INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 2

APAR AHUJA | ENTRY NO. 2019CS10465

Course - COL380 | Prof. Subodh Kumar

February 14, 2023

Contents

Chapter 1

Summary

1.1 Approach 1:

Idea: Read matrix from input file and store as vector of vector of blocks, `rowBlocks`. Create tasks to sort rows of `rowBlocks`. Square the matrix by creating tasks on the outermost for-loop that traverses `rowBlocks` and store the answer in a result vector. Write the result vector to the output file.

Why attempted: To establish a baseline for correctness and performance without any complicated pipeline.

Results: 79488ms (read + square + write) with $n = 2000000$, $m = 100$, $k = 2^{15}$, 8 cores/threads.

Drawbacks: Memory required is high due to the result matrix taking extra memory.

1.2 Approach 2:

Idea: Read and write stay the same as in approach 1. Square the matrix by creating tasks on the middle for-loop that traverses `rowBlocks` and store the answer in a result vector.

Why attempted: To reduce task size but increase their quantity.

Results: 209679ms (read + square + write) with $n = 2000000$, $m = 100$, $k = 2^{15}$, 8 cores/threads. This is 2.6x slower than approach 1, with a speedup of 0.38x.

Drawbacks: Memory required is still high. Overhead for task creation and maintaining the task queue is also high.

1.3 Approach 3:

Idea: Read and write stay the same as in approach 1. Square the matrix by creating tasks on the outermost for-loop that multiplies the block matrices and store the answer in a result vector.

Why attempted: To test the trade-off between number of tasks vs time taken by each task, and to speed up the bottleneck caused by the block matrix multiplication.

Results: 149336ms (read + square + write) with $n = 2000000$, $m = 100$, $k = 2^{15}$, 8 cores/threads. This is 1.9x slower than approach 1, with a speedup of 0.53x.

Drawbacks: Memory required is still high. Overhead for task creation and maintaining the task queue is lower than in approach 2, but the time taken is still higher than in approach 1.

1.4 Approach 4:

Idea: Read and write stay the same as in approach 1. Square the matrix by creating tasks on both the outermost for-loop that traverses `rowBlocks` and the outermost for-loop that multiplies the block matrices. There was also a `omp` critical section to count the number of non-zero blocks in the output in previous approaches. I figured removing it from the loops might speed up the code.

Why attempted: To mix the advantages of approach 1 and approach 3 and remove the omp critical bottleneck.

Results: 73014ms (read + square + write) with $n = 2000000$, $m = 100$, $k = 2^{15}$, 8 cores/threads. This is 1.1x faster than approach 1, with a speedup of 1.1x.

Drawbacks: Memory required is still high due to the result matrix, but this can be mitigated by writing in parallel as soon as the blocks are ready.

1.5 Final Approach:

Idea: Read stays the same as approach 1. We square the matrix by creating tasks on the outermost for-loop (approach4.cpp: line 247) that traverses rowBlocks. We create a child-task to write the block as soon as the result is ready.

Why attempted: To reduce memory usage by not storing the intermediate result matrix.

Results: 72212ms (read + square + write) with n = 2000000, m = 100, k = 2^{15}, 8 cores/threads. This is 1.1x faster than approach 1, with a speedup of 1.1x.

Drawbacks: The memory required is much lower than in the previous approaches, but there is still some overhead for task creation and maintaining the task queue. The block matrix multiplication algorithm is naive. We can implement strassen or other cache efficient block based multiplication schemes. This will speed up for larger values of m, however for small m this might not be optimal due to overhead costs of other algorithms.

1.6 Code Description:

1. Block: Used to store a block of a sparse matrix. The Block structure has three members: `row` and `col` are integers that store the row and column indices of the block, respectively. `data` is a vector of integers that stores the non-zero blocks.

2. readMatrix: This function reads a sparse matrix stored in a binary file. It takes the filename, the number of rows, the block size, and a vector of blocks as input. It also sets a flag indicating if the input matrix is in out-matrix-format.

3. compareColumn: This function compares two blocks based on their column index. It is used in the readMatrix function to sort the blocks in each row based on their column index.

4. writeMatrix: This function writes a block sparse matrix to a binary file. It takes the filename, the number of rows, the number of columns, the number of blocks, a vector of blocks, and a flag indicating if it is writing an out-matrix-format matrix.

5. matrix_mult: This function computes the result of a block-matrix multiplication. It takes the output vector, two input vectors, and the block size as input.

6. squareMatrix: This function computes the square of a block sparse matrix. It takes the input matrix, the output matrix, the number of rows, the block size, and the number of blocks as input. It uses OpenMP to parallelize computation.

7. squareMatrixWrapper: This function reads a matrix from an input file, computes its square matrix, and writes the resulting matrix to an output file. It also measures the time taken to perform these operations and prints it to the console.

8. compareOutputFiles: This function takes as input two filenames representing binary files that store information about two sparse matrices, and then reads and compares the data in these files. The function first reads and compares the `n`, `m`, and `k` values, which specify the number of rows and columns of the matrices, and the number of blocks in the block representation of the matrices. If any of these values are different in the two files, the function prints an error message and returns. If the values are the same, the function proceeds to read the block representation of the matrices from the files and stores them in row-wise order in two vectors of vectors called `rowBlocks1` and `rowBlocks2`. Finally, the function calls the `compareMatrices` function to compare the two matrices stored in `rowBlocks1` and `rowBlocks2`. If the matrices are equal, the function prints a success message.

9. compareMatrices: This function takes in two matrices represented as vectors of vectors of Block objects, and compares them element-wise. It first checks if the size of the two matrices is equal, and returns an error message if they differ in size. Then, for each element of each matrix, it checks if the `row`, `col`, and `data` values are equal. If the matrices differ, it prints out an error message indicating the row and block number where the difference occurred, as well as the values of the corresponding blocks in both matrices. If the function completes without encountering a difference between the two matrices, it prints out a success message.

1.7 Final Scalability Analysis:

n = 2000000, m = 100

Non-0 input blocks	Non-0 output blocks	2 cores	4 cores	8 cores	16 cores
1024	2048	4117.24	2050.07	1183.04	1070.89
32768	71414	57931.88	29014.51	15812.32	12235.75

Table 1.1: Performance comparison of block matrix multiplication on different number of cores