {
       14:  447:        r_dot_r += r[i]*r[i];
       14:  448:    }
        -:  449:
        8:  450:    for (int iteration = 0; iteration < max_iterations;
        iteration++) {
        7:  451:        spmv_csr(A, p, Ap);
```

```
        -:  452:
        7:  453:        double p_dot_Ap = 0.0;
      105:  454:        for (int i = 0; i < A->num_rows; i++) {
       98:  455:            p_dot_Ap += p[i] * Ap[i];
       98:  456:        }
        7:  457:        double alpha = r_dot_r / (p_dot_Ap + 1e-16);
      105:  458:        for (int i = 0; i < A->num_rows; i++) {
       98:  459:            x[i] += alpha * p[i];
       98:  460:            r[i] -= alpha * Ap[i];
       98:  461:        }
        7:  462:        double r_dot_r_new = 0;
      105:  463:        for (int i = 0; i < A->num_rows; i++) {
       98:  464:            r_dot_r_new += r[i]*r[i];
       98:  465:        }
        7:  466:        double beta = r_dot_r_new / (r_dot_r + 1e-16);
      105:  467:        for (int i = 0; i < A->num_rows; i++) {
       98:  468:            p[i] = r[i] + beta * p[i];
       98:  469:        }
        7:  470:        r_dot_r = r_dot_r_new;
        -:  471:
        -:  472:        // Compute the residual norm
        7:  473:        double residual = 0.0;
        7:  474:        double* Ax = (double*)malloc(A->num_rows *
        sizeof(double));
        7:  475:        spmv_csr(A, x, Ax);
      105:  476:        for (int i = 0; i < A->num_rows; i++) {
       98:  477:            residual += (Ax[i] - b[i]) * (Ax[i] - b[i]);
       // Residual = Ax - b
       98:  478:        }
        7:  479:        residual = sqrt(residual);
        7:  480:        free(Ax);
        -:  481:
        7:  482:        if (iteration == 0 || residual < best_residual) {
        3:  483:            best_residual = residual;
       45:  484:            for (int i = 0; i < A->num_rows; i++) {
       42:  485:                best_x[i] = x[i];
       42:  486:            }
        3:  487:        }
```

```
        -:  488:
        -:  489:        // printf("Iteration: %d\tResidual: %lf\tBest
       Residual: %lf\n", iteration, residual, best_residual);
        -:  490:
        7:  491:        if (residual < tolerance) {
        1:  492:            residual = best_residual;
        1:  493:            break;
        -:  494:        }
        7:  495:    }
        -:  496:
        1:  497:    free(r);
        1:  498:    free(p);
        1:  499:    free(Ap);
        -:  500:
       15:  501:    for (int i = 0; i < A->num_rows; i++) {
       14:  502:        x[i] = best_x[i];
       14:  503:    }
        -:  504:    // printf("Residual: %f\n", best_residual);
        1:  505:}
```

3.3 VTune Analysis

Further profiling of the program was done in VTune to acquire deeper insights into its performance. The VTune report for `jnlbrng1.mtx` as the A matrix is shown below.

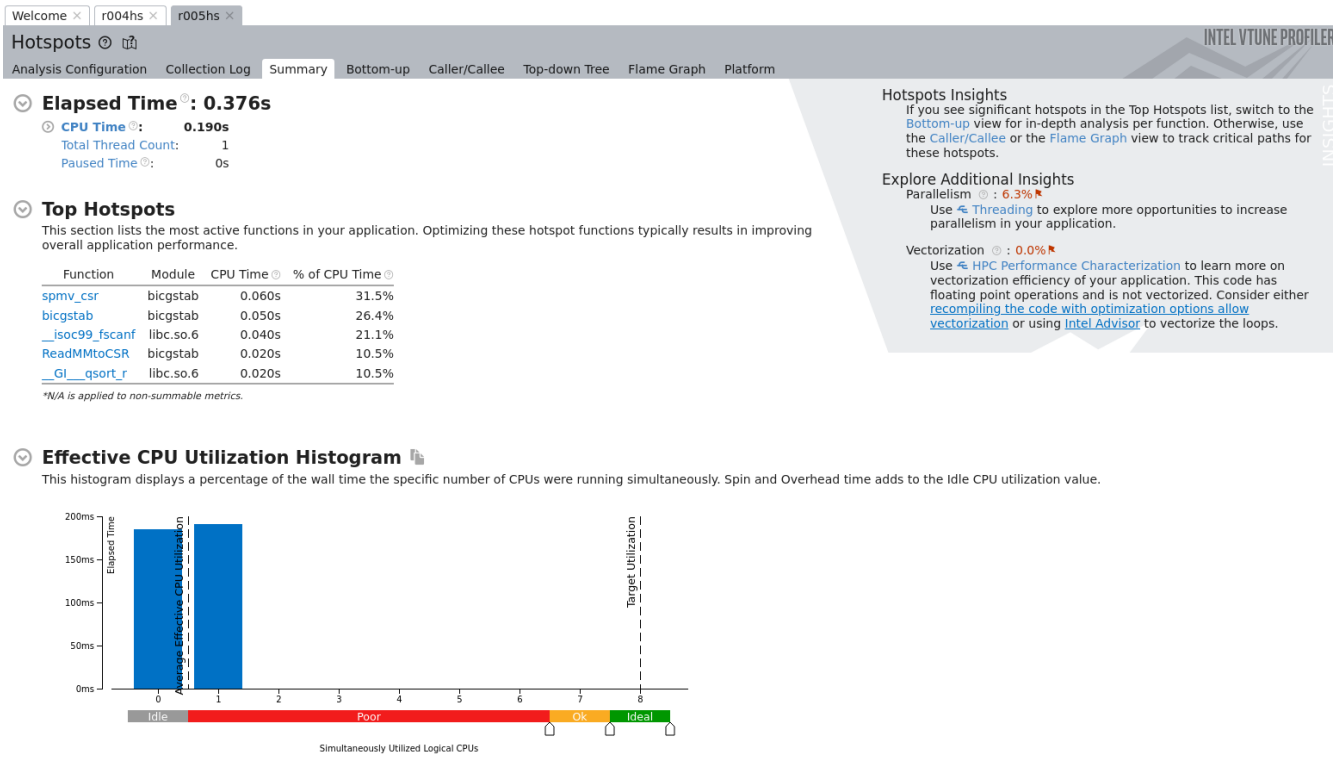

Figure 1: VTune report of `jnlbrng1.mtx`.

3.4 Sparse Pattern Visualization

As mentioned in Chapter 1, sparse pattern visualizations were implemented to determine the pattern of non-zero numbers in the input matrix. These can be seen below:

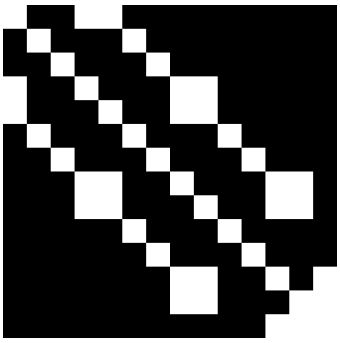

Figure 2: Sparse Pattern Image of `LFAT5.mtx`.

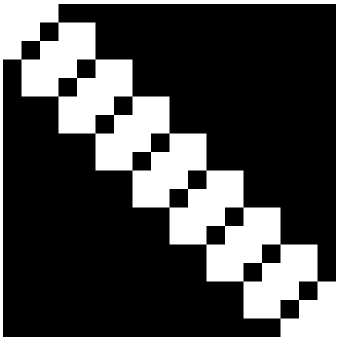

Figure 3: Sparse Pattern Image of `LF10.mtx`.

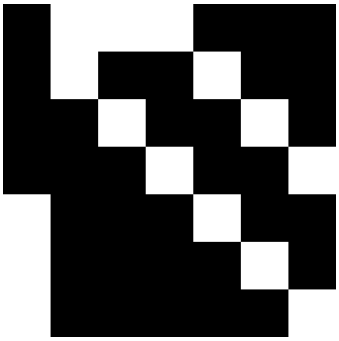

Figure 4: Sparse Pattern Image of `b1_ss.mtx`.

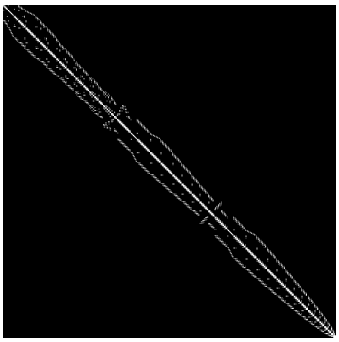

Figure 5: Sparse Pattern Image of `ex3.mtx`.