MASSIVE PARALLELISM WITH GPUS FOR CENTRALITY RANKING IN COMPLEX NETWORKS

Frederico L. Cabral, Carla Osthoff, Rafael Nardes, Daniel Ramos[1]

[1]Laboratório Nacional de Computação Científica

ABSTRACT

Many problems in Computer Science can be modelled using graphs. Evaluating node centrality in complex networks, which can be considered equivalent to undirected graphs, provides an useful metric of the relative importance of each node inside the evaluated network. The knowledge on which the most central nodes are, has various applications, such as improving information spreading in diffusion networks. In this case, most central nodes can be considered to have higher influence rates over other nodes in the network. The main purpose in this work is developing a GPU based and massively parallel application so as to evaluate the node centrality in complex networks using the Nvidia CUDA programming model. The main contribution of this work is the strategies for the development of an algorithm to evaluate the node centrality in complex networks using Nvidia CUDA parallel programming model. We show that the strategies improves algorithm´s speed-up in two orders of magnitude on one NVIDIA Tesla k20 GPU cluster node, when compared to the hybrid OpenMP/MPI algorithm version, running in the same cluster, with 4 nodes 2 Intel(R) Xeon(R) CPU E5-2660 each, for radius zero.

KEYWORDS

Complex Networks, Graphs, Centrality Ranking, GPGPU, CUDA.

1. INTRODUCTION

Historically, the performance improvement of computer applications is achieved by faster central processing units, with higher clock frequencies along each new generation. This way, it was possible to have more operations executed per second (FLOP/S or FLOPS).

As a solution to increase the computational power in terms of FLOPS, parallelism became widely used for the last few decades.

The great appeal of faster computers and the difficulties of producing supercomputers with hundreds and thousands of processors at a low cost and low energy consuming, impulse scientists to look for another alternative, which was found in the so called many core architecture, present in modern General Purpose Graphics Processing Units (GPGPUs). For last few years, the fastest supercomputers and the most efficient ones, in terms of MFLOPS/Watt, are all constructed with GPGPUs, like Nvidia cards, and accelerators, like Intel Xeon Phi, all based on many core concept.

On the other hand, the analysis of complex networks has become one of the main goals of science [1]. Describing, characterizing and above all extracting relevant data from complex networks currently draws the attention of many fields of research, as biology, economics, social science, computer science and so on. The interest of so many fields is due to the fact of a huge amount of systems shaped that way has come up by now such as the Internet, the World Wide Web, social networks, organizational networks, distribution networks, neural networks, traffic networks, etc. [1].

One of the goals in network analysis is figuring out the behaviour of the nodes along the net. One tool that was proposed to assay complex networks is the network centrality [2, 3]. Broadly speaking, network centrality tells the relevancy of each node along the net. In this context, different approaches have been proposed in years, each of them assesses the problem in a different point of view. For instance, the network centrality to evaluate network robustness to fragmentation [4, 5] or to identify the most important nodes in spreading information in diffusion networks [6, 7].

According to [8] many strategies to compute centralities have been proposed in the last decades, a lot of them having a high computing cost. Although there are low computing cost techniques as degree centrality (estimates the centrality based only on the neighbours of each node), more accurate procedures and nearer real world networks as closeness centrality commonly have very high computing costs.

Closeness centrality [9] depends on ascertaining the lowest path between all pairs of network nodes, and the centrality of every node is estimated by the minimum distances average to another nodes.

The high computing demand and the need of whole network topology knowledge (reason why these techniques are called global) are hitches that imposes a rare applying on large-scale networks of nowadays.

The DACCER method (Distributed Assessment of the Closeness Centrality Ranking) purposed by [8] adopts a local approach thus it is independent of full network awareness. That method is restricted to evaluate a determined neighbourhood bounded surrounding to each node in order to be generated the ranking nodes according their centralities.

In the complex networks study, the knowledge of nodes position in a centrality ranking is, invariably, more crucial than knowing the precise value associated to each node. Therefore, the local strategy suggested in [8] has its testified applicability because of its high correlation degree, vouched empirically, between DACCER ranking and the one obtained by the method of high overall cost: Closeness Centrality [9].

This paper presents a CUDA implementation of DACCER and its results with experiments that were carried out on a two nodes Linux cluster with an infini-band network channel and a NVIDIA Tesla k20 GPU per node. We simulate networks of sizes 10.000, 100.000, 200.000, 400.000, 600.000 and 1.000.000 vertices. For every size of network (in amount of vertices) aforementioned, we carried out tests with four different values of minimum degree, i.e. the least amount of edges connected to each vertex in the net, specified in Barabasi-Albert model for networks [15].

According to [8] a radius equal to 2, of each vertex degree can ensure a high correlation degree between the centrality ranking produced by DACCER and the one produced by global strategies. Based on that fact, performed tests were focused on radius smaller than or equal to 2. However, the developed solution is general enough to be used with any value of radius.

We evaluate the performance of DACCER Cuda parallel algorithm against an earlier DACCER OpenMP/MPI parallel algorithm version [8] with 32 simultaneous threads on a hybrid OpenMP/MPI environment.

The comparative results show that, on the tested networks, the execution for radius equal to zero and one on a GPU Tesla K20 was in average about two orders of magnitude faster than the execution on 32 cores CPU cluster.

The remainder of this paper is organized as follows. Section 2 presents Network Representation in Terms of Data Structures background and related work, section 3 presents Parallel Programming Models background, while section 4 presents the CUDA version algorithm strategy for DACCER. The performance evaluation, experimental results and experimental analysis are presented in section 5. The last section presents conclusions and future works.

2. NETWORK REPRESENTATION IN TERMS OF DATA STRUCTURES

The study of networks is a relevant field; in order to model networks the Mathematics applies the Graph Theory. Through that theory, one defines a network as a set of items called vertices or nodes. Connecting these items exist structures called edges [2].

A graph is an abstraction structure applied in a huge range of problems. In computer science, algorithms to deal with this kind of structure arises since an enormous variety of problems has its solution based on graph representations.

Visually, can be represented drawing items (vertices or nodes) as dot or circles. The interconnections between the items are represented by lines, which link the dots.

2.1 Representation of graphs

Thus, a graph *G* can be formally be defined as a pair of sets $G = (V,E)$, which $V = \{1,2,3\ldots n\}$ is the set that contains n vertices of the graph and $E=\{1,2,3,\ldots m\}$ is the set which contains all of its m edges. We are denoting $|V|$ and $|E|$, respectively, the number of vertices and the number of edges of the graph *G*.

If the connections between the elements depend on the origin, in other words, if given a node v, it can be related to another node w without necessarily reciprocal is true, we have a directed graph, therefore connections need to be represented by the use of arrows evincing the direction of the connection.

In order to represent graphs according to [10] there are two standard approaches: adjacency matrix or a collection of adjacency lists.

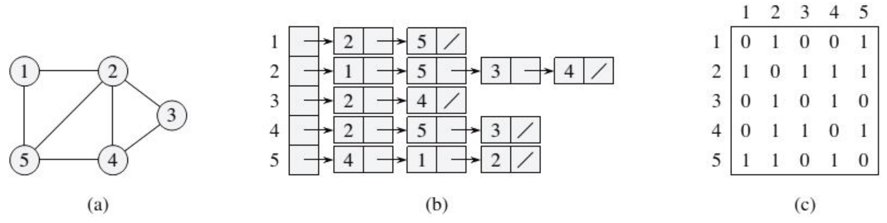
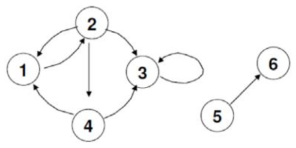


Figure 1. Two representations of an undirected graph. (a) An undirected graph G with 5 vertices and seven edges. (b) An adjacency list representation of G. (c) The adjacency matrix representation of G.

Adjacency matrices have an inconvenience in sparse graph ($|E|$ is much smaller than $|V|^2$). In that situations, one ends up using too big matrices for representing a structure that could be represented in a more compact way. That is the reason why generally representing a graph by adjacency lists is preferred. Nevertheless adjacency matrices are quite useful for very dense graphs ($|E|$ is closer to $|V|^2$) or when it is necessary to be able to tell quickly if there is a connection between two given vertices.

An adjacency matrix representation consists of a $|V|$ x $|V|$ matrix $A = a_{ij}$ such that $a_{ij} = 1$ if there is a connection between vertex i and vertex j, otherwise $a_{ij} = 0$.



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 & 0 & 0 \\ 4 & 1 & 0 & 1 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 1 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Figure 2. A directed graph represented by an adjacency matrix.

For adjacency lists representations one uses an array V of size |V|. Every element from that array represents a vertex v of the graph and contains a pointer to a linked list that keeps all its neighbours. Therefore there are linked list in this representation.

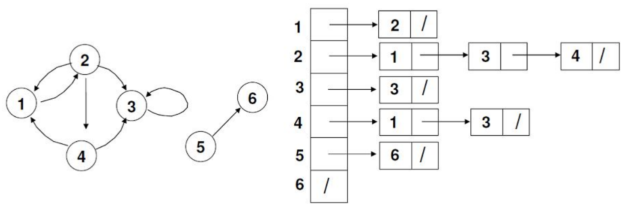


Figure 3. A directed graph represented by adjacency lists.

2.2 Representation of graphs in GPU

According to [11] many complex data structures have already been studied as hash tables in order to represent graphs in search of high performance CPU algorithms. However, the memory model of GPU's is optimized to graphic rendering not supporting efficiently data structures defined by the users. Nevertheless, CUDA model manages the memory as general arrays thus allowing CUDA to be able to support data structures in a more efficient way.

Adjacency matrix can be considered as a good choice to be used on GPU, but with a severe limitation: spatial complexity $O(|V|^2)$ make infeasible its use in large graphs thus limiting radically the size of supported graphs implemented on a GPU, where available memory is a limiting factor. Adjacency lists have spatial complexity of $O(|V| + |E|)$, thus they are more suitable to large graphs.

In this scenario, [11] proposed a compact adjacency list form, whose adjacency list is packed into a single large array.

Vertices of graph $G(V,E)$ are represented as an array Va. Another array Ea of adjacency lists stores the edges with edges of vertex $i+1$ immediately following the edges of vertex i for all i in V. Each entry in the array Va corresponds to the starting index of its adjacency list in the edge array Ea. Each entry of the edge array Ea refers to a vertex in vertex array Va.

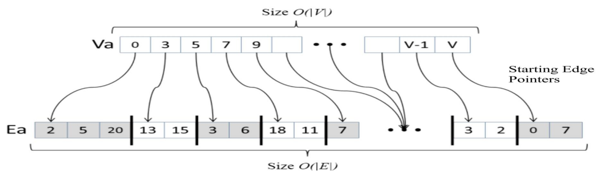


Figure 4. Graph representation is in terms of a vertex list that points to a packed edge list.

For DACCER algorithm, this paper introduces some extensions of compact adjacency list, in which *Ea* will store not only the direct neighbours of some node, but the neighbours of certain distance *k* in a structure called *Eak*, and all neighbours until certain *k* in *EakAccum*. These structures will be necessary for CUDA code in order to compute the volumes until certain radius, by adding the degrees of all elements inside the neighbourhood.

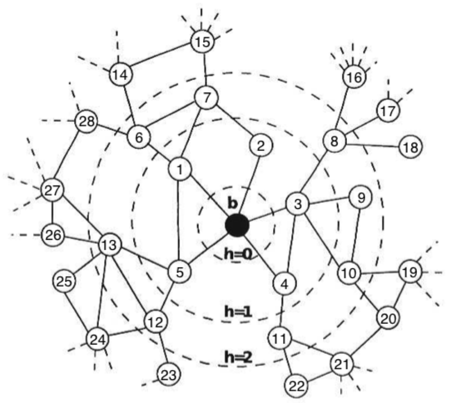


Figure 5. Example of a graph and a neighbourhood of some node

In figure 5 there is a graph example where node zero and its neighbours are shown. The direct neighbours (1-neighbourhood) are 1, 2, 3, 4 and 5, while the 2-neighbourhood is $H^0_2 = \{6, 7, 8, 9, 10, 11, 12, 13\}$. Figure 6 shows the *Eak* and *Vak* for node zero with k = 2 in graph example of figure 5.

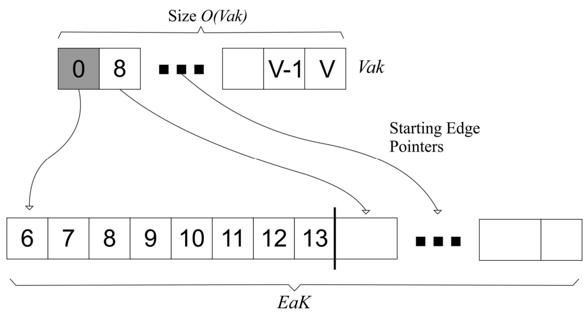


Figure 6. Compact Adjacency List for k = 2

Another important structure used in CUDACCER (DACCER on CUDA) is the Cumulative Compact k-Adjacency List (CCk-AL) which stores all the neighbours from a node until some radius. For example in figure 5, the neighbours of node zero until radius = 2 is $N^0_2 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$. Figure 7 shows *EakAccum* and *VakAccum* for node zero with k = 2 in graph example of figure 5.

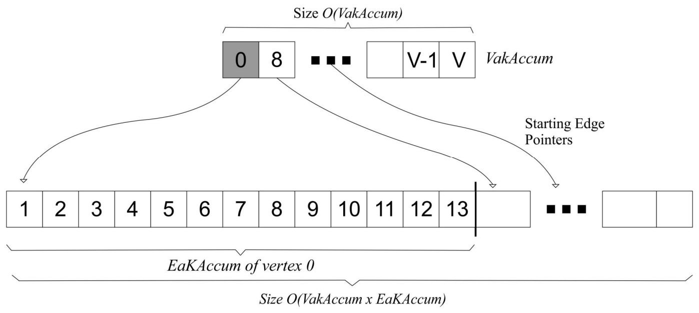


Figure7. Cumulative Compact Adjacency List for k = 2

3. PARALLEL PROGRAMMING MODELS

As a solution to increase the computational power in terms of FLOPS, parallelism became widely used for the last few decades. The Message Passing Interface (MPI) defines an API to write parallel code that allows processes to communicate with each other, exchanging messages. This is a very adequate approach for loosely coupled environment, in which each processor unit accesses a private memory. In general, these processes send/receive messages by a network link. The OpenMP standard defines an API for multi threading programming that is applied in tightly coupled environment, where all processor units share the same memory. A typical environment adequate to OpenMP using is the multi core technology so common nowadays. One can get a very efficient approach in parallel programming mixing MPI with OpenMP, in a hybrid model that fits perfectly in modern clusters that combine multi core processors that communicate by a network link [12].

There are programming models for GPGPUs like, OpenCL and Nvidia Compute Unified Device Architecture (CUDA). Conceptually, CUDA is an extension to C language that allows programmers to write code without the necessity of learning the details of graphics hardware, like shader languages or graphics primitives [13]. The code is target to both CPU, called host, and GPU, called device. The host creates and spawns multi thread tasks to run into the GPU, specifically inside the many CUDA cores. The functions that run in the device are called kernel functions in CUDA vocabulary, so a typical program have a CPU skeleton and some kernels that can be invoked by it.

Figure 8 shows the differences between the multi core and many core architectures. One can notice that the CPU has one Control Unit for all of its Arithmetic Units and a “big” cache memory, while GPU has many Control Units and Many “small” caches.

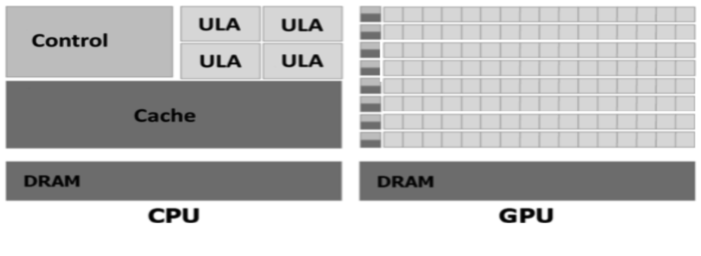


Figure 8. Multicore and manycore architectures

The GPU hardware schedules the threads by units called warps that are executed by some SIMD vector processors, which means that ideally, only one access to memory for an instruction and a broadcast of that instruction to all processors are needed. Because of this particular characteristics, branches in the code (if-then-else, for, while, etc.) can slow down the execution by causing divergence in the execution flow. This is an important issue to the application studied in this paper, as shown in further sections.

When a kernel function is invoked, a grid of threads is created. A grid contains hundreds, thousands or millions of threads, once not all of them are executed at the same time, due to the warp scheduling mentioned before. The number of threads created can be higher that the number of CUDA cores, for example. Inside a grid, threads are organized into blocks that can be one, two or three dimensions. This organization is shown in figure 9, where one can see a two dimensional grid with two dimensional blocks, which means that each thread is numbered by a ordered pair inside the block.

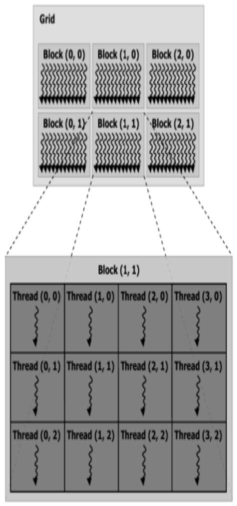


Figure 9. Two-dimensional grid with two-dimensional blocks.

A block consists of a group of threads that can cooperate in two different ways: (1) synchronize the execution through an instruction called *__syncthreads* that works as a barrier; (2) sharing data in the shared memory, which has a low latency. Inside a block, threads are identified by the *threadIdx* reserved word. In a three-dimensional block, for example, there exists *threadIdx.x*, *threadIdx.y* and *threadIdx.z* coordinates. Similarly, blocks inside a grid are identified by *blockIdx.x* and *blockidx.y* coordinates, unless the grid is one dimensional, in which block are identified only by *blockidx.x* coordinate.

Another important issue for this paper is the memory model in CUDA. There are many types of memory according to its location and function. A hierarchy begins with registers that are assigned to threads followed by shared memory that are assigned to blocks. These two types are located inside the GPU chip. The next is the constant memory that can be used to store data that can only be read by the threads but can be written or read by the host code. The last level is represented by the global memory that can written and or read by the host and the threads. Constant and global memory are located outside the GPU chip, but located inside the GPU card, connected by a bus.
The communication between the host and device, to read/write the global memory is an important bound for a kernel speedup. Ideally, one want to increase the Compute to Global Memory Access Ratio (CGMA ratio) in order to obtain a significant gain for the kernel speedup. Figure 10 shows the CUDA memory model.

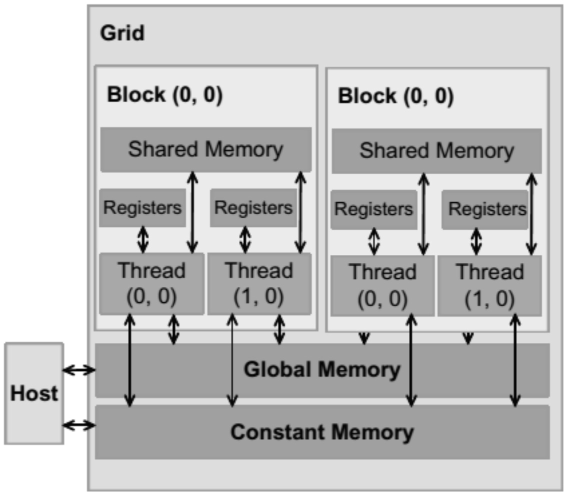


Figure 10. Memory Model in CUDA

In this work, for simplicity, all data structures describer in section 4, are allocated in global memory. The use of shared memory in some structures, in order to gain performance, are to be considered in further work.

3. ALGORITHM OUTLINE

In this section, the DACCER implementation with CUDA is presented. Before describing the pseudo code for the host and device, it is necessary to describe the data structures used.

VaD and **EaD** are, as seen before, the basic structures used to represent graphs in CUDA with the so called Adjacency Compact List. These two one dimensional arrays are the basic structures from where the others are conceived.

VakAccumD and **EakAccumD** are extensions from Vak and Eak. The main idea is to have structures that store the neighbours from each node until the "layer k". For example, these arrays for k = 2, stores all neighbours from 1 and 2 frontiers for each node. This pair of arrays its being introduced in this paper with the name Cumulative Compact k-Adjacency List (CCk-AL)

VakD and **EakD** are too, extensions of Va and Ea, but they store the neighbour of "k" distance from each node.

Basically, **VaAntD** and **EaAntD** store the same information that VakD and EakD, but for the anterior layer, "k-1".

VaAntAccumD and **EakAntAccumD** are similar to VakAccumD and EakAccumD, but for the anterior layer, "k-1". It means that they store all neighbours for each node until the "k-1" layer.
Once the five main pairs of data structures was presented, it is important to point out the need of more four arrays that stores the numbers of neighbours in different steps of the process described later in this section.

The first one is **ContNeighbourskD** that contains the number of k-neighbours for each node. **ContNeighboursAntkD** contains the number of (k-1)-neighbours. The next two structures are layer cumulative. **ContNeighboursAccumkD** and **ContNeighboursAntAccumkD** contain the number of neighbours of each node until k and k-1 layers respectively.

Two last arrays are necessary: **DegreesD** contains the number of neighbours for each node, which is known as degree of the node. **VolumesD** is the structure that stores the final result, the volume of each node. As explained before, the volume of a node is the sum of the degree from all the neighbours until some radius. DACCER proposal regards on using the volume as a local metric to approximate centrality.

The host code, which runs on CPU, is shown next. The kernel functions invoked by this code are explained further in this section.

4.1 Algorithm 1: CPU/HOST Skeleton

```
01: Read the graph from a text file and creates Ea and Va in the HOST.
02: Allocates EaD and VaD in the DEVICE
03: Transfers Ea and Va from HOST to DEVICE in EaD and VaD
04: Invokes kernel InitializeArrays
05: Invokes kernel ComputeDegrees
06: if (radius == 1)
07:     Invokes kernel ComputeVolumeR1
08: if (radius > 1)
09:     for (layer from 2 to radius) do
10:         Invokes kernel ComputeNumberOfkNeighbours
11:         Invokes kernel ComputeNumberOfUntillkNeighbours
12:         Transfers ContNeighbourskD from DEVICE to HOST
13:         Fills Vak and updates VakAccum using ContNeighbourskD
14:         Transfers Vak and VakAccum from HOST to DEVICE
15:         Computes the size of new EakD using ContNeighbours and Vak
16:         Invokes kernel FillEakD
17:         Computes the size of EakAccumD using ContNeighboursAccumkD and
            VakAccum
18:         Invokes kernel FillEakAccumD_1step
19:         Invokes kernel FillEakAccumD_2step
20:     end-for
21:     Invokes kernel ComputeVolumes
22: end-if
23: Transfers VolumesD from DEVICE to HOST
```

Important to notice that the entries for this algorithm are the graph itself and the radius in which the volumes are desired. Each kernel function is described next.

4.2 Algorithm 2: Kernel function InitializeArrays

This function initialize some arrays in the GPU assigning zero to all their positions.

```
1. If (thread_index < number of vertices)
2.     DegreesD[thread_index] = 0
3.     ContNeighbourskD[thread_index] = 0
4.     VolumesD[thread_index] = 0
5. end-if
```

4.3 Algorithm 3: Kernel function ComputeDegrees

This function counts how many direct neighbours each node has and stores it in the corresponding position of DegreesD.

```
1. If (thread_index < number of vertices)
2.     x = thread_index + 1
3.     DegreesD[thread_index] = VaD[x] – VaD[thread_index]
4.     VolumesD = DegreesD
5. end-if
```

After this execution, VolumesD has exactly the same values from DegreesD, because for radius = 0, the volumes correspond to the degree of each node of the graph. In this case, both tests in algorithm 1, in lines 6 and 8 will result false and execution jumps to line 23 with the volumes already computed properly.

4.4 Algorithm 4: Kernel function ComputeNumberOfkNeighbours

This kernel computes the number of neighbours at layer k and stores it at ContNeighbourskD. For each neighbour at layer k-1, a group of its neighbours are identified at EaD. The size of these groups are used to count the number of neighbours at layer k.

```
1. if (thread_index < number of vertices)
2.     ContNeighbourskD[thread_index] = 0
3.     for all (k-1)-neighbours of thread_index do
4.         Defines beginning of the group in EaD
5.         Defines ending of the group in EaD
6.         ContNeighbourskD[thread_index] += (groupEnd – groupBegin + 1)
7.     end-for
8. end-if
```

4.5 Algorithm 5: Kernel function ComputeNumberOfUntillkNeighbours

This kernel computes the total number of all neighbours until the layer k and stores this information at ContNeighboursAccumkD. It is a simple procedure: just add the number of neighbours accumulated at layer k-1 and number of neighbours at layer k.

```
1. if (thread_index < number of vertices)
2.     ContNeighboursAccumkD[thread_index] =
       ContNeighboursAntAccumkD[thread_index] + ContNeighbourskD[thread_index]
3.end-if
```

4.6 Algorithm 6: Kernel function FillEakD

This kernel discovers the neighbours in the k layer and fills EakD with the appropriate information. It is important to remember that Eak is an extension of original Ea, but for a specific layer. The merge procedure is key in this process. Each group of elements found in EaD and EakD itself are already sorted, so it is possible to merge them into an auxiliary structure that will end up sorted too with *O(n)* complexity, allowing to eliminate repeated values.

```
1. if (thread_index < number of vertices)
2.     for all (k-1)-neighbours of thread_index
3.         Defines beginning of the group in EaD
4.         Defines ending of the group in EaD
5.         Merges the group in EaD with actual EakD into as auxiliary structure
6.         Stores the auxiliary structure in EakD
7.     end-for
8. end-if
```

The next two kernels fill EakAccumD, in two separate steps, which is the structure where the neighbours until layer k of each node is stored. This structure will be used to compute the volume. The pair VakAccumD and EakAccumD are called Cumulative Compact k-Adjacency List (CCk-AL).

4.7 Algorithm 7: Kernel function FillEakAccumD_1step

```
1. if (thread_index < number of vertices)
2.        pos = VakAccumD[thread_id]
3.        pivot = pos
4.        for (index i varying from pivot to pivot + ContNeighboursAntkD[thread_index]) do
5.                EakAccumD[pos] = EakAccumD[i]
6.                pos ++
7.        end-for
8. end-if
```

4.8 Algorithm 8: Kernel function FillEakAccumD_2step

```
1. if (thread_index < number of vertices)
2.        merges EakAccumD with EakD into an auxiliar structure
3.        stores the auxiliary structure in EakAccumD
4. end-if
```

The next two kernel functions compute the volume of each node from the graph, using EakAccumD and VakAccumD after the layer for loop. ComputeVolumes compute the general case, where radius greater than one because for radius equals one there is a separated ComputeVolumesR1 function which is much simpler.

4.9 Algorithm 9: Kernel function ComputeVolume

```
1. if (thread_index < number of vertices)
2.        firstPos = VakAccumD[thread_index]
3.        lastPos = VakAccumD[thread_index + 1] - 1
4.        for (index i varying from firstPos to lastPos) do
5.                VolumesD[thread_index] += DegreesD[EakAccumD[i]]
6.        end-for
7. end-if
```

4.10 Algorithm 10: Kernel function ComputeVolumeR1

```
1. if (thread_index < number of vertices)
2.        firstPos = VaD[thread_index]
3.        lastPos = VaD[thread_index + 1] - 1
4.        for (index i varying from firstPos to lastPos) do
5.                VolumesD[thread_index] += DegreesD[EakD[i]]
6.        end-for
7. end-if
```

5. PERFORMANCE ANALYSIS

The experiments were carried out on a Linux cluster with an infini-band network channel. The cluster is composed of 4 nodes which are 2 Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz (16 cores per node) with 64GB DDR3 DIMMs memory and a NVIDIA Tesla k20 GPU per node. The experiments with the MPI/OpenMP were compiled with the following products: (1) Portland Group Compiler Collection (pgcc) 14.2, that supports OpenMP 3.0 and OpenACC 2.0 standards; (2) MVAPICH 2.2.0, that fully supports MPI 2.0 standard. The CUDACCER was compiled with Nvidia Compiler Driver version 5.5.0.

In order that we could generate networks for test with known and controlled features, we used a python language software-package known as NetworkX [14]. NetworkX is used for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks [14]. In our experiments, we used networks based on Barbási-Albert (BA) model [15] of sizes 10.000, 100.000, 200.000, 400.000, 600.000 and 1.000.000 vertices. Another parameter that was varied in our tests was the minimum degree of the networks. The minimum degree corresponds to the least amount of edges connected to each vertex in the net. For every size of network (in amount of vertices) aforementioned, we carried out tests with four different values of minimum degree specified in the networks generation: 1, 2, 3, 4.

Another point that must be cleared up about the tests is the used radii. As aforementioned in section 2, the solution proposed in this paper comes from a local strategy for the calculation of centrality. The analysed radius surrounding each vertex is a parameter chosen by the user. According to [8] a radius equal to 2 can ensure a high correlation degree between the centrality ranking produced by DACCER and the one produced by global strategies. This fact lead us to the conclusion that the use of radii greater or equal to 3 are not justifiable in practical terms since very similar results of centrality ranking are produced by radius equal to 2.

Based on that fact, performed tests were focused on radii smaller than or equal to 2. However, the developed solution is general enough to be used with any value of radius.

The tests execution was divided into two steps: the first one was the execution of DACCER with the aforementioned networks on a 32 simultaneous processes OPENMP/MPI environment. The second step was the execution of CUDACCER on an Nvidia Tesla K20 graphic card. The comparative results are shown below.

It is important to say that on the charts below the vertical axis is the log plot of time taken in milliseconds and on the horizontal axis are the numbers of vertices.

5.1. Results using radius equal to zero

The results prove that, on the tested networks, the execution for radii equal to zero on a GPU was in average about two orders faster than the execution on multicore CPU environment. The median speedups of the CUDACCER in relation to the multicore CPU DACCER when the radius is equal to zero are shown on the table 1:

Table 1. Median GPU Speedup for radius equal to zero

Minimum Degree=1	**Minimum Degree=2**	**Minimum Degree=3**	**Minimum Degree=4**
≈ 152x faster	≈ 125x faster	≈ 121x faster	≈ 84x faster

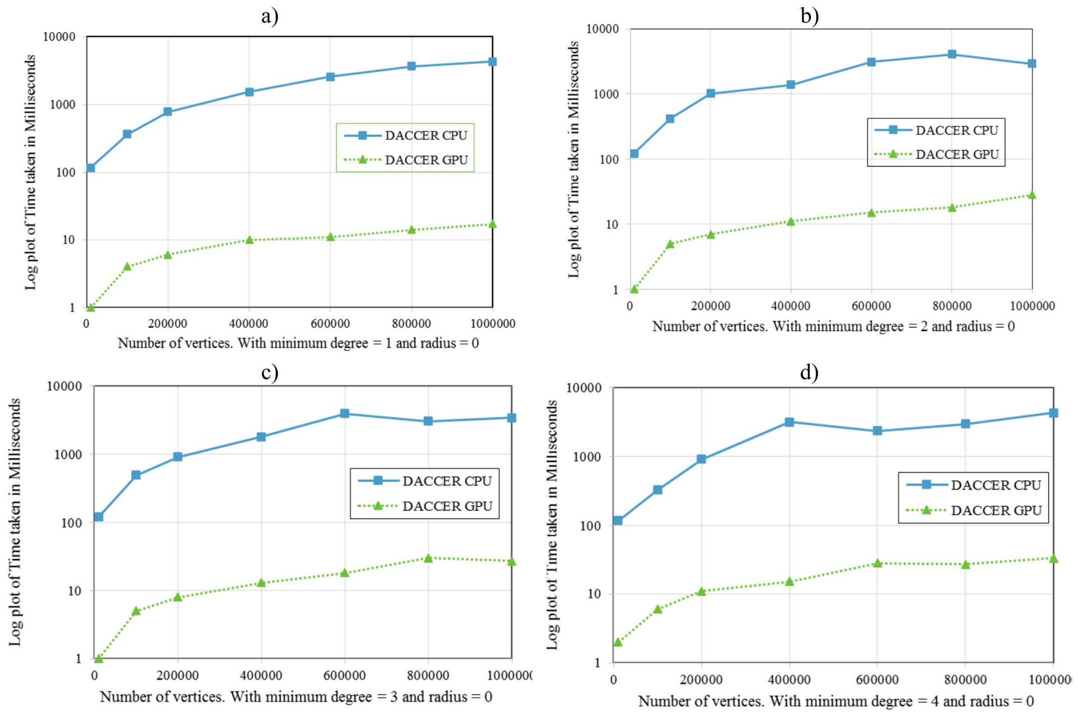


Figure 11. Results using radius equal to 0.

5.2. Results using radius equal to one

The results prove that, on the tested networks, the execution for radii equal to one on a GPU was in average about two orders faster than the execution on multicore CPU environment. The median speedups of the CUDACCER in relation to the multicore CPU DACCER when the radius is equal to one are shown on the table 2:

Table 2. Median GPU Speedup for radius equal to one

Minimum Degree=1	**Minimum Degree=2**	**Minimum Degree=3**	**Minimum Degree=4**
≈ 142x faster	≈ 135x faster	≈ 130x faster	≈ 136x faster

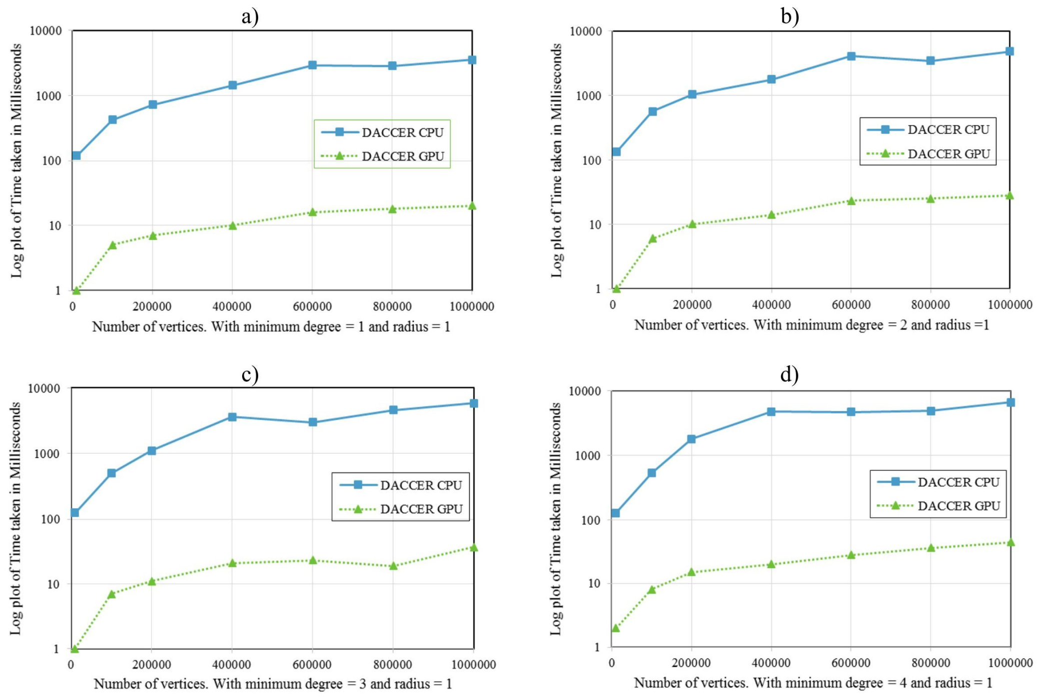


Figure 12. Results using radius equal to 1.

5.3. Results using radius equal to two

The results prove that, on the tested networks, the execution for radii equal to two on a GPU was in average about 5x slower than the execution on multicore CPU environment. The median speedups of the CUDACCER in relation to the multicore CPU DACCER when the radius is equal to two are shown on the table 3:

[

Table 3. Median GPU Speedup for radius equal to two

Minimum Degree=1	**Minimum Degree=2**	**Minimum Degree=3**	**Minimum Degree=4**
≈ 3x slower	≈ 5x slower	≈ 6x slower	≈ 7x slower

Using Nvidia Profiler (nvprof) it's possible to notice that kernel FillEakD consumes up to 98% of execution time when radius equals to two is selected. This disproportional distribution is the main cause of the poor performance result.

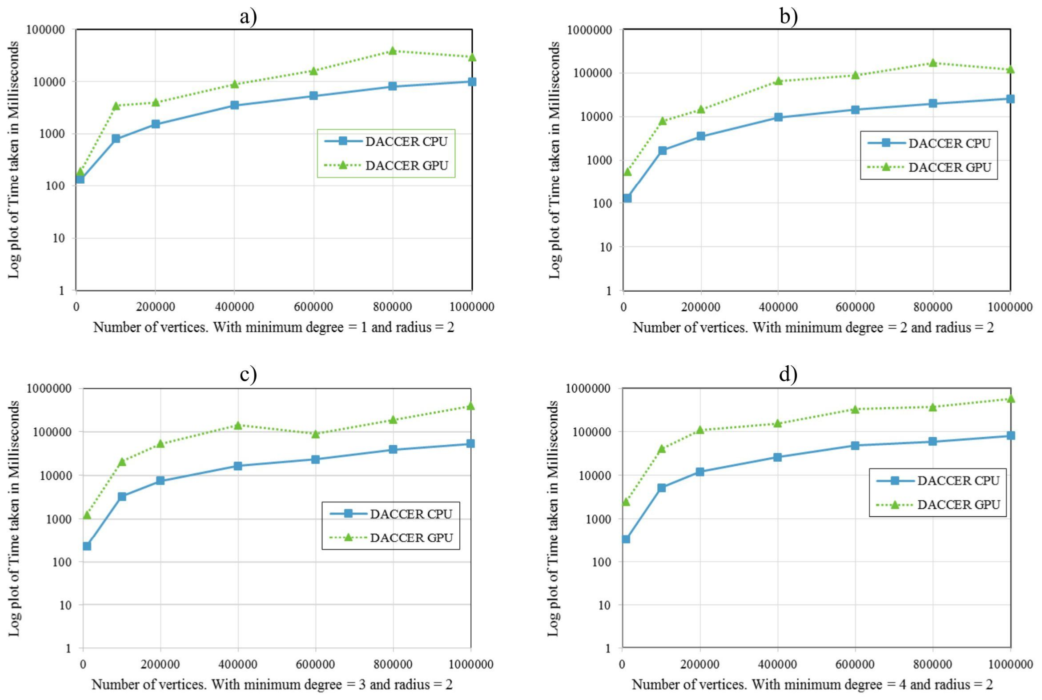


Figure 13. Results using radius equal to 2.

6. CONCLUSIONS

Parallel programming allows one to obtain a gain of performance for many problems as it makes use of multi or many simultaneous threads. CUDA programming allows parallelism to reach a high level speedup as it's based on manycore concept and DACCER is a particular problem benefited from parallelism. This work presents strategies for the development of an algorithm to evaluate the node centrality in complex networks using Nvidia CUDA parallel programming model. We show that the strategies improves algorithm´s speed-up in two orders of magnitude on one NVIDIA Tesla k20 GPU cluster node, when compared to the hybrid OpenMP/MPI algorithm version, running in the same cluster, with 4 nodes 2 Intel(R) Xeon(R) CPU E5-2660 each, for radius zero. The CUDA implementation of DACEER proposed in this paper has speedup by two orders of magnitude for radius zero and radius one, but poor results for radius equals two because the FillEakD kernel function, which presents high computing complexity. There is also a high memory occupation for Cumulative Compact k-Adjacency List. These two issues are to be studied in order to improve speedup results for those cases in future works.

ACKNOWLEDGEMENTS

[1]The authors are thankful to the SINAPAD network (http://www.lncc.br/sinapad) for providing the HPC resources to the experiments described herein.
[2]Authors are thankful to Nvidia for all support.
[3]Partially supported by the “Conselho Nacional de Desenvolvimento Científico e Tecnológico” (CNPq) Brazilian agency.

REFERENCES

[1] Scardoni, Giovanni & Laudanna, Carlo, (2012) "Centralities Based Analysis of Complex Networks", *New Frontiers in Graph Theory,* Dr. Yagang Zhang Edition, InTech, pp323-349.

[2] Newman, M. E. J., (2003) "The structure and function of complex networks", *SIAM Review*, Vol. 45 pp167–256.

[3] Albert, Reka & Barabási, Albert-László, (2002) "Statistical mechanics of complex networks", *Reviews of Modern Physics* Vol. 74, No. 01, pp47-97.

[4] Albert, Reka & Jeong, Hawoong & Barabási, Albert-László, (2000) "Error and attack tolerance of complex networks", *Nature* Vol. 406, pp378–382.

[5] Kermarrec, A. M. & Le Merrer, E. & Sericola, B. & Trédan, G., (2011) "Second order centrality: distributed assessment of nodes criticity in complex networks", *Computer Communications* Vol. 34, Issue 5, pp619–628.

[6] Kim, J.Y., (2010) "Information diffusion and δ-closeness centrality", *Sociological Theory and Methods,* Vol.25, Issue 1, pp95–106.

[7] Kim, H. & Yoneki, E., (2012) "Influential neighbours selection for information diffusion in online social networks", *IEEE International Conference on Computer Communication Networks (ICCCN),* pp. 1–7

[8] Wehmuth, Klaus & Ziviani, Artur, (2013) "Distributed Assessment of the Closeness Centrality Ranking in Complex Networks", *Computer Networks,* Vol. 57, No. 13, pp2536-2548.

[9] Freeman L., (1979) "Centrality in social networks conceptual clarification", *Social Networks* Vol.1, Issue 3, pp215–239.

[10] Cormen, Thomas H. & Leiserson, Charles E. & Rivest, Ronald L. & Stein, Clifford, (2009) *Introduction to Algorithms 3ª Ed.* The MIT Press.

[11] Harish, Pawan & Vineet, Vibhav & Narayanan, P. J., (2009) "Large Graph Algorithms for Massively Multithreaded Architectures", *International Institute of Information Technology Hyderabad.* Report No: III T/TR/74.

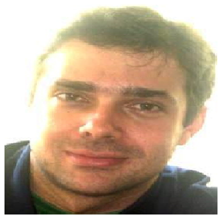
[12] Quinn, M. J., (2004) *Parallel Programming in C with MPI and OpenMP*, McGrawHill.

[13] Kirk, David B. & Hwu, Wen-mei W, (2013) *Programming Massively Parallel Processors: A Hands-on Approach 2nd Ed*, Elsevier, Morgan Kaufmann.

[14] Hagberg, Aric A. & Schult, Daniel A., & Swart, Pieter J. NetworkX. https://networkx.lanl.gov.

[15] Barabási, A. & Albert R., (1999) "Emergence of scaling in random networks", *Science*, Vol. 286 Issue 5439, pp509–512.

Authors



Frederico Luís Cabral received his M.Sc in Computer Science at Universidade Federal Fluminense in 2001 and his B.Sc. at Universidade Católica de Petrópolis. During his graduation worked as assistant researcher in mathematics teaching through computers. His actual interests are in the areas of cluster computing, hybrid computing with CPU/GPU, high performance scientific computing applications, parallel programming and numerical methods for PDEs. He is currently a fellow researcher at the National Laboratory for Scientific Computing (LNCC).

Carla Osthoff, received her B.S. in Electronics Engineering at Pontifícia Universidade Católica do Rio de Janeiro, a M.S. and a Ph.D. in Computer Science from COPPE / Universidade Federal do Rio de Janeiro. Her research interests are in the areas of cluster computing, hybrid computing with CPU/GPU, high performance scientific computing applications and parallel programming. Is currently a researcher at the National Laboratory for Scientific Computing (LNCC) and LNCC High Performance Computing Center coordinator.

Rafael Nardes received his B.S. in Computer Science at UNIFESO, Teresopolis, Brazil. Nowadays he is a student for the M.Sc degree in Scientific Computing at the National Laboratory for Scientific Computing (LNCC), which is a research unit of the Brazilian Ministry of Science, Technology and Innovation (MCTI) located in Petropolis. His research interests are high performance scientific computing applications, GPU computing, problems involving large graphs and Oil & Gas simulations.

Daniel Ramos is an undergraduate senior student in Computer Science at Unifeso, a college located in Teresopolis, Brazil. He is also a scholarship student of scientific initiation at the National Laboratory for Scientific Computing (LNCC), a research unit of the Brazilian Ministry of Science, Technology and Innovation (MCTI) located in Petropolis, Brazil. His current research interest is the use of high performance computing on problems involving large graphs.