The Journal of Supercomputing (2019) 75:1410–1428
https://doi.org/10.1007/s11227-018-2654-5

On parallel computation of centrality measures of graphs

Juan F. García[1] · M. V. Carriegos[1]

Published online: 27 October 2018

Abstract
Centrality measures or indicators of centrality identify most relevant nodes of graphs. Although optimized algorithms exist for computing of most of them, they are still time consuming and are even infeasible to apply to big enough graphs like the ones representing social networks or extensive enough computer networks. In this paper, we present a parallel implementation in C language of some optimal algorithms for computing of some indicators of centrality. Our parallel version greatly reduces the execution time of their sequential (non-parallel) counterpart. The proposed solution relies on threading, allowing for a theoretical improvement in performance close to the number of logical processors (cores) of the single computer in which it is running. Our software has been tested in several platforms, including the supercomputer Calendula, in which we achieved execution times close to 18 times faster when running our parallel implementation instead of our sequential one. Our solution is multi-platform and portable, working on any machine with several logical processor which is capable of compiling and running C language code.

Keywords Parallel computation · High-performance computing · Centrality measures · Network · Graph

This work has been partially supported by the Spanish National Cybersecurity Institute (*Instituto Nacional de Ciberseguridad*, INCIBE). This research uses the resources of the *Centro de Supercomputación de Castilla y León* (SCAYLE, www.scayle.es), funded by the "European Regional Development Fund (ERDF)".

✉ Juan F. García
jfgars@unileon.es

M. V. Carriegos
miguel.carriegos@unileon.es

1 RIASC. Instituto CC. Aplicadas a Ciberseguridad, Universidad de León, León, Spain

1 Introduction

Centrality measures or indicators of centrality identify most relevant nodes of graphs. In this work, we focus on degree centrality, closeness centrality, and betweenness centrality parameters of a given graph.

Although optimized algorithms exist for calculation of most of them, they are still time consuming when dealing big graphs.

Degree centrality (C_d from now on), represented as $C_\mathrm{d}(v) = \deg(v)$, is defined as the number of links incident upon a node. For all the nodes in a graph $G := (V, E)$, computing degree centrality takes $O(|V|^2)$ for a dense adjacency matrix.

Closeness centrality (C_c from now on), represented as $C_\mathrm{c}(x) = \frac{1}{\sum_y d(y,x)}$, is the average length of the shortest path between a given node and all of the others. We can compute shortest paths by means of Dijkstra's algorithm [15], which runs in $O(|V|^2)$ for its sequential implementation.

Betweenness centrality (C_b from now on) quantifies the number of times a node appears in a geodesic of the graph and equals $C_\mathrm{b}(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where σ_{st} is the number of shortest paths from s to t and $\sigma_{st}(v)$ is the number of them that pass through node v. Brandes algorithm [9,10] and its variations allow for optimal computation which runs in $O(|V||E|)$ for unweighted networks; however, the algorithm itself does not benefit much from parallelization [31].

Even when using optimal algorithms to obtain these indicators, their execution time can become infeasible for calculation of indicators for big enough graphs like underlying graphs of social networks or big computer networks. Our work faces this task.

Hence we present a parallel implementation in C language of the optimal algorithms used for calculation of the aforementioned indicators of centrality.

Our parallel implementation greatly reduces the execution time of their sequential (non-parallel) version. Our software is also multi-platform and portable (it runs on Windows and Unix-like systems—Mac OS, Android, Linux…—as well); when code samples are given in this paper, only UNIX version will be depicted for space reasons.

The rest of the paper is organized as follows: In Sect. 2, we briefly summarize our implementation. In Sect. 3, we describe our tests and initial results and calculate relevant metrics. In Sect. 4, we sum up related work. Finally, in Sect. 5, we summarize conclusions and envision future work.

2 Implementation of parallel calculation of indicators of centrality in C language

We have chosen C language since, even nowadays, it is the most relevant high-level programming language for the Engineering Community: According to IEEE, C and C++ languages occupy the second and fourth rank in the Top Ten Programming Languages for 2016 [11]. They are also the only high-level programming languages useful for low-level and embedded programming. If Web programming is not considered, their edge would even be greater. C/C++ power comes especially from their low-level functions and dynamic memory control, which make both capable of unmatched performance results among high-level languages [19].

From a parallel computation point of view, the proposed solution to compute parameters of a given graph relies on threading, allowing for a maximum theoretical improvement in performance equal to the number of logical processors (cores) of the single computer which is running it.

The number of available computation cores of our machine is an upper bound to the number of threads to use for our designed algorithm (creating more threads than cores wastes resources and creates overhead, thus decreasing performance). As long as the program is (at least partially) parallelizable, the more threads you use, the more efficient your algorithm should perform, although the speedup is not linear and will always be limited by the serial part of the program according to Amdahl's law [1]. However, this policy (using the maximum theoretical number of thread) should not force crisped decisions. In fact, you might get better performance by using less threads than cores in some cases [37].

Two levels of parallelization are considered in our application: subgraph-level parallelization and centrality measure-level parallelization.

2.1 Subgraph-level parallelization

Real-world networks rarely have just one single level. (They usually encompass subnetworks—logical subdivision of the network—which can at the same time contain other subnetwork(s) and so on.) Sometimes our networks are stratified. Being able to split their graph representation into all their subgraphs and then independently and in parallel calculate indicators of centrality for each of them can then help us to notably improve calculation performance in a real-world scenario.

Our implementation is able to compute indicators of centrality for several graphs in parallel, no matter they belong to the same network (representing its subnetworks) or are completely independent. This parallel computation level is also independent of the indicators of centrality to be computed.

An specific example of how this parallelization level can be really useful is its application to RDA (random decentering algorithm [42]), in which calculating centrality measures is essential for assessing the quality of the de-centralized network generated; our code allows us to compute these measures in parallel for the adjacency matrix A_t calculated in each algorithm step, greatly improving performance.

```
#ifdef WindowsNoPthreads
DWORD WINAPI calculateBetweennesCentralityThread( LPVOID lpParam )
#else
void* calculateBetweennesCentralityThread(void* lpParam )
#endif
{
    cbAllNodesThreadData* pDataArray = (cbAllNodesThreadData*)lpParam;

    pDataArray->Cb[pDataArray->t] =
    calculateBetweennesCentrality(incidenceMatrixNumberOfNodes -
        nSteps + pDataArray->t);
    releaseSemaphore(&allStepsSemaphore);
}
void getBetweennesCentralityForAllGraphsParallel (float*** Cb) {

    int numberOfIterations = nGraphs + 1;
    cbAllNodesThreadData pDataArray[numberOfIterations];

    #ifdef WindowsNoPthreads
        /* Code omitted for clarity */
    #else
        pthread_t hThreadArray[numberOfIterations];
        initSemaphore(&allStepsSemaphore);
    #endif

    int i = 0;
    int j = 0;

    // Create up to MAX_THREADS worker threads.
    for(i=0; i<numberOfIterations; i++) {
        // Allocate memory for thread data.
        #ifdef WindowsNoPthreads
            /* Code omitted for clarity */
        #else
            pDataArray[i] = malloc(sizeof(cbAllNodesThreadData));
        #endif
        // Generate unique data for each thread to work with.
        pDataArray[i].Cb = (*Cb);
        pDataArray[i].t = i;
        waitForSemaphore(&allStepsSemaphore);
        // Create the thread to begin execution on its own.
        #ifdef WindowsNoPthreads
            /* Code omitted for clarity */
        #else
            pthread_create(
                &hThreadArray[i],
                NULL,
                &calculateBetweennesCentralityThread,
                (void*)&pDataArray[i]);
        #endif
    } // End of main thread creation loop.

    // Wait until all threads have terminated.
    #ifdef WindowsNoPthreads
        /* Code omitted for clarity */
    #else
        for(j=0; j<i; j++)
        {
            pthread_join(hThreadArray[j], NULL);
            free(pDataArray[j]);
        }
    #endif
    destroySemaphore(&allStepsSemaphore);
}
```

Listing 1 Subgraph–level parallelization of C_b calculation for a set of n graphs

Listing 1 shows the extract of our program which calculates C_{b} for n (sub)graphs, independent or not, for Unix-based operating systems where *pthreads* library is available. A thread is created for every graph, and semaphores are used to prevent active threads to ever be greater than the actual number of cores of the machine. C_{b} is then calculated by each thread. (The function used is an implementation of the optimal Brandes algorithm previously introduced.)

Although the code snip is for C_{b}, the C_{d} and C_{c} (and the code for any other centrality measure) is almost identical.

2.2 Centrality measure-level parallelization

Any centrality indicator calculation for a given graph is parallelized by splitting the graph nodes in MAX_THREADS groups.

(MAX_THREADS is the maximum number of simultaneously active threads); each thread will compute the centrality indicator for $\frac{N}{\text{MAX\_THREADS}}$ nodes ($\frac{N}{\text{MAX\_THREADS}}$ is the quotient of the Euclidean division) for threads from 1 to $t-1$, and $\frac{N}{\text{MAX\_THREADS}} + N \bmod \text{MAX\_THREADS}$ nodes (N mod MAX_THREADS is the remainder of the Euclidean division) for the last thread.

The same logic is applied to every centrality measure indicator calculation we have implemented, although performance gains differ for some of them: On the one hand, C_{d} calculation is pretty lightweight on its own, which makes use of threads not useful for small graphs (less than a couple of thousand nodes); C_{c} calculation benefits from using threads sooner (for smaller graphs—a couple of hundred nodes) than C_{d} calculation. In the case of C_{b}, parallelization benefit is negligible due to Brandes algorithm being unparallelizable for the most part.

Note that nodes are split in groups in a way that allows for better complying with the sequential locality principle, a special case of spatial locality (spatial locality refers to the use of data located in relatively close storage locations) which occurs when data elements are arranged and accessed linearly, as it is the case when traversing the elements in a one-dimensional array [36].

In practice, this means that first thread gets assigned nodes from 1 to $\frac{N}{\text{MAX\_THREADS}}$, second thread gets assigned nodes from $\frac{N}{\text{MAX\_THREADS}} + 1$ to $\frac{N}{\text{MAX\_THREADS}} + \frac{N}{\text{MAX\_THREADS}}$, and so on.

Complying with this principle allows our parallel implementation to achieve better results.

```
#ifdef WindowsNoPthreads
DWORD WINAPI calculateClosenessCentralityForNodeGroupThread( LPVOID
    lpParam )
#else
void* calculateClosenessCentralityForNodeGroupThread(void* lpParam )
#endif
{
    ccNodeGroupThreadData* pDataArray =
        (ccNodeGroupThreadData*)lpParam;

    calculateClosenessCentralityForNodeGroup
        (pDataArray->Cc, pDataArray->firstNodeIndex,
         pDataArray->lastNodeIndex, pDataArray->numberOfNodes,
         pDataArray->dist);
}

float* calculateClosenessCentralityParallel(int numberOfNodes) {

    int numberOfIterations = MAX_THREADS;
    int nodesPerIteration = numberOfNodes / MAX_THREADS;
    int nodesOfLastIteration = numberOfNodes
    int firstNodeCurrentIteration = 0;
    int numberOfNodesForCurrentIteration = 0;
    ccNodeGroupThreadData pDataArray[numberOfIterations];

    #ifdef WindowsNoPthreads
        /* Code omitted for clarity *//* Code omitted for clarity */
    #else
        pthread_t hThreadArray[numberOfIterations];
    #endif

    int i = 0;
    int j = 0;

    // Dijkstra is not parallelized in current version
    tAij** dist;
    init2DArray(&dist, numberOfNodes, numberOfNodes);

    // Get shortest paths
    for (i = 0; i < numberOfNodes; i++)
        dijkstra(&incidenceMatrix, numberOfNodes, i, &dist);

    float* Cc = malloc (numberOfNodes * sizeof(float));
    numberOfNodesForCurrentIteration = nodesPerIteration;

    // Create up to MAX_THREADS worker threads.
    for(i=0; i<MAX_THREADS; i++) {

        if (i == MAX_THREADS -1)
            numberOfNodesForCurrentIteration =
                numberOfNodesForCurrentIteration + nodesOfLastIteration;

        // Generate unique data for each thread to work with.
        pDataArray[i].Cc = &Cc;
        pDataArray[i].firstNodeIndex = firstNodeCurrentIteration;
        pDataArray[i].lastNodeIndex = firstNodeCurrentIteration +
            numberOfNodesForCurrentIteration - 1;
        pDataArray[i].numberOfNodes = numberOfNodes;
        pDataArray[i].dist = dist;
        // Create the thread to begin execution on its own.
        #ifdef WindowsNoPthreads
            /* Code omitted for clarity */
        #else
            pthread_create(
                &hThreadArray[i],
                NULL,
                &calculateClosenessCentralityForNodeGroupThread,
                (void*)&pDataArray[i]);
        #endif
            firstNodeCurrentIteration = firstNodeCurrentIteration +
                numberOfNodesForCurrentIteration;

    }// End of main thread creation loop.

    // Wait until all threads have terminated.
    #ifdef WindowsNoPthreads
        /* Code omitted for clarity */
    #else
        for(j=0; j<i; j++)
            pthread_join(hThreadArray[j], NULL);
    #endif

    // Clean
    for ( i = 0; i < numberOfNodes; i++ )
        free(dist[i]);

    free(dist);

    return Cc;

}
```

Listing 2 Centrality measure–level parallelization of C_c calculation

Listing 2 shows the extract of our program which calculates C_c in parallel for a given graph for Unix-based operating systems where *pthreads* library is available. A total of MAX_THREADS threads are created, each of them calculating C_c for a group of nodes following the aforementioned splitting.

Although the code snip is for C_c, the C_d version is almost identical.

3 Tests and initial results

To validate our implementation from an efficacy point of view (to check whether the software does what it is supposed to do), unit, integration and functional testing have been performed. Furthermore, MatlabBGL [20], a third-party MATLAB non-parallel implementation of the same algorithms, has been used to double-check that our results are correct.

Our software can run in any multi-core machine, automatically making use of as many logical cores as it has. To validate our implementation from an efficiency point of view (to check whether the software does what it is supposed to do as fast as it can), our software have been tested in two platforms:

– A high-performance laptop equipped with an Intel Core CPU i7-6820HK @2.7 GHz of 4 cores each (thus totaling 8 cores) with 16 GB RAM memory.
– The Supercomputer Calendula, from SCAYLE (see www.scayle.es). Since our solution focuses on single-node computing resources, the single best machine (as of May 2018) of SCAYLE calculation cluster was used: an Intel Xeon CPU E5-2670 v2 @ 2.50GHz of 10 cores each (thus totaling 20 cores) with 128 GB RAM memory.

Our results are consistent with the theoretical maximum speedup which can be achieved when using multiple processors: The speedup does not depend just of the number of processor, but according to Amdahl's law, it is limited by the serial part of the program.

When running our parallel implementation instead of the sequential one, we achieved execution times close to six times faster in the laptop and close to 18 times faster in the supercomputer in the best scenario (when the task to perform is almost completely parallelizable).

Regarding graph size, the bigger the network, the better performance increase we achieved in general.

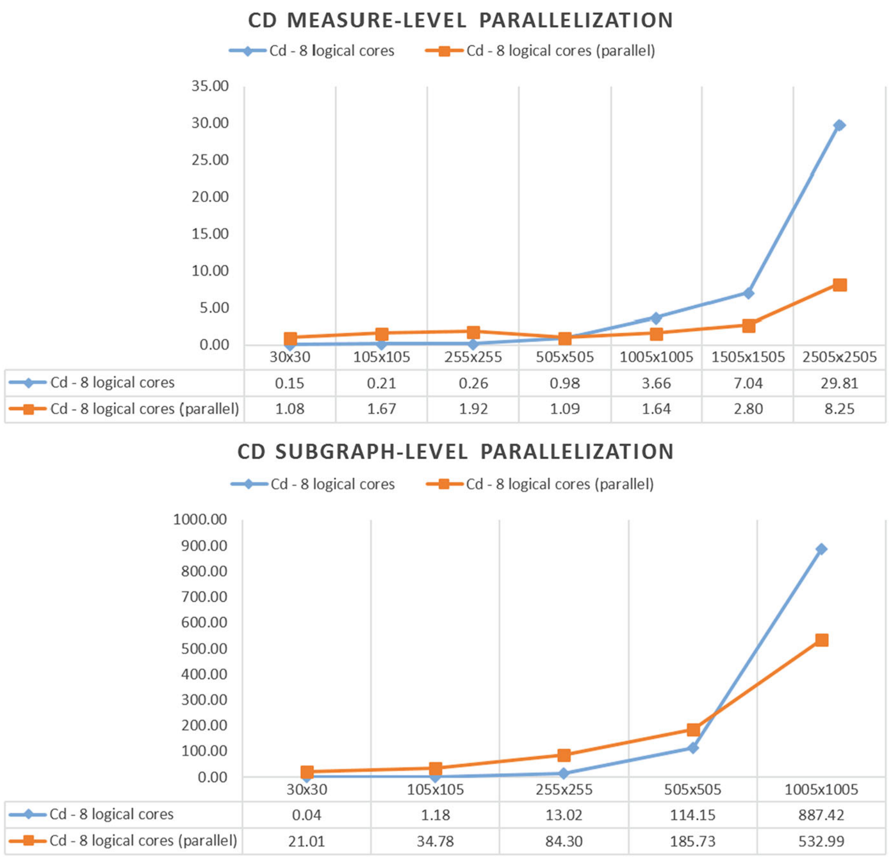
On the one hand, subgraph-level parallelization is usually meaningful even for small (less than a hundred nodes) graphs; on the other hand, centrality measure-level parallelization is not recommended for graphs with less than a hundred nodes (a couple thousand nodes in the case of simpler indicator like C_{d}), since the overhead inherent to thread creation, synchronization, and termination hinders and diminish any performance boost gained.

3.1 Specific centrality measure results

In this section, we provide several pairs of diagrams regarding number of nodes of graphs (dimension of adjacency matrix) versus centrality measure to better illustrate each scenario particularities; for each pair of diagrams, the one on the top corresponds to centrality measure-level parallelization, while the one on the bottom corresponds to subgraph-level parallelization.

We show results for just one of the two testing platforms we used, the high-performance laptop. However, since the performance improvement is linear between the two platforms we used (the high-performance laptop and the supercomputer Calendula), with no anomalies between them, the behavior and the performance improvement ratio shown in these results also apply to the supercomputer. For more details regarding the latter platform, please see next subsection.

Please note x-axis of every diagram represents the dimensions of the matrix, while y-axis represents the time it took to calculate that specific centrality measure; axis labels are omitted for clarity. Also, a data table with the specific execution time values obtained is attached at the bottom of each diagram.



	30x30	105x105	255x255	505x505	1005x1005	1505x1505	2505x2505
Cd - 8 logical cores	0.15	0.21	0.26	0.98	3.66	7.04	29.81
Cd - 8 logical cores (parallel)	1.08	1.67	1.92	1.09	1.64	2.80	8.25

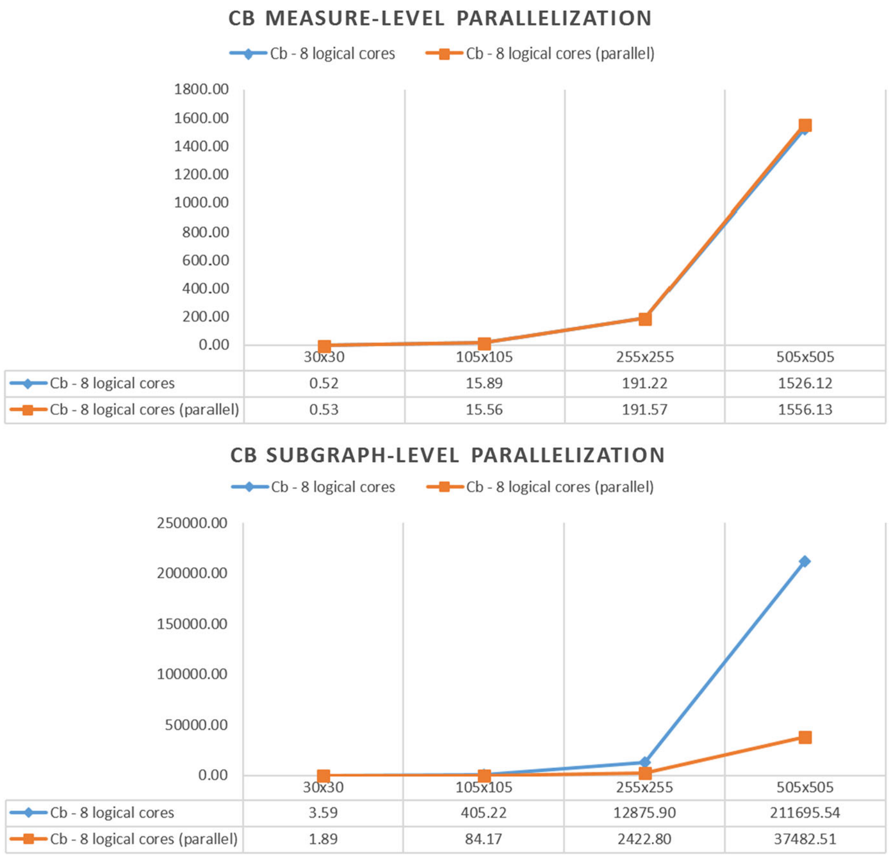
	30x30	105x105	255x255	505x505	1005x1005
Cd - 8 logical cores	0.04	1.18	13.02	114.15	887.42
Cd - 8 logical cores (parallel)	21.01	34.78	84.30	185.73	532.99

Fig. 1 Parallelization of C_d calculation

Figure 1 represents C_d calculation. Given it is one of the most simple ones to calculate, several thousand nodes are necessary for centrality measure-level parallelization to get better results than the regular algorithm. Subgraph-level parallelization does not make up for its overhead for graphs with less than several hundred nodes, although it ramps up fast in benefit after that point.

Figure 2 illustrates C_b calculation. As it was previously explained, C_b centrality measure-level parallelization benefit is almost negligible because most steps of Brandes algorithm are inherently not parallelizable. Note also that even small graphs with just 30 nodes greatly benefit from subgraph-level parallelization, given the complexity of this measure calculation.

Figure 3 depicts C_c calculation. The C_c algorithm parallelizable part is also relatively simple, so it takes several hundred nodes for centrality measure-level parallelization results to get better results than the regular algorithm (although not as much as it takes for C_d). Subgraph-level parallelization is as meaningful as it was for C_b (with better results for small 30×30 matrices), since the whole algorithm (including



	30x30	105x105	255x255	505x505
Cb - 8 logical cores	0.52	15.89	191.22	1526.12
Cb - 8 logical cores (parallel)	0.53	15.56	191.57	1556.13

	30x30	105x105	255x255	505x505
Cb - 8 logical cores	3.59	405.22	12875.90	211695.54
Cb - 8 logical cores (parallel)	1.89	84.17	2422.80	37482.51

Fig. 2 Parallelization of C_b calculation

Dijkstra's algorithm) for C_c calculation is fairly complex from a computational point of view.

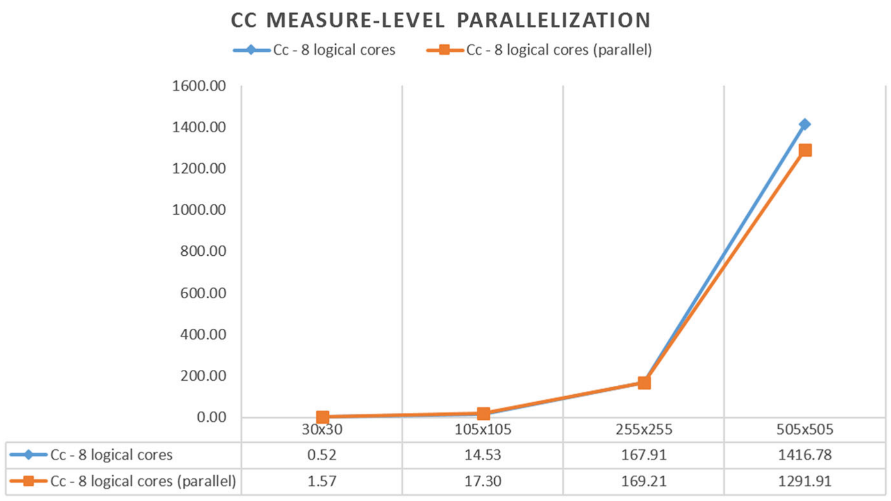
Since we deal with big enough graphs, it follows that parallel version of algorithms always halt and becomes at least as fast as the non-parallel version. Once we get to the point (once we work with a big enough graph) where the parallel version is faster than the regular one, it will always remain so (parallel version will always be faster than the regular one for graphs bigger than the one for which it was initially faster).

For this reason, the graphs shown contain data (x-axis values) only until this tendency or inflexion point (where parallel version becomes faster the regular one) seems clear.

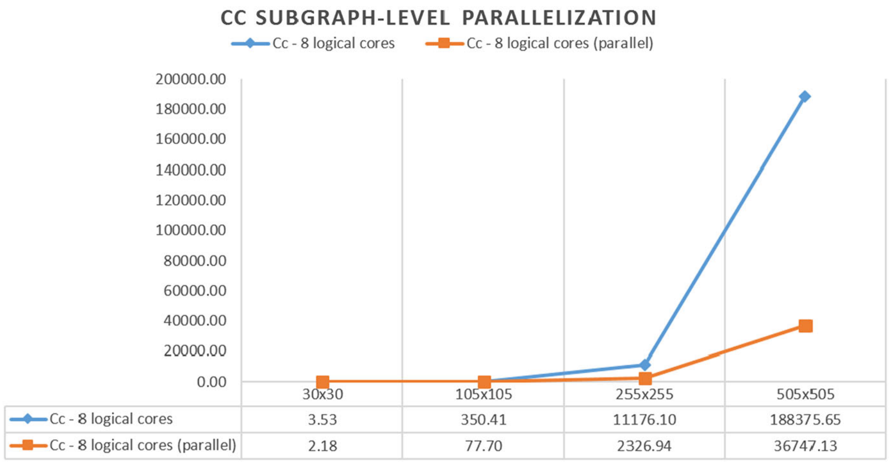
Statistical significance of above rules of thumb is a matter of future work.

3.2 Metrics

In this section we evaluate speedup, efficiency, and scalability of our solution.



	30x30	105x105	255x255	505x505
Cc - 8 logical cores	0.52	14.53	167.91	1416.78
Cc - 8 logical cores (parallel)	1.57	17.30	169.21	1291.91



	30x30	105x105	255x255	505x505
Cc - 8 logical cores	3.53	350.41	11176.10	188375.65
Cc - 8 logical cores (parallel)	2.18	77.70	2326.94	36747.13

Fig. 3 Parallelization of C_c calculation

Parallel speedup $S(p)$ is the fraction of time-to-single-processor-solution $T(n, 1)$ over time-to-parallel-solution $T(n, p)$ for a level p of parallelism, being p in this case the number of logical processors used and n the size of the input [16].

$$S(p) = \frac{T(n, 1)}{T(n, p)}$$

Efficiency $E(p)$ is the ratio of speedup to the number of processors [16], and it measures the fraction of time for which a processor is usefully utilized.

$$E(p) = \frac{S(p)}{p}$$

Table 1 Speedup and efficiency for C_d measure-level (ml) and subgraph-level (sl) parallelization with $p = 8$ and $p = 20$

Indicator	Metric	Network nodes						
		30	105	255	505	1005	1505	2505
C_d ml ($p = 8$)	Speedup	0.14	0.14	0.14	0.90	2.23	2.51	3.61
	Efficiency	0.02	0.02	0.02	0.11	0.28	0.31	0.45
C_d ml ($p = 20$)	Speedup	0.04	0.16	1.00	3.58	6.18	13.20	15.81
	Efficiency	0	0.01	0.05	0.18	0.31	0.66	0.79
C_d sl ($p = 8$)	Speedup	0.01	0.03	0.15	0.61	1.66	2.18	3.20
	Efficiency	0	0	0.02	0.08	0.21	0.27	0.40
C_d sl ($p = 20$)	Speedup	0.01	0.06	0.32	1.53	4.16	7.02	14.18
	Efficiency	0	0	0.02	0.08	0.21	0.35	0.71

Tables 1, 2, and 3 show the speedup and efficiency achieved for both measure-level (ml) and subgraph-level (sl) parallelization with $p = 8$ (which is the number of cores of the high-performance laptop used) and with $p = 20$ (the number of logical cores of the server we used in the supercomputer Calendula).

As expected, speedup and efficiency results obtained for the Supercomputer (20 logical processors) are better or at least equal to the results for the high-performance laptop (8 logical processors). Also, the bigger the network, the better the results for both metrics.

The reason for the latter is that, when processing networks with more nodes, the execution time dedicated to create, synchronize, and destroy threads remains constant, while the execution time specific to the algorithm increases (the ratio of parallelization-specific overhead time to centrality indicator calculation time decreases with network size).

Please note that speedup values bellow 1 mean that the serial version of the program is faster than the parallel one. As it has already been commented at the beginning of Sect. 3, this regularly happens for networks which are not big enough to compensate the parallelization overhead.

Also consider that, as pointed out in Sect. 2, according to Amdahl's law, speedup is not proportional to the number of threads/cores used and is always limited by the serial part of the program; hence, speedup for C_b measure-level parallelization is almost nonexistent (speedup is equal to 1) due to the algorithm being almost completely unparallelizable.

In respect to efficiency, although along with an increase in speedup comes a decrease in efficiency [16], this is true when just increasing the number of processors p while keeping everything else (the problem to be solved and the algorithms to use) constant. To put it simple: Increasing the number of processors (p) decreases efficiency, while increasing the problem size (n) increases efficiency.

The reason that in this case we get increasing efficiency values along with increasing speedup values is that the tables show results obtained for networks of different sizes (with different number of nodes), not results obtained for a different number of

Table 2 Speedup and efficiency for C_b measure-level (ml) and subgraph-level (sl) parallelization with $p = 8$ and $p = 20$

Indicator	Metric	Network nodes				
		30	105	255	505	1005
C_b ml ($p = 8$)	Speedup	0.98	1.01	1.00	0.98	0.99
	Efficiency	0.12	0.13	0.12	0.12	0.12
C_b ml ($p = 20$)	Speedup	1.01	1.02	1.01	1.00	1.00
	Efficiency	0.05	0.05	0.05	0.05	0.05
C_b sl ($p = 8$)	Speedup	1.90	4.81	5.31	5.65	5.92
	Efficiency	0.24	0.60	0.66	0.71	0.74
C_b sl ($p = 20$)	Speedup	3.63	12.38	15.38	16.19	16.93
	Efficiency	0.18	0.62	0.77	0.81	0.85

Table 3 Speedup and efficiency for C_c measure-level (ml) and subgraph-level (sl) parallelization with $p = 8$ and $p = 20$

Indicator	Metric	Network nodes				
		30	105	255	505	1005
C_c ml ($p = 8$)	Speedup	0.33	0.84	0.99	1.10	1.18
	Efficiency	0.04	0.10	0.12	0.14	0.15
C_c ml ($p = 20$)	Speedup	0.63	0.99	1.00	1.20	3.31
	Efficiency	0.03	0.05	0.05	0.06	0.17
C_c sl ($p = 8$)	Speedup	1.62	4.51	4.80	5.13	5.47
	Efficiency	0.20	0.56	0.60	0.64	0.68
C_c sl ($p = 20$)	Speedup	4.78	12.38	15.55	16.43	17.31
	Efficiency	0.24	0.62	0.78	0.82	0.87

processors working with a specific network size: We are varying n, the size of the input, while keeping p constant. If we kept the network (and thus n) and platform constant and started increasing the number of processors p, we would get increasing speedup values along with decreasing efficiency, due to speedup diminishing returns.

Finally, regarding scalability, the results presented confirm that our parallel system is scalable since it can keep efficiency when increasing the number of processors and the problem size simultaneously. (Scalability of a parallel algorithm on a parallel architecture is a measure of its capacity to effectively utilize an increasing number of processors [28].)

As it can be seen in Tables 1, 2, and 3, efficiency is better for the supercomputer ($p = 20$) than it is for the other platform used (with $p = 8$) for networks which are big enough (a couple hundred or thousand nodes depending of the centrality indicator).

As it was described in Sect. 2, our solution automatically splits the task to perform (calculation of a given centrality measure for a graph and/or subgraph(s)) between all available processors, giving each one an equal subset of nodes to deal with. This allows our system to be made cost-optimal by adjusting the number of processors to the problem size.

4 Related work

As we already introduced in Sect. 1, sequential algorithms for computation of centrality measures of graphs (even the optimal ones which run in $O(|V||E|)$) struggle to deal with big enough networks, like the ones representing social networks or extensive computer networks. Parallel computation is a great tool to further push performance of the already optimal sequential algorithms we have at our disposal.

In this work, we presented a solution for parallel computation of degree centrality C_{d}, betweenness centrality C_{b}, and closeness centrality C_{c} indicators of a given graph. There are many works related to parallel computation of centrality measures, although most of them rely on computer clusters or architecture, machine or technology-specific solutions (unlike ours, which makes use of all available logical processors of any machine able to compile and run C language code—which is the case for almost all machines).

For instance, in [4] the authors present parallel algorithms and implementation details of these (among others) centrality metrics on two classes of shared-memory systems: Symmetric multiprocessors (SMPs) such as the IBM p5 575, and multi-threaded 3 architectures such as the Cray MTA. However, we want our solution to be as portable and universal as possible, so we have not considered any machine-specific features.

There are also solutions based on GPU instead of CPU, like [35,38–40], or [33]. We preferred to focus on logical processor parallelization (CPU parallelization) since this way our solution can be applied to almost any machine, even in user environments, instead of requiring a specific graphic card and chip and depend of their manufacturer and the drivers they provide. Also, performance difference between CPU and GPU is close when using proper optimization [29], even more if the task to perform is not fully parallelizable.

Finally, there exist many works regarding parallel calculation of other graph or network-related indicators: Modularity, a popular measure for clustering quality in social network analysis [6,34]; eigenvector centrality and Bonacich's $c(B)$, which can be used in signed and valued graphs [2,8,30]; or Katz centrality, a generalization of C_{d} which measures the number of all nodes that can be connected through a path, while penalizing the contributions of distant nodes [25,26], to name a few. Although they apply to networks and graphs, almost all of them propose computer cluster or machine-specific solutions, and none of them deal with C_{d}, C_{d}, or C_{c} indicators.

In the following subsection, we further detail existing work specifically related to parallel calculation of each of these indicators.

4.1 C_d, C_c, and C_b parallel computation

Both C_{d} and C_{c} are pretty straightforward and lightweight to calculate (see Sect. 2 for details about the approach we followed), both sequentially an in parallel, when compared to C_{b}.

On the one hand, calculating C_{d} using the adjacency matrix is as easy as iterating through its columns (or through its rows for undirected graphs, which we advise to do if that's the type of graph we are working with, since it is computationally faster to do it that way) and adding the elements for each of them. Note that reading elements by rows is faster (for languages which store array rows sequentially in memory) thanks to memory access optimization related to sequential locality principle, which we take into account when splitting nodes in groups for measure-level parallelization, as explained in Sect. 2.2. The sequential locality principle is extensively used in many works, with researchers always looking for ways to either make the best use of or enhance locality [23,24,27,32,43].

On the other hand, and as previously explained in Sect. 1, to calculate C_{c} (defined as the average length of the shortest path between a given node and all of the others) we first need to calculate the shortest paths between all nodes, for which we used Dijkstra's algorithm [15]. Once we got the shortest paths, we calculate their average after applying the splitting mentioned in Sect. 2.2, which once again complies with the aforementioned sequential locality principle as it was the case for C_{d}.

Despite our solution calculates this average value in parallel, our implementation of Dijkstra is sequential, so we could theoretically further improve this part of our proposal by using a parallel version as proposed in [13]. However, we would first have to extend our work to use computer clusters and shared memory. (The proposal in [13] is implemented in shared-memory abstract machine.) On top of that, more memory and computation resources would be required, so it should be carefully evaluated whether the performance improvement compensate for them or not, since we are already parallelizing other parts of the C_{c} calculation algorithm.

There are also alternative to the use of Dijkstra algorithm. We could Floyd–Warshall algorithm [18], which computes the shortest path between all pair of nodes. (Another interesting property of Floyd–Warshall algorithm is that it can be implemented in a distributed system.)

BellmanFord algorithm (slower than Dijkstra but allowing for negative weights) or Johnson's algorithm (which uses Bellman–Ford's algorithm to transform the graph and eliminate negative weighted edges, hence being able to apply Dijkstra afterward) is also an interesting alternative; for a detailed list of algorithms available for shortest paths calculation, including the ones we have just mentioned, please see [12]. There also exist parallel versions (which are approximations in some cases) for these algorithms [3,14].

Regarding optimal computation of C_{b}, we have implemented and adapted the Brandes algorithm presented in [9,10]. Although the algorithm itself does not benefit at all from parallelization [31], thus making our measure-level parallelization benefit negligible, it allows for good results for graph-level parallelization.

On top of that, Brandes proposal was the top choice for us given that, to the best of our knowledge, no parallel and general purpose (not machine, architecture, or network-specific) algorithm exists for C_b computation. For instance, the alternative proposed in [7] seems to be faster, but it applies just in the context of social networks.

Similarly, in [4], authors propose a parallel algorithm for computing betweenness centrality on low-diameter graphs with the same work complexity as Brandes algorithm. However, the memory requirements scale as $O((m+n)p)$ for this approach, where p is the number of processors in the parallel system. The authors also apply their solution to interdisciplinary research in [5], where they compute C_b for a protein interaction network (PIN)—C_b is positively correlated with the essentiality and evolutionary age of a protein.

However, since these works restrict the type and size of networks to which the algorithm can be applied, we prefer to go with the sequential algorithm for our initial proposal. On top of that, their solution relies on using computer clusters, which our implementation does not use in its current version. (We restrict our solution to multiple logical processors of a single machine.)

These are just some of the solutions which use computer clusters to calculate C_b. Some other works in this regard are [41], where authors present a multi-grained parallel algorithm for computing C_b (they use data processor mapping, an edge-numbering strategy, and a new triple array data structure recording the shortest path for eliminating conflicts to access the shared memory) or [21], where authors give a proposal for faster computation of both exact and approximated C_b.

Another interesting work is [17], where authors present a parallelizable algorithm based on a sequential algorithm. Their work requires several processing units (servers) and is suited to distributed memory processing since it is implemented using coarse-grained parallelism.

There are also several other work which adapt existent algorithms to specific computer architectures (they make the algorithm machine-specific to improve performance on a given platform). To do so, they use architecture-specific features or elements which work in parallel; by doing so they can, for instance, parallelize data access, although the algorithm is not parallel per se. (It is the machine which is capable of making specific operation in parallel.)

Examples of the latter are the optimization of betweenness centrality implementations for parallel systems like the Cray XMT, with the massively multithreaded Threadstorm processor, and the multi-core Sun UltraSPARC T2 server, as presented in [31].

As we previously stated, we want our solution to be as portable as possible, so we do not currently consider any machine or architecture-specific solutions.

Also, there is an interesting solution for computing (or, in this case, updating) C_b when new edges are inserted into a graph (to avoid a full re-computation of betweenness centrality) [22], although this falls beyond the scope of our research.

5 Conclusions and future work

We have presented a C language software for parallel computation of centrality measures of graphs. We achieve parallelization at two different levels: Subgraph-level parallelization (meaningful no matter the graph size) and centrality measure-level parallelization (in which the bigger graph is, the better the results are).

Initial results are very promising: Our software have been tested in several platforms, including the supercomputer Calendula, in which we achieved execution times close to 18 times faster when running our parallel implementation instead of the sequential ones.

Our proposal is multi-platform, portable, and greatly improves some of the already optimal sequential algorithms by (partially) computing them in parallel, making use of all logical processors available in the machine running the code. Unlike solutions which depend or rely on machine, architectural or technological-specific features to run in parallel and increase performance, our implementation works on any machine with several logical processor which is capable of compiling and running C language code (which should be most of available machines, both for personal and professional use).

In respect to future work, scalability of our solution could be improved by not always using all available processors. We may not take into account the number of logical cores, but also the network(s) size, to prevent our parallel algorithm to have slower execution times when dealing with small networks (as pointed out in sec:test): Using all available cores can be counterproductive for small graphs due to the overhead inherent to thread creation and synchronization.

The software implemented can be further improved to run on computer clusters: Our current implementation makes use of all available logical processors of a single machine; by means of Message Passing Interface (MPI), we could use computer clusters rather than an single node, further improving the execution times of our approach. In this scenario, some of the algorithms described in Sect. 4 can be useful.

The more intuitive improvement in this case would require migrating subgraph-level parallelization to each separate computer in the cluster while keeping measure-level parallelization at each local machine.

References

1. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the 18–20 April 1967, Spring Joint Computer Conference, ACM, pp 483–485
2. Arefin AS, Berretta R, Moscato P (2013) A GPU-based method for computing eigenvector centrality of gene-expression networks. In: Proceedings of the Eleventh Australasian Symposium on Parallel and Distributed Computing-Volume 140, Australian Computer Society, Inc., pp 3–11
3. Awerbuch B, Bar-Noy A, Gopal M (1994) Approximate distributed bellman-ford algorithms. IEEE Trans Commun 42(8):2515–2517
4. Bader DA, Madduri K (2006) Parallel algorithms for evaluating centrality indices in real-world networks. In: International Conference on Parallel Processing. ICPP 2006. IEEE, pp 539–550
5. Bader DA, Madduri K (2008a) A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms. Parallel Comput 34(11):627–639

6. Bader DA, Madduri K (2008b) Snap, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In: IEEE International Symposium on Parallel and Distributed Processing. IPDPS 2008. IEEE, pp 1–12
7. Baglioni M, Geraci F, Pellegrini M, Lastres E (2012) Fast exact computation of betweenness centrality in social networks. In: Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012). IEEE Computer Society, pp 450–456
8. Bonacich P (2007) Some unique properties of eigenvector centrality. Soc Netw 29(4):555–564
9. Brandes U (2001) A faster algorithm for betweenness centrality. J Math Sociol 25(2):163–177
10. Brandes U, Pich C (2007) Centrality estimation in large networks. Int J Bifurc Chaos 17(07):2303–2318
11. Cass S (2015) The 2017 top programming languages-IEEE spectrum. IEEE Spectrum: Technology, Engineering, and Science News. https://spectrum.ieee.org/computing/software/the-2017-topprogramming-languages. Accessed 10 April 2017
12. Cherkassky BV, Goldberg AV, Radzik T (1996) Shortest paths algorithms: theory and experimental evaluation. Math Program 73(2):129–174
13. Crauser A, Mehlhorn K, Meyer U, Sanders P (1998) A parallelization of Dijkstra's shortest path algorithm. In: International Symposium on Mathematical Foundations of Computer Science. Springer, pp 722–731
14. Davidson AA, Baxter S, Garland M, Owens JD (2014) Work-efficient parallel GPU methods for single-source shortest paths. In: Proceedings of 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE, pp 349–359
15. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numer Math 1(1):269–271
16. Eager DL, Zahorjan J, Lazowska ED (1989) Speedup versus efficiency in parallel systems. IEEE Trans Comput 38(3):408–423
17. Edmonds N, Hoefler T, Lumsdaine A (2010) A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In: 2010 International Conference on High Performance Computing (HiPC), IEEE, pp 1–10
18. Floyd RW (1962) Algorithm 97: shortest path. Commun ACM 5(6):345
19. García JF, Carriegos MV, Balsa J, Sánchez F, Fernández M, Fernández A, Cadenas C, Rodríguez J, Lebedev V (2017) C secure coding standards performance: Cmu sei cert vs misra. In: III Jornadas Nacionales de Investigacion en Ciberseguridad, JNIC2017, Servicio de Publicaciones de la URJC, pp 168–169
20. Gleich D (2008) Matlab bgl. matlab central
21. Green O, Bader DA (2013) Faster betweenness centrality based on data structure experimentation. Proc Comput Sci 18:399–408
22. Green O, McColl R, Bader DA (2012) A fast algorithm for streaming betweenness centrality. In: 2012 International Conference on Privacy, Security, Risk and Trust (PASSAT), and 2012 International Conference on Social Computing (SocialCom), IEEE, pp 11–20
23. Kandemir M, Choudhary A, Ramanujam J, Banerjee P (1998) A matrix-based approach to the global locality optimization problem. In: 1998 International Conference on Parallel Architectures and Compilation Techniques. Proceedings. IEEE, pp 306–313
24. Kandemir M, Ramanujam J, Choudhary A (1999) Improving cache locality by a combination of loop and data transformations. IEEE Trans Comput 48(2):159–167
25. Kang U, Papadimitriou S, Sun J, Tong H (2011) Centralities in large networks: algorithms and observations. In: Proceedings of the 2011 SIAM International Conference on Data Mining. SIAM, pp 119–130
26. Katz L (1953) A new status index derived from sociometric analysis. Psychometrika 18(1):39–43
27. Kowarschik M, Weiß C (2003) An overview of cache optimization techniques and cache-aware numerical algorithms. In: Meyer U, Sanders P, Sibeyn J (eds) Algorithms for memory hierarchies, vol 2625. Springer, Berlin, pp 213–232
28. Kumar VP, Gupta A (1994) Analyzing scalability of parallel algorithms and architectures. J Parallel Distrib Comput 22(3):379–391
29. Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, Satish N, Smelyanskiy M, Chennupaty S, Hammarlund P (2010) Debunking the 100$\times$ GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. ACM SIGARCH Comput Architect News 38(3):451–460
30. Lohmann G, Margulies DS, Horstmann A, Pleger B, Lepsien J, Goldhahn D, Schloegl H, Stumvoll M, Villringer A, Turner R (2010) Eigenvector centrality mapping for analyzing connectivity patterns in fmri data of the human brain. PLoS ONE 5(4):e10232

31. Madduri K, Ediger D, Jiang K, Bader DA, Chavarria-Miranda D (2009) A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In: IEEE International Symposium on Parallel & Distributed Processing. IPDPS 2009. IEEE, pp 1–8
32. Mahapatra NR, Venkatrao B (1999) The processor-memory bottleneck: problems and solutions. Crossroads 5(3es):2
33. McLaughlin A, Bader DA (2014) Scalable and high performance betweenness centrality on the GPU. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press, pp 572–583
34. Newman ME (2006) Modularity and community structure in networks. Proc Natl Acad Sci 103(23):8577–8582
35. Pande P, Bader DA (2011) Computing betweenness centrality for small world networks on a GPU. In: 15th Annual High Performance Embedded Computing Workshop (HPEC)
36. Patterson DA, Hennessy JL, Goldberg D (1990) Computer architecture: a quantitative approach, vol 2. Morgan Kaufmann, San Mateo
37. Pusukuri KK, Gupta R, Bhuyan LN (2011) Thread reinforcer: dynamically determining number of threads via os level monitoring. In: 2011 IEEE International Symposium on Workload Characterization (IISWC). IEEE, pp 116–125
38. Sariyüce AE, Kaya K, Saule E, Çatalyürek ÜV (2013) Betweenness centrality on gpus and heterogeneous architectures. In: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. ACM, pp 76–85
39. Shi Z, Zhang B (2011) Fast network centrality analysis using gpus. BMC Bioinform 12(1):149
40. Sriram A, Gautham K, Kothapalli K, Narayan P, Govindarajulu R (2009) Evaluating centrality metrics in real-world networks on gpu. In: 16th Annual International Conference on High Performance Computing-HiPC 2009 Student Research Symposium. https://hipc.org/hipc2009/documents/HIPCSS09Papers/1569256361.pdf. Accessed 26 Oct 2018
41. Tan G, Tu D, Sun N (2009) A parallel algorithm for computing betweenness centrality. In: International Conference on Parallel Processing. ICPP'09. IEEE, pp 340–347
42. Trobajo M, Cifuentes-Rodríguez J, Carriegos M (2018) On dynamic network security: a random decentering algorithm on graphs. Open Math 16(1):656–668
43. Wong KC, Wu CH, Mok RK, Peng C, Zhang Z (2012) Evolutionary multimodal optimization using the principle of locality. Inf Sci 194:138–170