

## [2021-2022] Εργασία 4

# Parallel & Distributed Computer Systems (Jan 26, 2022) Exercise 4

## Bring Your Own Project

Propose a project of your own, or from another class or hobby of yours. But it has to be non-trivial and I have to agree to it.

## Help Us on Our Own Project

Help us port [FGLT](#) to Julia, using Base.Threads, [CUDA.jl](#) and [distributed computing](#) to demonstrate that pure Julia can be as good as or better than OpenCilk with MPI and CUDA.

Let us know if you are selecting this project, to coordinate the effort from the beginning.






**FGLT** is a C/C++ multi-threading library, for Fast Graphlet Transform of large, sparse, undirected networks/graphs. The graphlets in dictionary  $\Sigma_{16}$ , shown in Table 1, are used as encoding elements to capture topological connectivity quantitatively and transform a graph  $G=(V,E)$  into a  $|V| \times 16$  array of graphlet frequencies at all vertices. The 16-element vector at each vertex represents the frequencies of induced subgraphs, incident at the vertex, of the graphlet patterns. The transformed data array serves multiple types of network analysis: statistical or/and topological measures, comparison, classification, modeling, feature embedding and dynamic variation, among others. The library FGLT is distinguished in the following key aspects. (1) It is based on the fast, sparse and exact transform formulas which are of the lowest time and space complexities among known algorithms, and, at the same time, in ready form for globally streamlined computation in matrix-vector operations. (2) It leverages prevalent multi-core processors, with multi-threaded programming in Cilk, and uses sparse graph computation techniques to deliver high-performance network analysis to individual laptops or desktop computers. (3) It has Python, Julia, and MATLAB interfaces for easy integration with, and extension of, existing network analysis software.












More details in [the paper](#).

```
@inproceedings{floros2020b,
  title = {Fast Graphlet Transform of Sparse Graphs},
  booktitle = {{{IEEE High Performance Extreme Computing Conference}}},
  author = {Floros, Dimitris and Pitsianis, Nikos and Sun, Xiaobai},
  year = {2020},
}
```

Table 1: Sparse and exact formulas for the Fast Graphlet Transform (FGLT) with dictionary  $\Sigma_{16}$ . There are

Table 1. Sparse and exact formulas for the Fast Graphlet Transform (FGT) with dictionary  $\Sigma_{16}$ . There are two sets of formulas. The first set is for computing the raw frequencies  $\hat{d}_j$ ,  $0 \leq j \leq 15$ , i.e., the frequencies of subgraphs with the graphlet patterns in dictionary  $\Sigma_{16}$ . The formulas are tabulated in the two tables to the left. For two vectors  $a$  and  $b$ ,  $a - b$  denotes the sparse/rectified difference  $\max\{b - a, 0\}$ . The difference between two sparse matrices is similarly denoted. The auxiliary formulas are in the bottom table. The second set of formulas is for converting the raw frequencies to the net frequencies  $d_j$ ,  $0 \leq j \leq 15$ , i.e., the frequencies of induced subgraphs with the graphlet patterns. The matrix for forward conversion, from the net frequencies to the raw ones, is  $U_{16} = U_5 \oplus U_{11}$ . The matrix is shown to the right. The backward conversion matrix  $U_{16}^{-1}$  is explicitly formed and used, it has the same nonzero pattern as  $U_{16}$ .

$\Sigma_{16}$	Graphlet, incidence node	Formula in vector expression
	$\sigma_0$ singleton	$\hat{d}_0 = e$
	$\sigma_1$ 1-path, at an end	$\hat{d}_1 = p_1$
	$\sigma_2$ 2-path, at an end	$\hat{d}_2 = p_2$
	$\sigma_3$ bi-fork, at the root	$\hat{d}_3 = p_1 \odot (p_1 - 1)/2$
	$\sigma_4$ 3-clique, at any node	$\hat{d}_4 = c_3$

	$\sigma_5$ 3-path, at an end	$\hat{d}_5 = p_3$
	$\sigma_6$ 3-path, at an interior node	$\hat{d}_6 = p_2 \odot (p_1 - 1) - 2c_3$
	$\sigma_7$ claw, at a leaf	$\hat{d}_7 = A((p_1 - 1) \odot (p_1 - 2))/2$
	$\sigma_8$ claw, at the root	$\hat{d}_8 = p_1 \odot (p_1 - 1) \odot (p_1 - 2)/6$
	$\sigma_9$ paw, at the handle tip	$\hat{d}_9 = A c_3 - 2c_3$
	$\sigma_{10}$ paw, at a base node	$\hat{d}_{10} = C_3(p_1 - 2)$
	$\sigma_{11}$ paw, at the center	$\hat{d}_{11} = (p_1 - 2) \odot c_3$
	$\sigma_{12}$ 4-cycle, at any node	$\hat{d}_{12} = c_4$
	$\sigma_{13}$ diamond, at an off-cord node	$\hat{d}_{13} = D_{4,c} e/2$
	$\sigma_{14}$ diamond, at an on-cord node	$\hat{d}_{14} = D_{4,3} e/2$
	$\sigma_{15}$ 4-clique, at any node	$\hat{d}_{15} = T e/6$

$U_5$	$d_0$	$d_1$	$d_2$	$d_3$	$d_4$
$\hat{d}_0$	1				
$\hat{d}_1$		1			
$\hat{d}_2$			1		2
$\hat{d}_3$				1	1
$\hat{d}_4$					1

$$d_{0:4} = U_5^{-1} \hat{d}_{0:4}$$

$$d_{5:15} = U_{11}^{-1} \hat{d}_{5:15}$$

$U_{11}$	$d_5$	$d_6$	$d_7$	$d_8$	$d_9$	$d_{10}$	$d_{11}$	$d_{12}$	$d_{13}$	$d_{14}$	$d_{15}$
$\hat{d}_5$	1				2	1		2	4	2	6
$\hat{d}_6$		1			1	2	2	2	4	6	
$\hat{d}_7$			1		1	1			2	1	3
$\hat{d}_8$				1			1			1	1
$\hat{d}_9$					1				2		3
$\hat{d}_{10}$						1			2	2	6
$\hat{d}_{11}$							1			2	3
$\hat{d}_{12}$								1	1	1	3
$\hat{d}_{13}$									1		3
$\hat{d}_{14}$										1	3
$\hat{d}_{15}$											1

Auxilliary formulas	
$P_2 = A^2 - \text{diag}(p_1)$	$p_1 = A e$
$C_3 = A \odot A^2$	$p_2 = A p_1 - p_1$
$C_{4,2} = P_2 \odot (P_2 - 1)$	$p_3 = A p_2 - p_1 \odot (p_1 - 1) - 2c_3$

$D_{4,c} = A \odot (A (C_3 - A))$	$c_3 = C_3 e/2$
$D_{4,3} = A \odot C_{4,2}$	$c_4 = C_{4,2} e/2$
$T = A \odot [q_{ij}^T A q_{ij}], q_{ij} = a_i \odot a_j$	

## Or Do This One

We will implement step-by-step a shared memory vantage point tree construction, given a data set of  $d$ -dimensional points  $X$ . The structure is used to perform the  $k$  nearest neighbor search (kNN) of the points on the vantage-point tree as introduced in the publication:

Peter N Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, In Fourth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics Philadelphia, volume 93, pages 311–321, 1993.

The algorithm is based on a very simple idea: select a point as the vantage point, compute the distance of every point to the vantage point and split the points according to the median distance.

```
clear

n = 100000; d = 2; % works for any d

X = rand(n,d);

% assume vantage point is the last one
% get squares of distances from it
d = sqrt( sum((X(1:n-1,:) - X(n,:)).^2,2) );
% and find the median distance
medianDistance = median(d);

% plot them to confirm
clf; hold on; axis equal

plot(X(d <= medianDistance,1), X(d <= medianDistance,2), 'r.')
plot(X(d > medianDistance,1), X(d > medianDistance,2), 'b.')
plot(X(n,1), X(n,2), 'bo') % vantage point
```

And then repeat the same on the inner and outer partitions until no point is left, to build a balanced binary tree.

```

function T = vpTree(X)
% function T = vpTree(X)
% computes the vantage point tree structure with
%   T.vp : the vantage point
%   T.md : the median distance of the vantage point to the other points
%   T.idx : the index of the vantage point in the original set
%   T.inner and T.outer : vantage-point subtrees
% of a set of n points in d dimensions with coordinates X(1:n,1:d)
%
T = vpt(X, 1:size(X,1));

function T = vpt(X, idx)

    n = size(X,1); % number of points
    if n == 0
        T = [];
    else
        T.vp = X(n,:);
        T.idx = idx(n);

        d = sqrt( sum((X(1:n-1,:) - X(n,:)).^2,2) );

        medianDistance = median(d);
        T.md = medianDistance;

        % split and recurse
        inner = d <= medianDistance;
        T.inner = vpt(X( inner,:), idx( inner));
        T.outer = vpt(X(~inner,:), idx(~inner));
    end
end
end

```

You do not need to follow the MATLAB implementation shown here, but make sure you understand all of the above and design tests to confirm correctness so that we can rewrite it in C and then parallelize it.

## 0. Implement the vantage-point tree in C (2 points)

Pay attention your implementation to be correct and efficient. Decide what needs to be copied and what can be done in place. Find the median distance using [Quickselect](#).

## 1. Parallelize your implementation on multicore CPUs (3 points)

There are two things you can do in parallel

1. compute the distances in parallel
2. compute the inner set in parallel with the outer set

We can also compute the median in parallel, using the exact same techniques we discuss here, but it won't be as critical so you can skip it.

## 2. Threshold the parallel calls (1 point)

Assigning too little work to be done in parallel will actually slow down your computations. Modify your parallel implementation to switch to call the sequential code after a certain threshold size. Also restrict the maximum number of live threads to an upper bound. Experiment to identify the optimal threshold value and maximum number of threads for your implementation and hardware.

## 3. Calculate the all- $k$ -NN.

Calculate  $k$  nearest neighbors of all points, with  $k = 2^{[1:8]}$ .

## 4. Improve performance with MPI or CUDA

Extend your implementation to take advantage of either multiple CPUs with MPI or GPUs with CUDA.

## What to submit

- A 4-page report in PDF format (any pages after the 4th one will not be taken into account). Report execution time of your implementations with respect to the number of data points  $n$  and the number of dimensions  $d$ .
- Upload the source code on GitHub, BitBucket, Dropbox, Google Drive, etc. and add a link in your report.

## Κατάσταση Υποβολής

Κατάσταση Υποβολής	Καμία προσπάθεια
Κατάσταση βαθμολόγησης	Χωρίς βαθμό
Λήξη υποβολής εμπρόσθρων εργασιών	Sunday, 6 March 2022, 11:59 PM

---

**Υπολειπόμενος  
χρόνος**

Η εργασία είναι εκπρόθεσμη κατά: 147 ημέρες 12 ώρες

---

**Τελευταία  
τροποποίηση**

-

---

**Σχόλια υποβολής**

► Σχόλια (0)

Προσθήκη εργασίας με αποδοχή των όρων εξέτασης

Δεν έχετε κάνει ακόμη υποβολή.

