Parallel and Distributed Systems - Assignment 1: Sparse matrices

Aristotle University Thessaloniki - Electrical and Computer Engineering

Authored by:

Antonios Antoniou - 9482 - aantonii@ece.auth.gr

Efthymios Grigorakis - 9694 - eegrigor@ece.auth.gr

The goal of this assignment

The first assignment of the Parallel and Distributed Systems course requires us to read an `.mtx` file, depiciting a square, symmetric, undirected, non-weighted graph, in the form of a triangular matrix, whose complete set of values has to be determined by the programmer. For this table, we have to calculate how many triangles (i.e subgraphs with 3 vertices and 3 edges forming a closed structure) each vertex is a member of. **We will try to keep this report as short as possible, since you can find the full codebase on GitHub**

This is realized by calculating the $H = A \odot A^2$ matrix (where $\odot$ denotes the Hadamard, element-wise multiplication of two matrices). Afterwards, we calculate the $C = (H \cdot e)/2$ vector (where e is a vector filled with ones, the same size as A). Each element on the `C` matrix now represents how many triangles this vertex is in.

After reading the `.mtx` file, chances are the table we're dealing with is sparse. This essentially means that the amount of nonzero values is greatly outweighed by the total zero values in the matrix. For this, we use specialized formats, mainly:

- COO (coordinate list): an array of `(row, column, value)` tuples,
- CSR & CSC (Compressed Sparsed Row - Column): When it comes to the CSR format, it consists of 3 arrays, containing a value and its corresponding column index, along with the `row index` array that dictates how many nonzero elements lie before the current row. The same holds for the CSC format, but column-wise. It is obvious that the matrices addressed in this assignment produce the same result either way, due to symmetry.

More info on sparse matrices can be found **on Wikipedia**

Preferred format and data quirks utilized

The format our team decided to use was CSR. As has been already mentioned, the CSR and CSC formats are equivalent in the case of **square, symmetric tables**. This means that we were able to replicate the matrix to matrix multiplications by calculating the dot product between the i-th row and the j-th **row** (instead of the j-th **column**), to produce the $B_{i,j}$ element of the $B = A^2$ matrix.

Calculating the C matrix

Instead of calculating the square dot product of A first, and then plugging it into the Hadamard operation, it was **less time and memory consuming** to only calculate the values of $A^2_{i,j}$ that

would then be multiplied with a **nonzero value**. This meant that we only made any operation for positions that coincided with the nonzero values of the original table, since A^2 was very likely to become fairly dense.

This method was used througout every algorithm, serial or not. The core difference between the two was the workload, that would be distributed between the threads while implementing the latter. Since each row in a CSR structure is completely independent of any other row, each thread was assigned a **subset of the rows of the original table**. The way the subsets would be distributed was found to save more time when it took nonzeros into account. Essentially, each thread had a sub-array with -almost- equal number of nonzero values, regardeless of how many rows it consisted of. Loosely, that could ensure a comparable amount of operations executed by each thread, which is exactly what we want when implementing a parallel algorithm.

Testing and measuring execution times

After finalizing the sequential algorithm and the parallel ones, using `pthread`, `openMP` and `openCilk`, it was time to find various data structures and run tests on them. Our team used five sparse tables:

1. **DIMACS10/belgium_osm**
2. **LAW/dblp-2010**
3. **DIMACS10/NACA0015**
4. **mycielskian13**
5. **snap/com-Youtube**

Each matrix was used once for the sequential algorithm, while the parallel algorithms were tested for **2, 4 and 8 threads**. Each test consisted of 12 repetitions, the first two of which were left out of the data measurement, to avoid the appearance of outliers. Then, the mean time was taken and added to the `data.csv` file, with info about the algorithm used and the number of threads, if any. Below is the resulting table with values converted to `ms`.

	library_threads	belgium_osm	dblp2010	NACA0015	mycielskian13	comYoutube
1	"seq"	145.228	556.915	1602.79	8392.35	26261.5
2	"pth2"	72.792	158.947	617.177	1626.4	8279.06
3	"pth4"	51.953	98.29	358.159	1005.21	6672.24
4	"pth8"	48.292	79.41	256.984	707.644	4941.19
5	"omp2"	73.077	158.168	578.438	1916.33	9448.68
6	"omp4"	49.048	106.31	339.67	1057.29	7567.76
7	"omp8"	34.904	87.354	254.788	799.552	5976.21
8	"cilk2"	65.193	151.117	558.098	1225.27	6651.93
9	"cilk4"	54.023	99.319	370.289	762.743	5140.38
10	"cilk8"	48.643	88.669	270.016	575.426	4414.42

```
df
```

The time measuring routine was handled by `Makefile`:

```
./openmp 1 2
./openmp 1 3
./openmp 1 4
./openmp 2 0
./openmp 2 1 # [...]
```

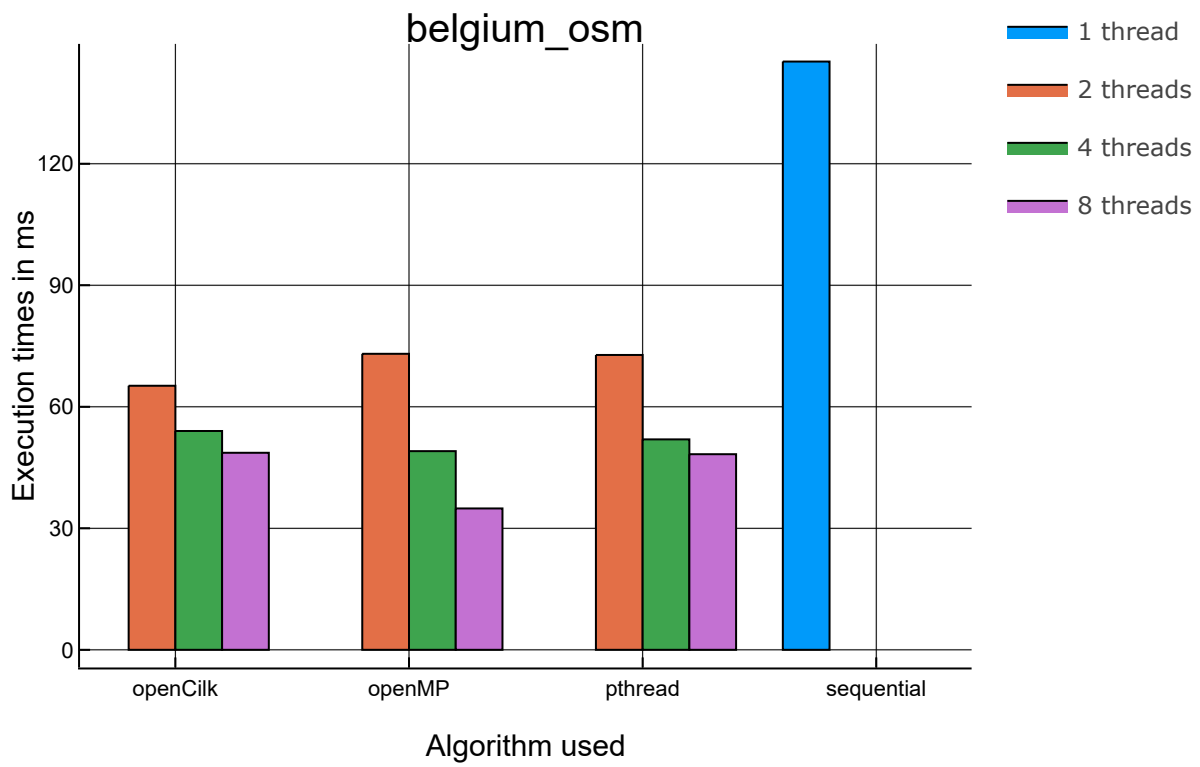
The `.c` files were given two arrays of filenames and number of threads, whose index was given to the terminal by the `measure_times` routine:

```
char *filenames[5] = {
    "tables/belgium_osm.mtx",
    "tables/dblp-2010.mtx",
    "tables/NACA0015.mtx",
    "tables/mycielskian13.mtx",
    "tables/com-Youtube.mtx"
};
```

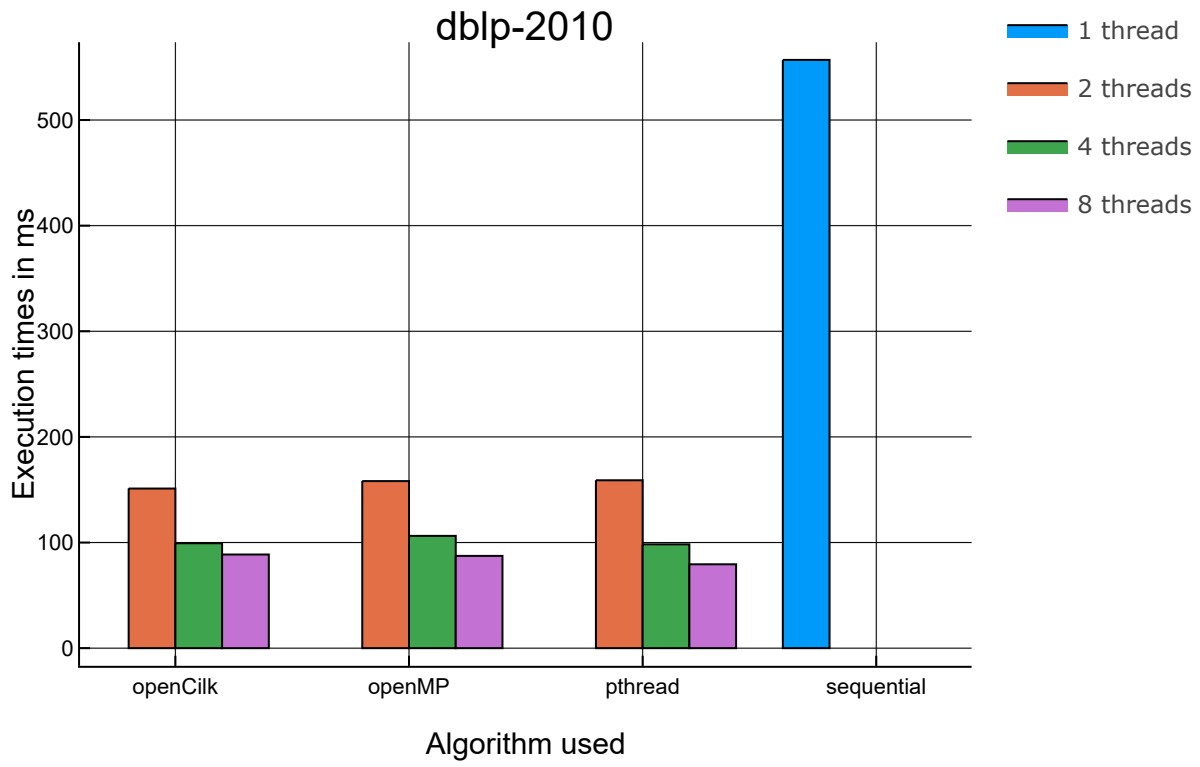
```
char *num_threads[3] = {2, 4, 8};
```

Each of the test was then written into the `data.csv` file. Below are the conclusions of the research process for each individual sparse matrix.

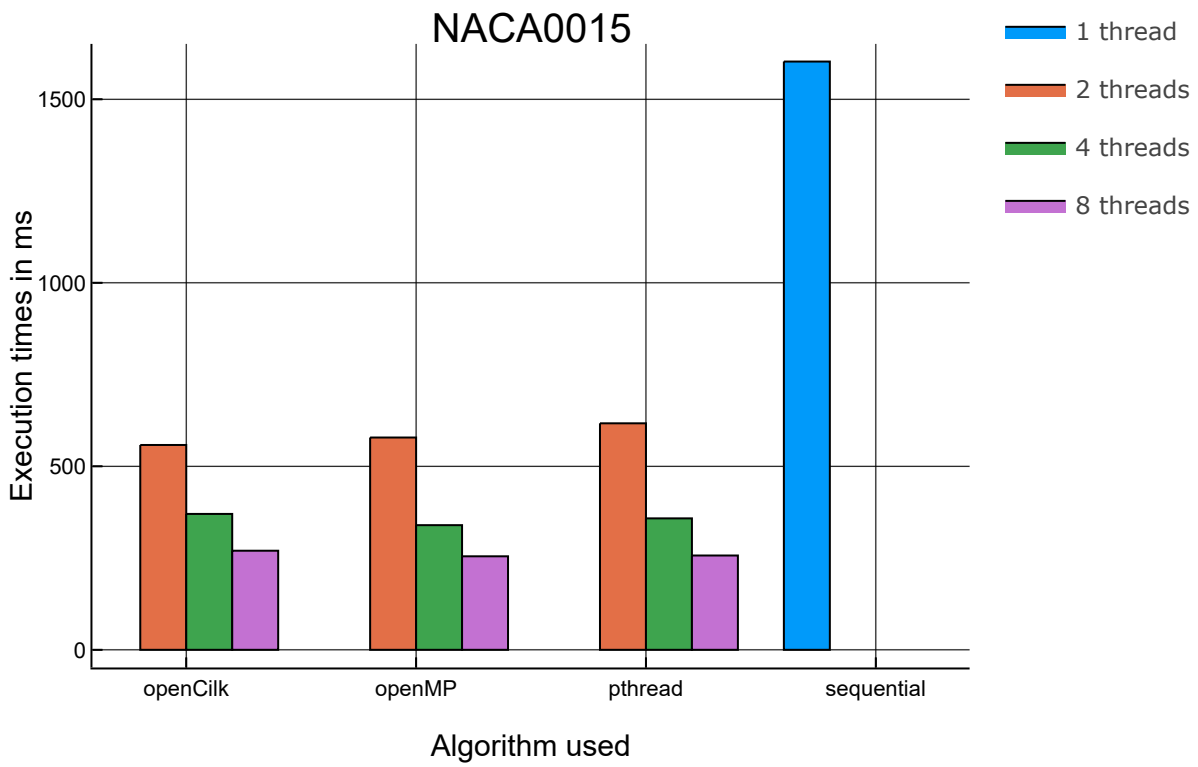
b1 =


belgium_osm
1 thread
2 threads
4 threads
8 threads
Execution times in ms
0
30
60
90
120
openCilk
openMP
pthread
sequential
Algorithm used

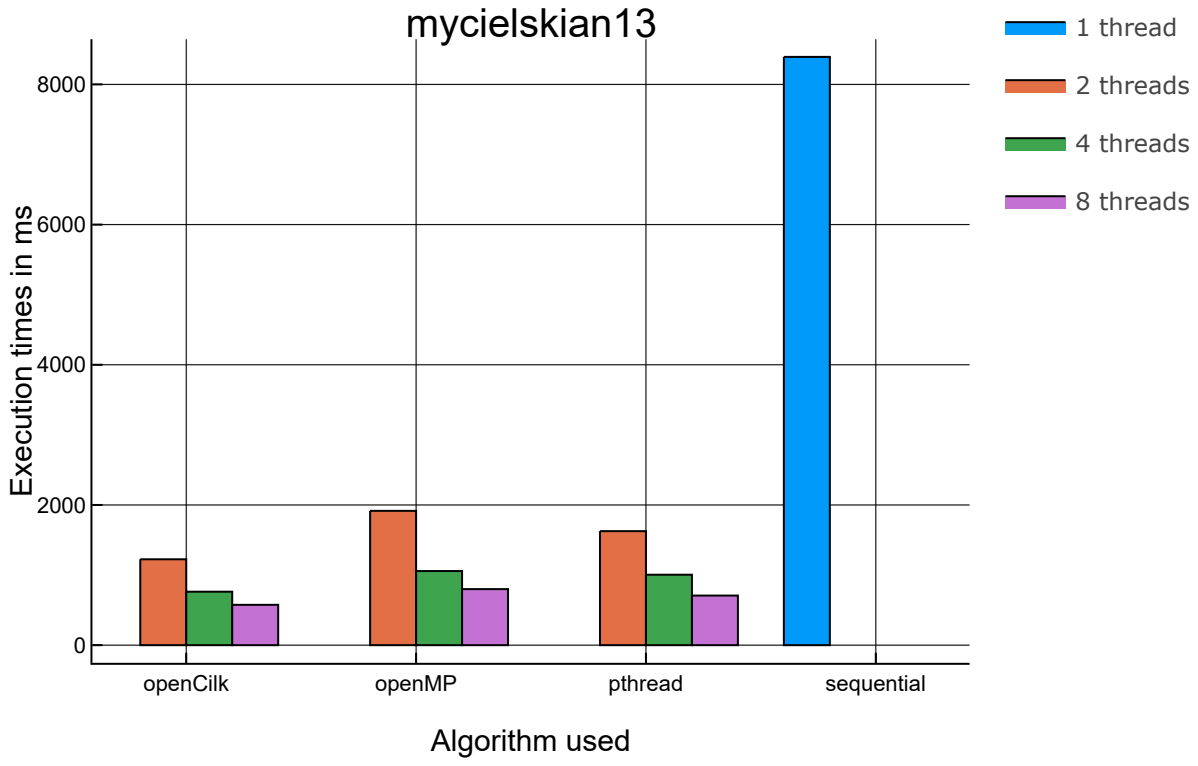
b2 =


dblp-2010
1 thread
2 threads
4 threads
8 threads
Execution times in ms
0
100
200
300
400
500
openCilk
openMP
pthread
sequential
Algorithm used

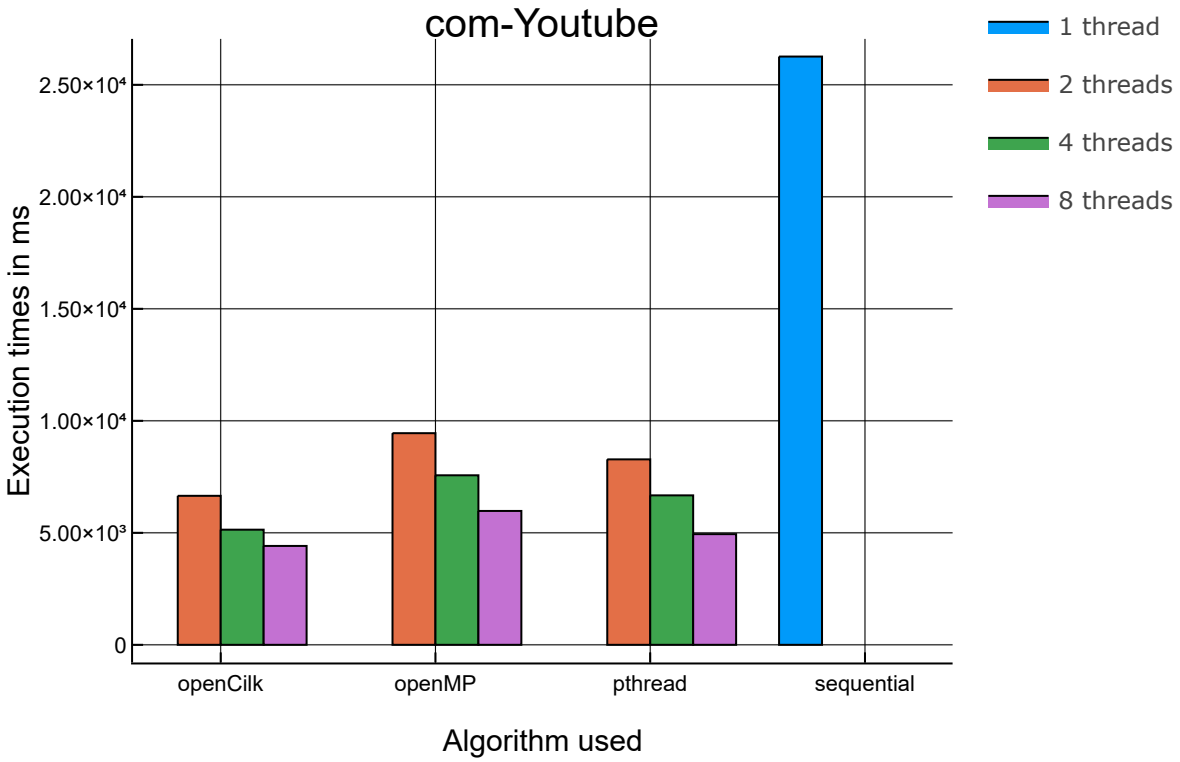
b3 =


NACA0015
1 thread
2 threads
4 threads
8 threads
Execution times in ms
0
500
1000
1500
openCilk
openMP
pthread
sequential
Algorithm used

b4 =


mycielskian13
1 thread
2 threads
4 threads
8 threads
Execution times in ms
0
2000
4000
6000
8000
openCilk
openMP
pthread
sequential
Algorithm used

b5 =


com-Youtube
Execution times in ms
2.50×10⁴
2.00×10⁴
1.50×10⁴
1.00×10⁴
5.00×10³
0
openCilk
openMP
pthread
sequential
Algorithm used
1 thread
2 threads
4 threads
8 threads