Παράλληλα και Διανεμημένα Συστήματα

[Εργασία 1 2020-2021] Υποβολή/Αξιολόγηση

Return to: Εργασίες και Τε...

Φάση υποβολής

Φάση εγκατάστασης	**Φάση υποβολής** Τρέχουσα φάση	Φάση αξιολόγησης	Φάση αποτίμησης βαθμολόγησης	Λήξη
	Υποβολή της εργασίας σας Ανοιχτό για υποβολές από Thursday, 12 November 2020, 11:00 PM (σήμερα) Καταληκτική ημερομηνία	Ανοιχτό για αξιολόγηση από Thursday, 3 December 2020, 12:00 AM (20 μέρες ακόμα) Προθεσμία αξιολόγησης: Thursday, 10		

Οδηγίες για την υποβολή

Exercise 1: Vertexwise triangle counting

We will implement, step-by-step, a shared memory implementation to count the number of triangles adjacent with each node in an undirected, unweighted graph $G(V, E)$, where $V = \{v_i\}_{i=1}^{n}$ is the set of n nodes and $E = \{(i, j), i \text{ connected to } j\}$ is the set of m edges among the nodes.

1 Triangle detection and counting

In graph theory, the triangle graph is the complete graph K_3, consisting of three vertices and three edges. Triangle counting has gained increased popularity in the fields of network and graph analysis. The application of triangle detection, location, and counting are multi-fold: detection of minimal cycles, graph-theoretic clustering techniques, recognition of median, claw-free, and line graphs, and test of automorphism.

2 Preliminaries

(V1). Assume that a graph is represented by its adjacency matrix `A[i][j]` where the rows and columns correspond to the vertices and `A[i][j] == 1` when there exists an edge between nodes `i` and `j` and `0` otherwise. In our studies we deal with undirected graphs so `A[i][j] == A[j][i]`.

Implement and run your first attempt to count triangles incident with each node as a triple loop that checks whether `A[i][j] == A[j][k] == A[k][i] == 1` and if so, increments the counts on all three nodes `c3[i]++; c3[j]++; c3[k]++;`

To test your code, generate random graphs, using any way you can come up. It is a bit tricky to generate random graphs with given properties but this is not the purpose of this project, any random graphs will do. See Erdős-Rényi random graphs for instance to get a better idea.

Implement and test your code. Persuade yourselves it is correct, otherwise we cannot go any further!

(V2). There is an obvious limitation in this first approach. We will find the same triangle 6 different times (all permutations of three vertices), so we can make our code 6 times faster by enumerating and checking only `i<j<k` triplets. Make this correction.

Measure the time it takes to run the two versions above. Use `clock_gettime` as described here.

(V3). So far so good, but now the data structure is really limiting our ability to process interesting graphs with millions of nodes and edges (think Facebook mutual friends, for instance).

To remedy this we will use a sparse matrix representation. For this assignment, we will use the most common sparse matrix storage scheme, the Compressed Sparse Columns (CSC) format. When the matrix is symmetric (as in our case), it is equivalent to the Compressed Sparse Row (CSR) format. Both `MATLAB` and `Julia` use the `CSC` format to store sparse matrices.

Now we can use graph datasets from the web. Make sure to download undirected, unweighted graphs. To load the sparse adjacency matrix use the Matrix Market (MM) format. Examples for parsing such files are found at https://math.nist.gov/MatrixMarket/mmio-c.html, e.g., `example_read.c` which relies on `mmio.h` and `mmio.c` for reading any MM matrix.

Rewrite your code as V3 to use the `CSC` storage scheme and only follow edges. Adjust your input to be able to use large graphs from the web ($n \approx 1{,}000{,}000$ and $m < 30{,}000{,}000$).

Parallelize your V3 code with OpenCilk and OpenMP, run it in parallel and compare sequential and parallel agreement in results and run times for different large datasets (we will propose which ones soon) and produce diagrams to study the scalability of your code.

3 Main part

We are ready to jump a bit further using the mathematical properties of the adjacency matrix A. The `[i][j]` element of the matrix-matrix product `A*A` is equal to the number of possible walks from node `i` to node `j` of length exactly 2, i.e. how many `k` exist that I can go from `i` to `k` and then from `k` to `j`. If there is also an edge `A[i][j]`, then that is the number of triangles we are after!

Formally, the number of triangles c_3 incident with each node is

$$c_3 = (A \odot (AA))\, e/2,$$

where $\odot$ denotes the Hadamard (or element-wise) product and e is a vector of length n where every element equals 1. We divide the result by 2 because we count both `ikj` and `ijk`.

3.1 Sequential implementation

3.1.1 Implement and verify the formula

Check the validity of the formula by implementing it in a high-level language, such as `MATLAB, Octave, Python, Julia` or whatever else you feel comfortable with.

3.1.2 Sparse matrix-vector product

Using the CSC storage scheme, implement in `C/C++` the sparse matrix-vector product $C = Av$, where v is a dense vector. Validate your implementation by comparing your results against multiple matrices A and vectors v.

3.1.3 Masked sparse matrix-matrix product

(V4) Implement a routine to compute the masked sparse matrix-matrix product $C = A \odot (AA)$, where A is a symmetric, sparse matrix. *Note*: Due to the Hadamard product with A the result will have the same sparsity as the input matrix A. You do not need to compute the values $C(i,j)$, where $A(i,j) = 0$. The number of triangles c_3 at each vertex node is $c_3 = Ce/2$. **Caution:** $A \times A$ will become dense, if there is even one node connected to most of the others. Make sure you use the masking to calculate **only** the elements of $A \times A$ at the nonzero positions of A.

3.2 Parallel implementation

Parallelize your masked sparse matrix-matrix product code (V4) with OpenCilk, OpenMP, and Pthreads run it in parallel and compare sequential and parallel agreement in results and run times for different large datasets (we will propose which ones soon) and produce diagrams to study the scalability of your code. Compare the triangle counting

using the masked sparse matrix-matrix product and the V3 code, implemented in the preliminary.

4 Report

- A 3-page report in PDF format (any pages after the 3rd one will not be taken into account). Report execution time of your implementations, in thoughtful and dense diagrams, with respect to the number of data points n and the number of edges m. Explain your implementation choices and code behavior.
- Upload the source code of `V3` and `V4` (sequential and parallel) on GitHub, BitBucket, Dropbox, Google Drive, etc. and add a link in your report.

Groups of maximum 2 persons are allowed. Both students must upload their report on the submission system. Both names must in appear in each report.

Created: 2020-11-12 Thu 13:32

Η υποβολή σας

Δεν έχετε υποβάλει την εργασία σας ακόμα

Έναρξη προετοιμασίας της υποβολής σας

◄ [Εργασία 1 2020-2021] Επιλογή ομάδας

Βαθμοί Σεπτεμβρίου 2019 ►

Return to: Εργασίες και Τε...