A GPU-Based Solution to Fast Calculation of Betweenness Centrality on Large Weighted Networks – OpenMP implementation

Francesco Zinnari

Objectives

This presentation is part of a project for the course of Advanced algorithms and parallel programming (Politecnico di Milano).

The aim of this project is that of **analyzing** the paper ''A GPU-Based Solution to Fast Calculation of Betweenness Centrality on Large Weighted Networks'' (Rui Fan, Ke Xu and Jichang Zhao, 2017), in order to formulate an equivalent and efficient solution to the problem, **implemented in OpenMP** and intended to be executed on **General Purpose Processors.**

1

Understanding the problem

An analysis of the Betweennes Cenrality problem and Brandes' Solution

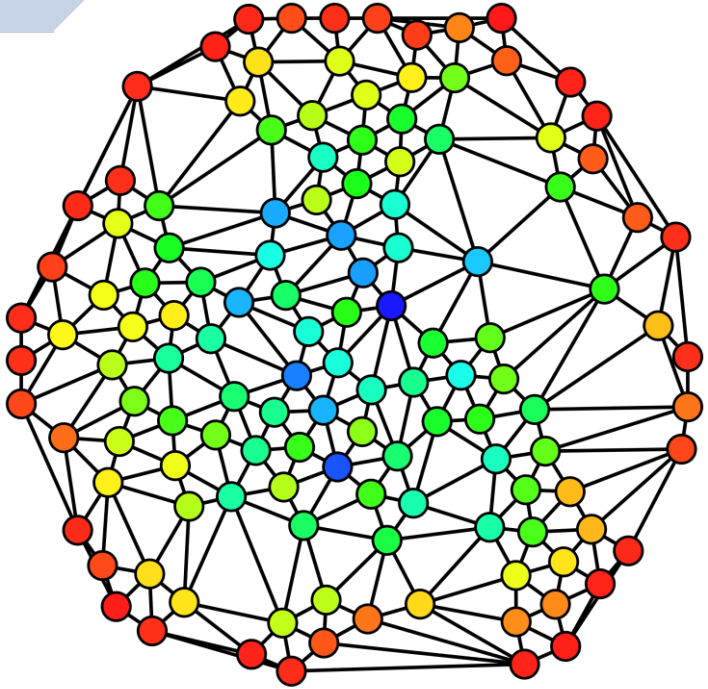
Betweenness centrality (BC)

Betweenness centrality is a measure of **centrality** in a graph, based on shortest paths.

Betweenness centrality measures the extent to which a vertex lies on paths between other vertices

$$C_B(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v).$$

where $\delta_{st}(v)$ is the pair-dependency, i.e. the ratio of shortest paths between s and t on which v lies on



Brandes' algorithm

Traditionally, the computation of betweenness was divided into two sub-tasks:

1. Compute the length and number of shortest paths between all pairs (SSSP problem)
   $O(nm)$ **unweighted** $O\left(nm + n^2 log(n)\right)$ **weighted**
2. Sum all pair-dependencies $\Theta(n^3)$ time complexity and $O(n^2)$ space complexity

The need for explicit summation of all pair-dependencies is eliminated by introducing the concept of **dependency** of a vertex $s \in V$ on a single vertex $v \in V$.

$$\delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}(v).$$

Brandes' algorithm (cont.)

$$CB(v) = \sum_{s,t \in V} \delta_{s,t}(v) = \sum_{s \in V} \delta_s(v)$$

The crucial observation is that these partial sums obey a recursive relation:

$$\delta_{s\bullet}(v) = \sum_{w\,:\,v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w)) \,.$$

so that, once given the directed acyclic graph of shortest paths from s in G, this equations allows to compute the dependencies of s on all of the other vertices in $O(m)$ time, reducing the BC algorithm's computational cost to $\boldsymbol{O(nm + n^2 log(n))}$ time and $O(n + m)$ space for weighted graphs. For unweighted graphs, running time reduces to $O(nm)$.

2

CUDA parallel implementation for weighted graphs

GPU-based Algorithm

The proposed GPU-based Algorithm accomplishes both coarse-grained and fine-grained parallelism.

- **Coarse-grained:** one thread block processes one root vertex s
- **Fine-grained:** threads within the block compute shortest paths and dependencies related to s

A **Compressed Sparse Row (CSR)** format is used to store the input graph, for space efficiency reason.

Note: The algorithm describes the idea behind the fine-grained part, being that the coarse-grained parallelism is simply obtained by assigning different roots to different thread blocks.

GPU-based Algorithm

The algorithm is divided into three main parts:

1. **Variables initialization**
2. **Shortest Path Calculation by Dijkstra Algorithm** (+ computation of the number of shortest paths from s to all reachable vertices + creation of the DAG of shortest paths from s)
3. **Dependency Accumulation**

The work-efficient version, where threads are assigned only to nodes being in the frontier set (so nodes that will perform the calculation job),
will be the one considered here

1. Variables initialization

As expected, since we are working on a SIMD architecture (GPU's Streaming Multiprocessors), the only kind of parallelism we will find is

Data Parallelism

for $v \in V$ **do in parallel**
  $U[v] \leftarrow 1$
  $d[v] \leftarrow \infty$
  $\sigma[v] \leftarrow 0$
  $\delta[v] \leftarrow 0$
  $ends[v] \leftarrow 0$
  $S[v] \leftarrow 0$
end for

Parallel Region

$d[s] \leftarrow 0$
$\sigma[s] \leftarrow 1$
$U[s] \leftarrow 0$
$F[0] \leftarrow s$
$F_{len} = 1$
$S[0] \leftarrow s; S_{len} \leftarrow 1$
$ends[0] \leftarrow 0; ends[1] \leftarrow 1; ends_{len} \leftarrow 2$
$\Delta \leftarrow 0$

2. Dijkstra and Parallelization

Due to the very same nature of the algorithm itself (picking one node from the frontier node each time), parallelizing it is a difficult task.
However, the restriction of picking one node at a time from the queue can be relaxed and **several nodes can be inspected simultaneously** in the next step if we apply a specific condition when settling the nodes, making them part of the frontier set.

The condition is that only the nodes $v \in U_i$ (unsettled nodes in iteration i) having $\mathbf{D(v) < \Delta_i}$ (that is distance from s, the root, to v less than Δ_i) can be added to the frontier and marked as unsettled.

$$\Delta_i = \min\{(\mathrm{D(u)} + \Delta_{node\ u}) : \mathrm{u} \in U_i\} \text{ and } \Delta_{node\ u} = \min(weight(u, v) : (u, v) \in E)$$

3. Dependencies accumulation

Having completed Dijkstra's execution, we now have the **DAG** of **shortest paths from the root**.

We proceed elaborating (in parallel) the nodes that have been added to the DAG during the same iteration, from the last iteration to the first one.

This will give us $\delta_{s(v)}$ where s is the root and $v \in V \backslash \{s\}$, which is one element of the summation $CB(v) = \sum_{s \in V} \delta_s(v)$

```
1:  depth ← ends_len − 1
2:  while depth > 0 do
3:    start ← ends[depth − 1]
4:    end ← ends[depth] − 1
5:    for 0 ≤ i ≤ end − start do in parallel
6:      w ← S[start + i]
7:      dsw ← 0
8:      for v ∈ neighbors(w) do
9:        if d[v] = d[w] + weight_wv then
10:         dsw ← dsw + σ[w]/σ[v] ∗ (1 + δ[v])
11:       end if
12:     end for
13:     δ[w] ← dsw
14:     if w ≠ s then
15:       atomicAdd(CB[w], δ[w])
16:     end if
17:   end for
18:   depth ← depth − 1
19: end while
```

Parallel Region

3

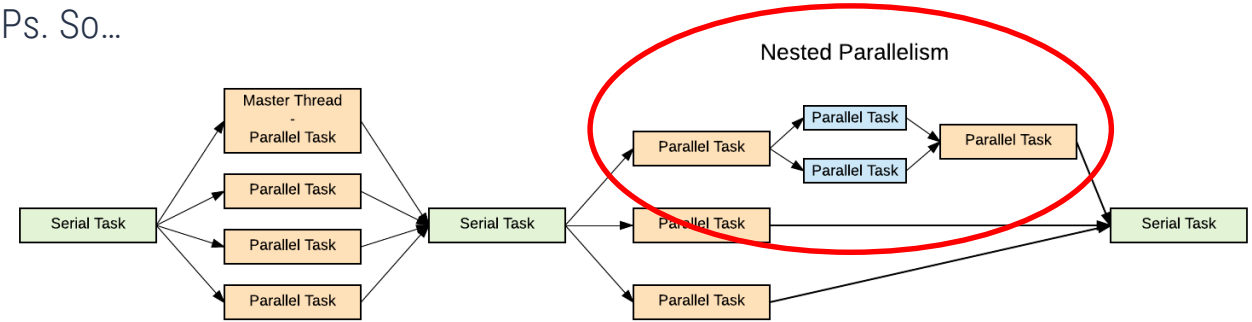
OpenMP implementation

What changes with respect to CUDA?

Coarse-grained parallelism

While CUDA supports **thread blocks**, being that it was conceived for Nvidia GPU's architecture, OpenMP does not.

The closest feature OpenMP possesses is **Nested Parallelism**, the ability of **creating parallel regions inside other parallel regions**. However, keep in mind that CUDA's algorithm was conceived for massively parallel architectures, like GPU's streaming multiprocessors, while we will be working on a limited amount of cores on GPPs. So…



Nested Parallelism: yes or no?

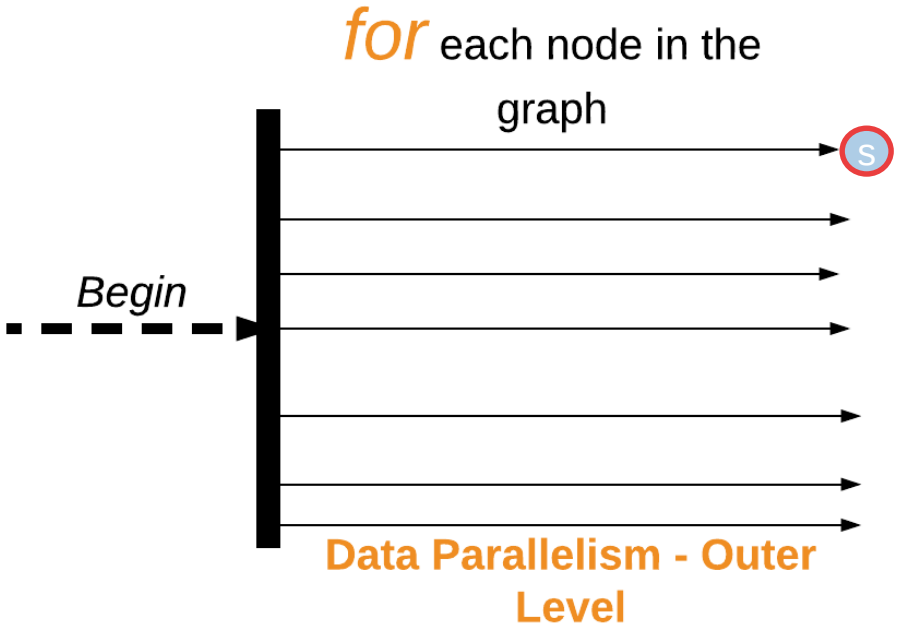
… While nested parallelism would be ideal to mantain the same structure as CUDA's algorithm and to exploit computational power, creating nested parallel regions **adds** significant **overhead**.

If there is **enough parallelism at the outer/inner level and the load is balanced**, nested parallelism might not be needed.

We decide to implement **both** a version of the algorithm that exploits nested parallelism and other two versions that only apply parallelism to the outer or inner level to compare their running times

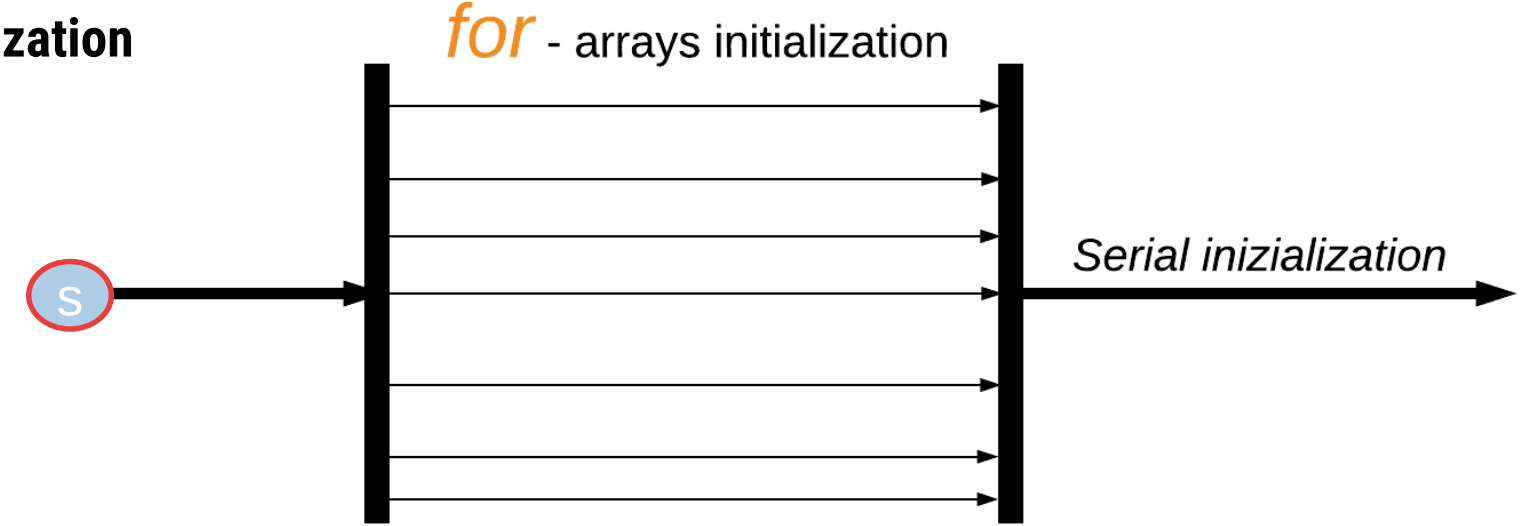
Trade off!

OpenMP Flow Graph



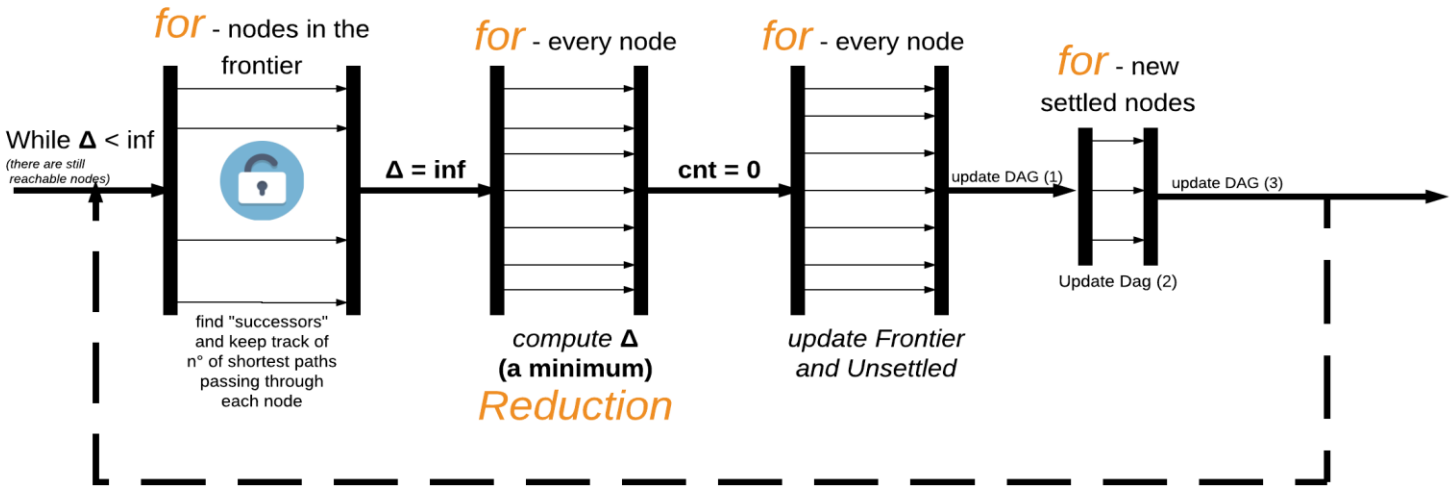
OpenMP Flow Graph

1) Variables Initialization



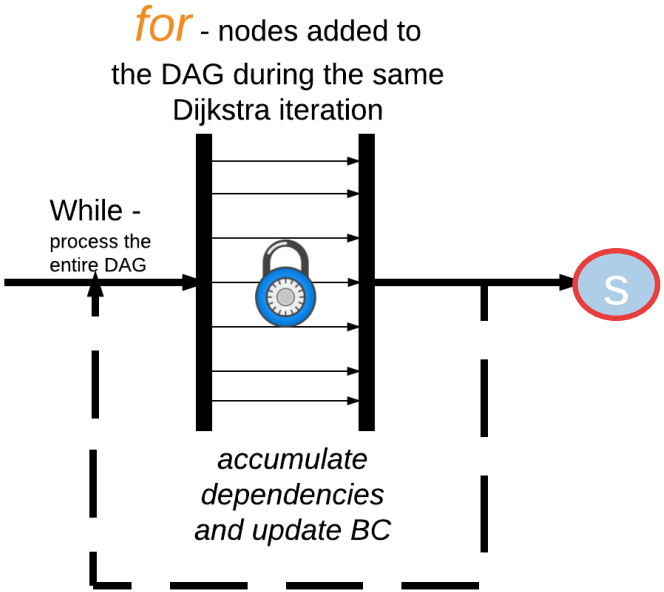
OpenMP Flow Graph

2) Dijkstra execution



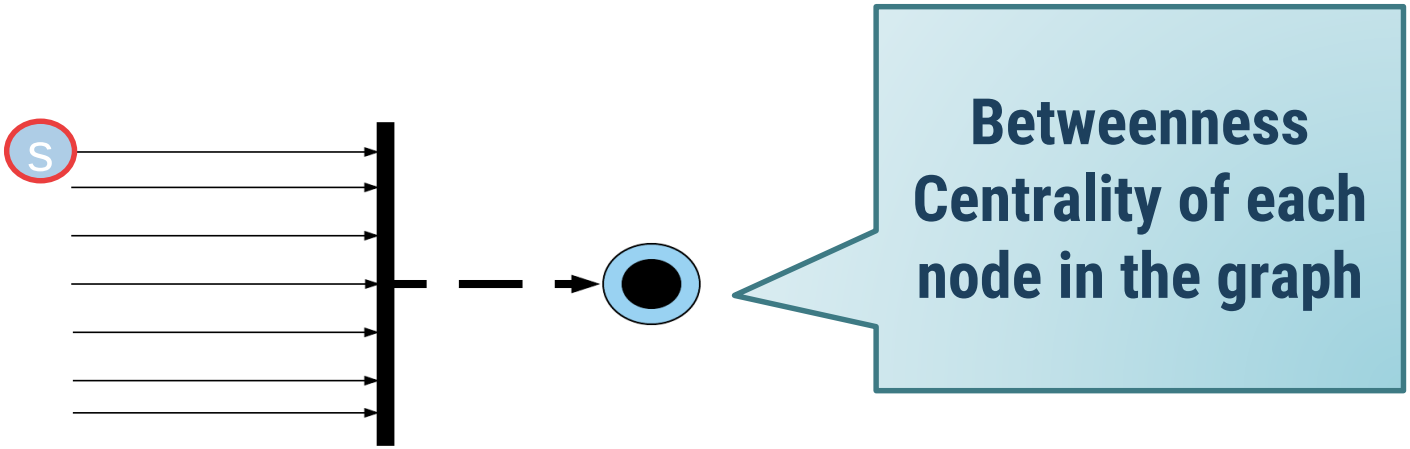
OpenMP Flow Graph

3) Dependencies accumulation



OpenMP Flow Graph

Obtaining the final result



OpenMP Flow Graph

Let’s take a look a the code…

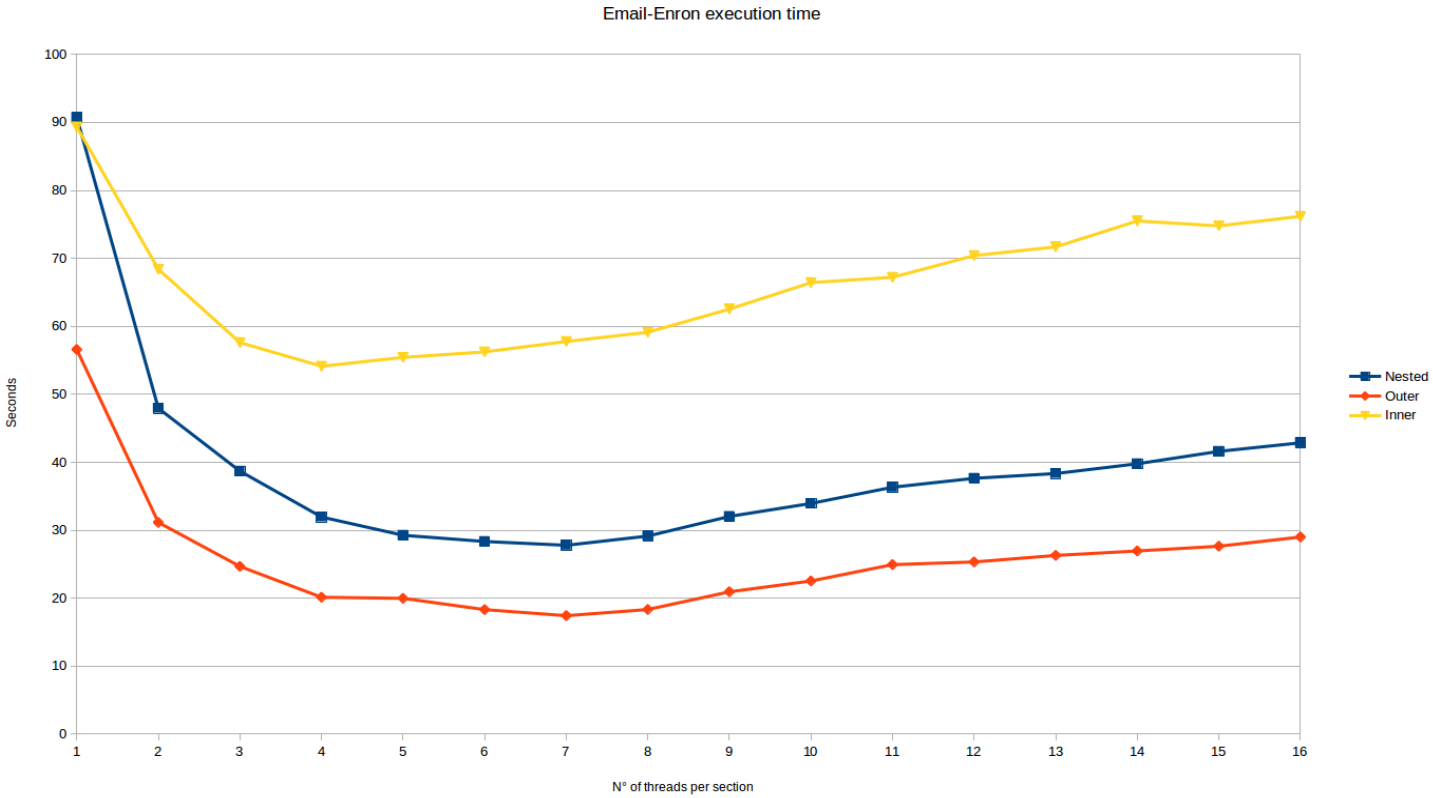
4

OpenMP Results

Email Enron dataset

Number of nodes: **8000**

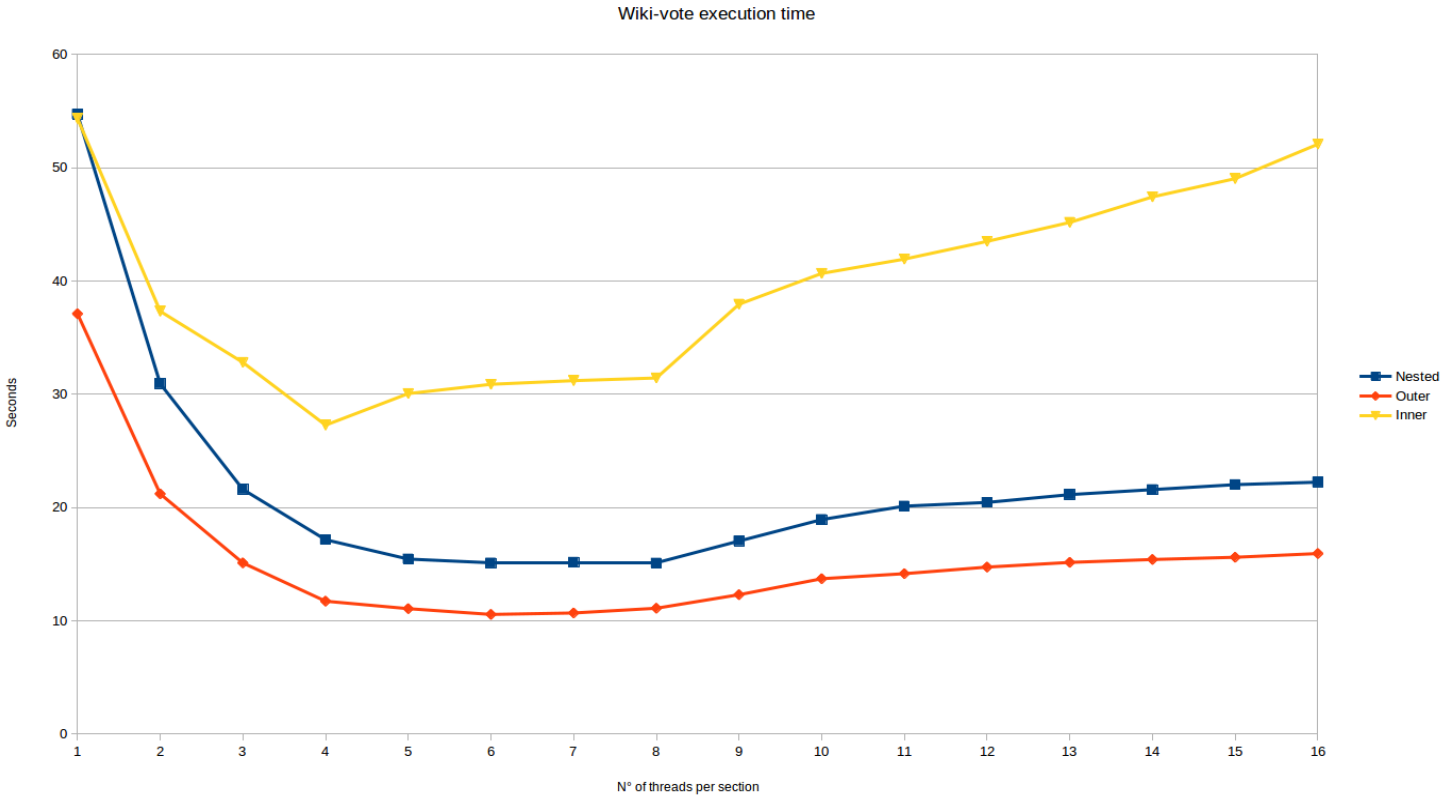
Number of edges: **193776**



Wiki Vote dataset

Number of nodes: **8500**

Number of edges: **103689**



Some comments on results

These results were obtained with an Intel Core i7-4700MQ CPU, a **Quad Core** processor supporting Hyper-threading (2 threads SMT)

We obtained a maximum speedup with respect to the single threaded version of 3.3 - 3.6 for the two datasets.

As expected, on large-scale networks and on GPP the **nested** version of the algorithm, which mimics the one implemented in CUDA, **didn't bring any advantage** when compared with the one that only exploits the **outer** parallelism. This is because the latter already exploits at best the parallelism offered by the machine, but **without the overhead** of creating threads at each cycle.

Some comments on results

The version that only exploits **inner** parallelism obviously performed the worst when only considering the execution time, since it might be unable to exploit parallelism at best each time. Remember however that this version has substantial advantages when it comes to its memory footprint.

Finally, observe how the best results are obtained with the use of 6/7 threads, since the processor uses Simultaneous MultiThreading

References

[1] A GPU-Based Solution to Fast Calculation of Betweenness Centrality on Large Weighted Networks *(Rui Fan, Ke Xu and Jichang Zhao)*

[2] Brandes U. A faster algorithm for betweenness centrality. The Journal of Mathematical Sociology. 2001;25(2):163–177.

[3] Crauser A, Mehlhorn K, Meyer U, Sanders P. A Parallelization of Dijkstra's Shortest Path Algorithm. In: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science; 1998. p. 722–731.