CSC190: Computer Algorithms and Data Structures
Assignment 2
Assigned: Feb 3, 2017; Due: Feb 24, 2017 @ 10:00 a.m.

1 Objectives

A dense matrix is a matrix in which almost all matrix entries are non-zero. On the other hand, a sparse matrix contains entries that are mostly zero. In this assignment, you will implement the Strassen's algorithm for multiplying two dense square matrices. The typical method used to multiply two matrices is:

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

where $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$ are the matrices being multiplied, $C \in \mathbb{R}^{n \times p}$ is the result of this multiplication and $C_{i,j}$ is a single element in matrix C residing in the i^{th} row and j^{th} column. This multiplication can be implemented using three nested loops in C and will be referred to as regular matrix multiplication in the remainder of this manual. If the matrices that are being multiplied are square matrices then the complexity of regular matrix multiplication is O(n^3) where n is the size of the square matrices. As the size of the matrices grow, the processing power required for computations also increases in a cubic manner. Amongst all arithmetic operations, multiplication is the most intensive and expensive operation. The Strassen's algorithm was proposed by Volker Strassen in the 1960s in an attempt to make dense matrix multiplication more efficient (i.e. entails less multiplication operations). This algorithm has been proven to be more efficient than regular multiplication and has an efficiency of approximately O($n^{2.8}$). Although many more efficient algorithms for dense matrix multiplication have been proposed at recent times, we will focus on implementing the Strassen's algorithm for this assignment.

2 Strassen's Algorithm

Following are assumptions made about the two matrices (A and B) which are operands of the dense matrix multiplication operation:

- Both are square matrices with size 2^n x 2^n (i.e. $A \in \mathbb{R}^{2^n \text{x} 2^n}$, $B \in \mathbb{R}^{2^n \text{x} 2^n}$)
- Almost all entries in both matrices are non-zero

Since both matrices A and B are square matrices with size 2^n x 2^n, these can be divided into four equal blocks as follows:

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} B = \begin{bmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{bmatrix}$$

Suppose that the size of A is 4 x 4. Then all sub-matrices $A_{0,0}, A_{0,1}, A_{1,0}, A_{1,1}$, are 2 by 2 equally sized blocks as illustrated in the following:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

$$A_{0,0} = \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}, A_{0,1} = \begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix}, A_{1,0} = \begin{bmatrix} a_{2,0} & a_{2,1} \\ a_{3,0} & a_{3,1} \end{bmatrix}, A_{1,1} = \begin{bmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{bmatrix}$$

Sub-matrices of B which are $B_{0,0}, B_{0,1}, B_{1,0}, B_{1,1}$, can also be computed in a similar manner. Multiplying matrices A and B results in matrix C.

$$A * B = C$$

$$\begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} * \begin{bmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{bmatrix} = \begin{bmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{bmatrix}$$

Following are the set of computations performed when regular matrix multiplication is used to compute C:

$$C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0}$$

$$C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1}$$

$$C_{1,0} = A_{1,0}B_{0,0} + A_{1,1}B_{1,0}$$

$$C_{1,1} = A_{1,0}B_{0,1} + A_{1,1}B_{1,1}$$

As you can observe from the above, the computation of C via regular multiplication has required 8 multiplications for each matrix block. With the Strassen's algorithm, only 7 multiplications will be required. In an implementation of the Strassen's algorithm, it is necessary to first compute 7 matrices M_0 to M_6 as follows:

$$\begin{aligned}
M_0 &= (A_{0,0} + A_{1,1})(B_{0,0} + B_{1,1})\\
M_1 &= (A_{1,0} + A_{1,1})B_{0,0}\\
M_2 &= A_{0,0}(B_{0,1} - B_{1,1})\\
M_3 &= A_{1,1}(B_{1,0} - B_{0,0})\\
M_4 &= (A_{0,0} + A_{0,1})B_{1,1}\\
M_5 &= (A_{1,0} - A_{0,0})(B_{0,0} + B_{0,1})\\
M_6 &= (A_{0,1} - A_{1,1})(B_{1,0} + B_{1,1})
\end{aligned}$$

As you can observe from the above, a total of 7 matrix multiplication operations are required to compute the above set of matrices. These matrices can be combined to obtain the resultant matrix C via addition and subtraction operations as follows:

$$\begin{aligned}
C_{0,0} &= M_0 + M_3 - M_4 + M_6\\
C_{0,1} &= M_2 + M_4\\
C_{1,0} &= M_1 + M_3\\
C_{1,1} &= M_0 - M_1 + M_2 + M_5
\end{aligned}$$

It is clear from the above that although fewer multiplication operations are required for the Strassen's algorithm in comparison to the regular multiplication method, more addition and subtraction operations are necessary. However, since multiplication operations are more expensive than addition or subtraction, the Strassen's algorithm results in being more efficient than regular matrix multiplication.

The Strassen's algorithm outlined above can be implemented recursively. Multiplications that are required to compute matrices $M_0 \dots M_6$ are also dense matrix multiplications. In order to compute these matrices, the dense multiplication function can be called recursively. At each recursive call, the size of the matrices that are multiplied is halved. For instance, suppose that the size of A and B is n x n. Matrix M_0 is computed by multiplying matrices that are half the size of the original set of matrices passed into the recursive function (i.e. size is $\frac{n}{2}$ x $\frac{n}{2}$). The original problem is therefore halved and this simpler sub-problem is of the same kind as the original multiplication problem. The base case of this multiplication problem occurs when $n = 1$. At this point, two numbers (not matrices) are multiplied. Hence, the Strassen's algorithm is a perfect candidate for recursive implementation!

3 Implementation of the Strassen's Algorithm

In order to implement the Strassen's algorithm, you are required to expand and implement the following functions:

- `void denseMatrixMult(int ** A, int ** B, int *** resultMatrix, int n)`
- `int **sum(int ** A, int ** B, int x1, int y1, int x2, int y2, int n)`
- `int **sub(int **A, int **B, int x1, int y1, int x2, int y2, int n)`

- `void initMatrix(int ***mat,int n)`
- `int **readMatrix(char * filename)`
- `void freeMatrix(int n, int ** matrix)`
- `void printMatrix(int n, int ** A)`

The first function is the recursive function that implements the Strassen's algorithm (i.e. `void denseMatrixMult(int ** A, int ** B, int *** resultMatrix, int n)`). All other functions serve as helper functions for the recursive function. The first argument to `denseMatrixMult` is `A` which is a double integer pointer representing one dense matrix. `B` is a double integer pointer representing the second dense matrix. `resultMatrix` is a triple pointer that points to the matrix multiplication results (i.e. pointer to C). `n` is the size of matrices A and B. This recursive function does not return any value and therefore is of type `void`.

In order to compute matrices $M_0 \ldots M_6$, you will use the `sum` and `sub` functions. For instance, to compute M_0, it is necessary to compute the sum of sub-matrices $A_{0,0} + A_{1,1}$ and $B_{0,0} + B_{1,1}$. To compute the first term $A_{0,0} + A_{1,1}$, you can call the function `sum(A, A, 0, 0, n/2, n/2, n/2)` where n is the size of A. Similarly to compute the second term $B_{0,0} + B_{1,1}$, you can call the function `sum(B, B, 0, 0, n/2, n/2, n/2)`. Results from these two summations are multiplied via a recursive call to the original function. All remaining matrices $M_1 \ldots M_6$ can be computed in a similar manner.

After matrices $M_0 \ldots M_6$ are computed, these can be combined to obtain $C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}$. This requires two nested loops (to add columns and rows of matrices). Ensure that all matrices dynamically allocated are freed using the `freeMatrix` function within each recursive call (remember the rule of thumb: one call to `free` for every call to `malloc`). Remember that all matrices returned by the `sum` and `sub` functions are dynamically allocated, the resultant matrix `C` which is assigned to `resultMatrix` is also dynamically allocated and all matrices $M_0 \ldots M_6$ are also dynamically allocated. All of these matrices must be carefully freed.

4 Materials Provided

You will download contents in the folder `Assignment 2` which contains two folders (`code` and `expOutput`) onto your ECF workspace. Folder `code` contains a skeleton of function implementations and declarations. Your task is to expand these functions appropriately in `assignment2.c` and include necessary libraries in `assignment2.h` as required for implementation. `main.c` evokes all the functions you will have implemented and is similar to the file that will be used to test your implementations. Use `main.c` file to test all your implementations. `matrix1.txt` and `matrix2.txt` are sample files containing data that will be read by your program. Folder `expOutput` contains outputs expected for the supplied data files `matrix1.txt` and `matrix2.txt`. Note that we will **NOT** use the same sample files for grading your assignment. Do **NOT** change the name of these files or functions.

5 Division of Assignment into Parts

Part 1: Initializing, Freeing and Printing Matrices

In this part, function implementations of `initMatrix`, `freeMatrix` and `printMatrix` in `assignment2.c` file will be tested. In the `initMatrix` function, a 2D array of size `n` will be dynamically allocated. The memory address stored in the double pointer resulting from this allocation will be copied into the variable pointed to by triple pointer `mat`. `freeMatrix` frees a 2D dynamically allocated array of size `n` represented by the double pointer `matrix`. `printMatrix` prints the content of the 2D array represented by the double pointer `A`. `MATSIZE` is set to be 8 and this value may be changed during the evaluation of this assignment. This part will be tested by executing the program with the commands:

- `./run 1` and
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 1`

The output resulting from this part must match the contents of file `Part1.txt`.

Part 2: Reading Matrix from Data File and Printing to the Console

In this part, function implementation of `readMatrix` in `assignment2.c` file will be tested. Refer to the `a2Prep.pdf` file for guidance on how to read files in C. The `readMatrix` function will read the contents of a data file called `filename` and store the contents of this file in a dynamically allocated 2D array which represents a square matrix of size `MATSIZE`. The double pointer representing this 2D array will be returned by this function. This part will be tested by executing the program with the commands:

- `./run 2`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 2`

The output resulting from this part must match the contents of the file `Part2.txt`.

Part 3: Adding and Subtracting Sub-Matrices

Here you will complete the implementation of two functions: `sum` and `sub`. Suppose that there are two square matrices A and B of the same size. Suppose that these matrices are of size 8 (this can change). The function `sum` will perform a matrix addition of two sub-matrices (one from matrix A and the other from matrix B) of size n and return the pointer to this new dynamically allocated matrix (i.e. double pointer). The sub-matrices are identified by coordinates (x_a, y_a) and (x_b, y_b) which denote the indices at which a sub-matrix begins in the main matrix. The arguments passed to these functions are `A, B,` x_a, y_a, x_b, y_b, `n` where `A` and `B` are double pointers to the main matrices. This part of the assignment will be tested via the following commands:

- `./run 3 1`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 3 1`
- `./run 3 2`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 3 2`

Outputs from these tests must match contents in `Part3_1.txt` and `Part3_2.txt` files resulting from the provided `matrix1.txt` and `matrix2.txt` files.

Part 4: Recursive Dense Matrix Multiplication

In this part, you will implement the Strassen's algorithm introduced in Section 2 by expanding the function `denseMatrixMult`. To this function, double pointers `matrix1` and `matrix2` representing two dense matrices, a triple pointer that will point to the double pointer representing the resultant matrix and `n` which is the size of these square matrices are passed as arguments. This function does not return any values as it is storing the results indirectly via the triple pointer which points to a double pointer which has been declared outside the scope of this function. This part of the assignment will be tested via the following commands:

- `./run 4`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 4`

Outputs from these tests must match the contents of `Part4.txt` which is the result of multiplying matrices obtained from the provided `matrix1.txt` and `matrix2.txt` files.

6 Grading: Final Mark Composition

It is **IMPORTANT** that you follow all instructions provided in this assignment very closely. Otherwise, you will lose a *significant* amount of marks as this assignment is auto-marked and relies heavily on you accurately following the provided instructions. Following is the mark composition for this assignment (total of 40 points):

- Successful compilation of all program files i.e. the following command results in no errors (3 points):
  ```
  gcc assignment2.c main.c -o run
  ```
- Successful execution of the following commands: (5 points)
  ```
  valgrind --quiet --leak-check=full --track-origins=yes ./run 1
  valgrind --quiet --leak-check=full --track-origins=yes ./run 2
  valgrind --quiet --leak-check=full --track-origins=yes ./run 3 1
  valgrind --quiet --leak-check=full --track-origins=yes ./run 3 2
  valgrind --quiet --leak-check=full --track-origins=yes ./run 4
  ```
- Output from Part 1, 2, 3a and 3b exactly matches expected output (2 points each for a total of 8 points)
- Output from Part 4 exactly matches expected output (14 points)
- Code content (10 points)

Sample expected outputs are provided in folder `expOutput`. We will test your program with a set of completely different data files. Late submissions will not be accepted.

7 Code Submission

- Log onto your ECF account
- Ensure that your completed code compiles
- Browse into the directory that contains your completed code (`assignment2.h`, `assignment2.c`)
- Submit by issuing the command:
  ```
  submitcsc190s 3 assignment2.h assignment2.c
  ```

ENSURE that your work satisfies the following checklist:

- You submit before the deadline
- All files and functions retain the same original names