Aristotle University of Thessaloniki

Electrical and Computer Engineering Department

Parallel and Distributed Systems

4th Assignment

Sparse Boolean Matrix Multiplication with

two levels of parallelization

Stavros Malakoudis

Email: samlako@ece.auth.gr

AEM: 9368

Antonios Ourdas

Email: ourdasav@ece.auth.gr

AEM: 9358

Source of code of our project can be found at https://github.com/Stavrenas/PDS_ex4

Introduction

In this final assignment of Parallel and Distributed Systems course we try to implement a fast sparse Boolean matrix multiplication (BMM) method using two forms of parallelism. In our implementation we used open MPI combined with OpenMP.

BMM has multiple applications in science and engineering problems. A couple of them are described below:

- **Triangle counting in unweighted graphs:** A graph with n nodes is often represented by an $n \times n$ adjacency matrix A with each element a_{ij} indicating the weight of the vertex connecting node i with node j. In many cases there are networks (e.g., social networks) where we are interested only in the existence of a connection between 2 nodes. Therefore, each vertex has a weight which can take only two values 1 and 0. The adjacency matrix of such graphs can be thought as a Boolean matrix. The total number of triangles of a graph (i.e., number of mutual connections between two nodes) can be calculated using the following formula:

  $$c = \big(A \odot (AA)\big)\frac{e}{2}$$

  where $\odot$ is an element-wise operation and e is a $n \times 1$ vector containing only 1 values. The above multiplication $(A \odot (AA))$ can be implemented as a masked boolean matrix multiplication using logical instead of arithmetic operators, since all values are either 1 or 0.

- **Transitive closure of a graph:** In graph theory the transitive closure detects whether there is way to get from one node to another following one or more edges. That is, if there is a path between two nodes. Typically, a transitive closure is stored on a matrix $M_{n \times n}$, where each element m_{ij} is either 1 or 0 indicating the existence of a path from node i to node j. Given a directed graph G and its adjacency matrix A, the transitive closure of G (denoted as G^+) can be found using the following formula:

  $$G^+ = G^1 \cup G^2 \cup \ldots \cup G^n$$
  $$A^+ = A^1 + A^2 + \cdots + A^n$$

  The above formula includes Boolean matrix multiplication and addition to estimate factors $A^k, k = 1, \ldots, n$ and then add them all together. If G has a small number of vertices, then its adjacency matrix A is sparse and thus boolean operations for sparse matrices can be used.

Algorithms description

To handle sparse and blocked matrices more efficiently we use the following structs:

- **Matrix:** stores a sparse matrix in either CSR or CSC format.
- **BlockedMatrix:** stores a blocked matrix as a list of Matrix structs.

Our implementation for simple matrix multiplication makes us of the following formula:

$$c_{ij} = \bigvee_{k=1}^{K} a_{ik} \wedge b_{kj}$$

Each element c_{ij} is calculated by checking whether row A_i and column B_j have any common elements by checking row and column indices of their elements inside sparse matrix

representation format (csr/csc). This process is done in parallel calculating multiples rows simultaneously using OpenMP. The second form of parallelization is achieved by blocking matrices and performing block-wise multiplications combined with MPI. First, we block both matrices to be multiplied into $n_b \times n_b$ blocks of size $b \times b$ using csr/csc format. We then perform block-wise multiplication to calculate each block C_{pq} using the following formula:

$$C_{pq} = \bigvee_{s=1}^{K} A_{ps} B_{sq}$$

Each product $A_{ps}B_{sq}$ is calculated using parallel matrix multiplication as described above.

We use MPI to distribute calculations among may processes. Given n MPI processes, each process calculates every n-th row of blocks (i.e., process i will calculate blocks C_{i1} to C_{in_b} for $i = rank : n : n_b$). If the number of block-rows isn't divisible by n, then the remaining block-rows are assigned to the first processes. Finally, all other processes send data to one process (with rank 0) and blocks are merged (by adding all of them together) into the final matrix C.

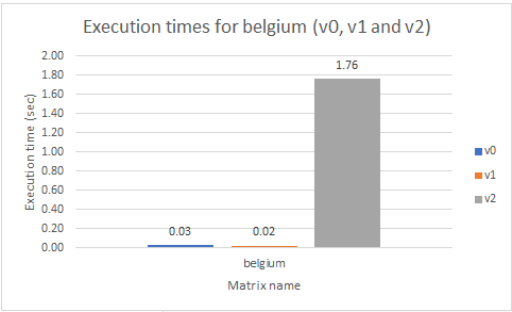
The above calculations are performed using a filter F which is multiplied elementwise with the product $F \odot (AB)$ and indicates which elements of the final product we want to calculate.

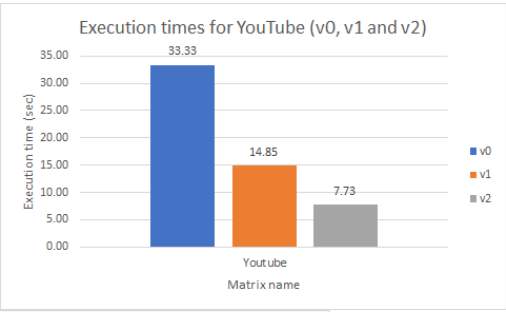
Each level of parallelization was introduced gradually using four code versions:

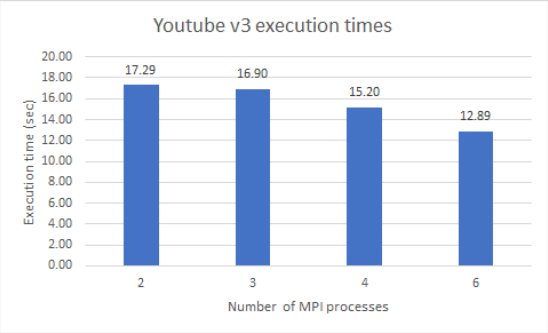
- **v0:** Simple matrix multiplication without blocking or parallelization.
- **v1:** Simple matrix multiplication without blocking using only openMP.
- **v2:** Matrix multiplication with blocking using only openMP.
- **v3:** Matrix multiplication using blocking, openMP and MPI parallelization.

Results

We tested our code with matrices taken from https://sparse.tamu.edu/. All tests were conducted on the HPC facility of AUTH. Execution times are reported below:







Blocksize VS Time , Youtube (n = 1134890)

Blocksize	**10000**	**25000**	**50000**	**75000**	**100000**	**200000**	**500000**
Time (s)	>30	20.12	7.75	7.29	8.05	10.38	13.64
RAM	>16GB	5.5GB	1.5GB	700MB	250MB	250MB	100MB

Blocksize VS Time , NACA (n = 1039183)

Blocksize	**10000**	**25000**	**50000**	**75000**	**100000**	**200000**	**500000**
Time (s)	>30	6.57	2.14	1.26	1.05	0.53	0.35
RAM	>16GB	4.3GB	1.4GB	450MB	400MB	200MB	200MB

We observe the following:

- For some matrices MPI implementation does not achieve proper scaling as the number of processes increases. Furthermore, we tested our code to run on multiple nodes and the results were similar (and sometimes worse) compared to deploying MPI processes on one computing node using multiple CPUs and tasks.
- Optimal block size varies across different matrices and depends on many factors:
    - One is matrix size. Selecting a small block size obviously results in a big number of blocks causing memory management issues. We made the following selection based on matrix size N:

$$block\_size = \begin{cases} \sqrt{N}, if\ N < 100000 \\ 50000, if\ 10000 \leq N < 1000000 \\ 100000 \end{cases}$$

    - Another one is how elements are distributed on each matrix. This has an impact on the amount of non-zero blocks created and the density of each block thus affecting computation time and memory usage.

References

[1] https://en.wikipedia.org/wiki/Transitive_closure

[2] https://www.ics.uci.edu/~irani/w15-6B/BoardNotes/MatrixMultiplication.pdf