Parallelized Vertexwise Triangle Enumeration using Matrix Multiplication In OpenMP, OpenCilk, Pthreads

Filosidis Stavros, 9456

Parallel & Distributed Systems @ ECE AUTh

Abstract

This report focuses on implementation and benchmarking of two algorithms for vertex-wise triangle enumeration in undirected and unweighted sparse matrices. The first algorithm referred to as $V3$ is almost equivalent to the brute force method for the computation. The second algorithm, $V4$ implements a linear algebra method based on *Matrix Multiplication.* The GitHub link for the code can be found here.

1 Introduction

In Graph Theory, the **Triangle Graph** is planar undirected graph with 3 vertices and 3 edges. It is also known as the cycle graph C_3 and the complete graph K_3. The number of triangles in a graph is an important matrix metric, and the individual edges that partake in triangles an even more useful one. Even though counting is not very computationally intensive, vertex-wise counting *(triangle enumeration or listing)* generating the actual triangle edges, is a computationally expensive operation.

All examples use symmetric sparse matrices in *MatrixMarket (COO or Coordinate)* format and are converted to *CSR (Compressed Sparse Row)*. Even though the original matrices only provide the upper triangular part, in the scope of this assignment, the complete adjacency matrix **A** is used.

2 Naïve algorithm

The simplest algorithm implemented is an extension of the brute force dense matrix algorithm, adjusted for sparse matrices in the *CSR* format for efficiency, using binary search for improved performance. *BinarySearch* returns 1 if there is an edge between (i, j), and 0 otherwise.

If only the upper triangular part was used, we would need to update all $c[i], c[j], c[k]$. However, accessing 3 different parts of an array requires a significant amount of locks and leads to worse performance when parallelized. Using the complete adjacency matrix solves this problem and allows for a single index access to each assigned thread, as seen in algorithm 1.

Algorithm 1 Brute force sparse matrix

```
1: procedure TRIANGLEENUMV3($A$)
2:     $c3 \leftarrow [0] * N$
3:     for $i$ in N - 2 do
4:         for $j$ in neighbors of $i$ do
5:             for $k$ from $j + 1$ in neighbors of $i$ do
6:                 $c3[i] \leftarrow c3[i] +$ BINARYSEARCH(j,k)
7:             end for
8:         end for
9:     end for
10: end procedure
```

3 Masked Matrix Multiplication

This algorithm utilizes a very interesting property of dense matrices. The formula for calculating the final *1-dimensional array c* containing the triangle vertices is as follows:

$$c_3 = (A \odot (A\,A))\; e/2 \tag{1}$$

where $\odot$ denotes the Hadamard or element-wise product and e is an n-length vector with all the elements initialized to 1. The result is divided by 2 because the full matrix is used in the calculations and each triangle is counted twice.

The square matrix product returns all node pairs (i, j) which are connected with paths of length 2. The element-wise multiplication essentially constitutes a *mask* to the generated NxN matrix A^2, and removes the elements that have no direct edge from i to j, since the elements $A[i][j] = 0$.

The algorithm takes advantage of the amount of zero elements of the sparce matrix combined with the *CSR* format, which allows for more efficient calculations. This is because only the *non-zero* elements are stored in memory, and computations can be performed by skipping the masking and multiplication steps. Furthermore, the row elements in *CSR* are sorted. This allows for a very efficient row-column multiplication method, by just counting the common elements between row_i and $column_j$ in $\mathcal{O}(n_i n_j)$ time.

4 Parallelism

The parallel implementation of algorithm 1 utilizes *OpenMP, OpenCilk* and algorithm 2 utilizes *OpenMP, OpenCilk and Pthreads*. All the code was written mostly in *C*.

A *shared memory* parallelism is used, by dynamically distributing each outer *for loop*, representing each matrix *row*, in each thread. For this problem, equal segmentation cannot be used because of the unequal calculations in each chunk. For

Algorithm 2 Masked matrix product

```
procedure TriangleEnumV4($A$)
    $c \leftarrow [0] * N$
    for $i$ in N do
        for $j$ adjacent to $i$ do
            $c[i] \leftarrow c[i] +$ MultiplyRowCol(i,j)
        end for
    end for
    $c3[i] \leftarrow c[i]/2$
end procedure

procedure MultiplyRowCol(row, col)
    $i \leftarrow row_0$
    $j \leftarrow col_0$
    while $i < row_{len}$ and $j < col_{len}$ do
        if $row_i == col_j$ then
            $c[row] \leftarrow c[row] + 1$
        else if $row_i < col_j$ then
            $i \leftarrow i + 1$
        else
            $j \leftarrow j + 1$
        end if
    end while
end procedure
```

example, if there are 8 threads available and each thread is assigned a chunk of size $N/8$, and the matrix elements are accumulated in the upper left region, mostly the first threads will perform the majority of the computations, leading to minor parallelism and thus, performance improvements. A possible solution could be to sort all the rows based on the non-zero elements they contain, or find the most densely populated regions and properly perform the assignment for more optimized scheduling, but optimal scheduling is out of the scope of this assignment.

4.1 OpenMP

For parallelism with **OpenMP**, only the scheduling option was used, and *dynamic scheduling* was used in particular.

4.2 OpenCilk

No particular work was needed here, due to the usage of the complete adjacency matrix **A** in the *CSR* format. No locks or reducers were used.

4.3 Pthreads

Pthreads was also implemented using dynamic scheduling with chunk size of 1. Each time a thread is done calculating, it accesses a global counter which denotes the next *row* to calculate. The global variable is locked during the accesses, and this is the only *locking* performed in the entire assignment.

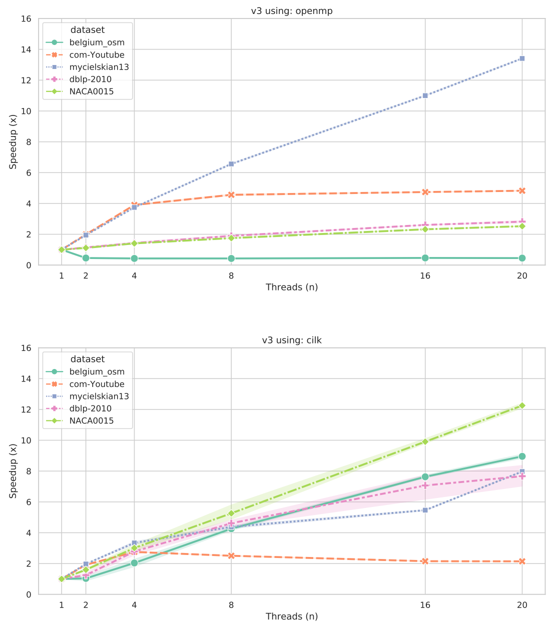

Figure 1: Speedups using algorithm 1.

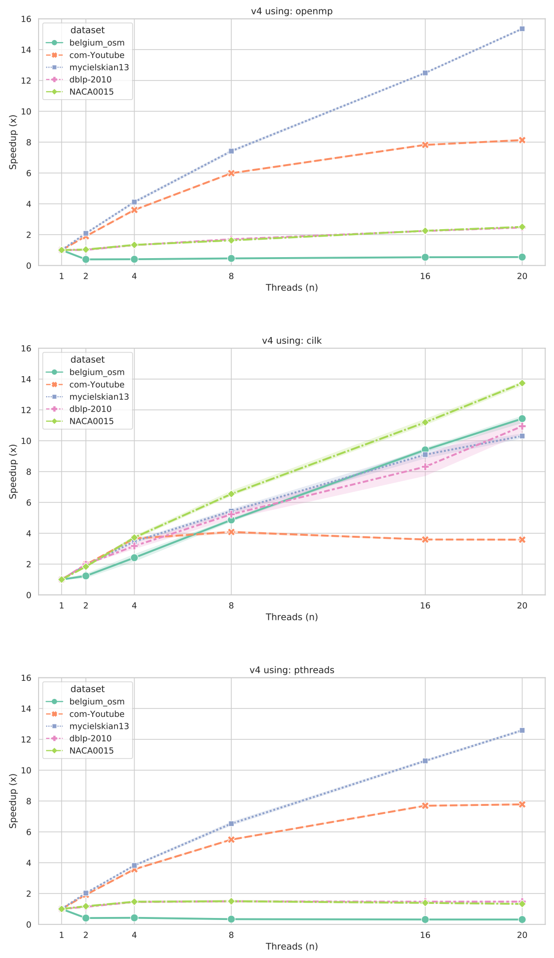

Figure 2: Speedups using algorithm 2.

Graph	Nodes (n)	Edges (m)	Triangles
belgium_osm	1,441,295	1,549,295	2420
com-Youtube	1,134,890	2,987,624	3,056,386
dblp-2010	326,186	807,700	1,676,652
mycielskian13	6,143	613,871	0
NACA0015	1,039,183	3,114,818	2,075,635

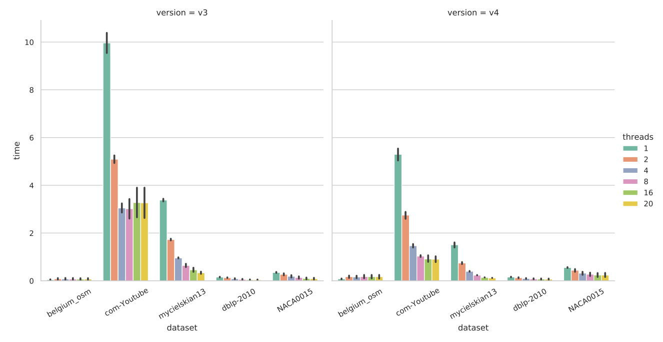

Figure 3: Runtime Comparison for the two versions.

5 Benchmarks & Results

Figures 1 and 2 demonstrate the acquired speedups for each experiment run on a HPC *(High Performance Computing)* node in our university. Each node is equipped with *20 physical CPU cores*, and has *hyperthreading* deactivated, thus the maximum amount of cores in the diagrams is 20.

Each experiment is executed 5 times, and the line plots represent the average speedup of each run. The baseline representing the initial $1x$ speedup is also the *mean* of each of the 5 runs for the corresponding experiment.

5.1 Analysis

From Figure 3, we can see that the theoretical complexity of Algorithm 2 over 1 is confirmed by the experiments. We can observe lower runtime, as well as better scalability when parallelized, beating 1 in almost every dataset.

5.1.1 OpenMP vs Cilk vs Pthreads

We will compare all the libraries used based on algorithm 2, which is the best performing one across the benchmarks both in scalability and runtime.

Mycielskian13 shows the best overall speedup across all libraries, probably due to its high density & uniformity, achieving better *scheduling* and less *cache misses*. We observe similar results with the **com-Youtube** dataset, which also does not contain densely populated areas.

OpenMP and **Pthreads** version seem to struggle with the **belgium_osm** dataset as seen in Figures 1 and 2, and underperform the serial counterparts. It is worth noting that this dataset is the largest in terms of vertices count and has unevenly distributed edges, with most of them located in the first rows. **OpenCilk** scales linearly with the particular dataset.

dblp-2010 and **NACA0015** produce similar results, having a very similar structure, with continuous regions of populated nodes.

As seen in Figure 4 **OpenCilk** beats the other libraries in 3/5 total datasets by a significant margin, while still competing in the other two. Scalability is also almost linear in all benchmarks except **com-Youtube**, as seen in Figure 2.

Overall what seems to differentiate **OpenCilk** from the other two networks is the scheduling, and it seems like it uses a wider chunk to split work between threads, reducing the need for reassignments when a thread is done working. We could further explore this statement by trying to use larger chunks when possible.

Our **Pthreads** implementation assigns a chunk of 1 each time, and it seems like that is also the case for **OpenMP** *dynamic scheduling* as well, since the performance is almost identical.

In general, **Pthreads** seem to be performing just as **OpenMP**, which is justified since OpenMP is based on Pthreads.

6 Further discussion

Further experimentation needs to be conducted to concisely conclude on the root cause for the performance abnormalities observed. Parameters such as cache misses, chunk size in scheduling, graph uniformity, complexity and size, need to be taken into account. Valgrind offers a great tool for profiling cache and branch prediction.

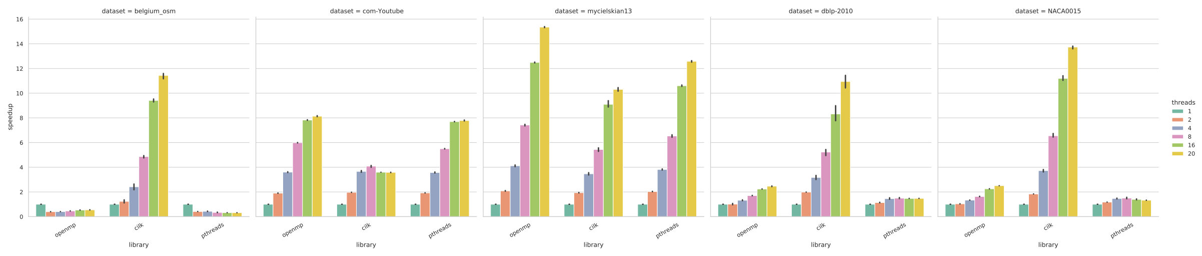

Figure 4: Performance for all datasets using V4 algorithm.