Critical Node Detection Problem Solving on GPU and in the Cloud

Cholpon Degenbaeva
Saarland University
Saarbruecken, Germany
Email: s9chdege@stud.uni-saarland.de

Matthias Klusch
German Research Center for Artificial Intelligence
Saarbruecken, Germany
Email: klusch@dfki.de

***Abstract*—The Critical Node Detection Problem (CNDP) is a well-known NP-complete, graph-theoretical problem with many real-world applications in various fields such as social network analysis, supply-chain network analysis, transport engineering, network immunization, and military strategic planning. We present the first parallel algorithms for CNDP solving in general, and for fast, approximated CND on GPU and in the cloud in particular. Finally, we discuss results of our experimental performance analysis of these solutions.**

I. Introduction

The critical node detection problem (CNDP) is a well-known NP-complete graph-theoretical problem[1] [3]. In its general version, it is defined as follows: Given an undirected graph $G = (V, E)$ and an integer k, the task is to identify a subset of nodes $A \subseteq V$, such that $|A| \leq k$ and the removal of A results in minimal pairwise connectivity of the residual graph $G(V \setminus A)$. In this sense, the nodes in A are called "critical" to the graph G. The critical node set A is defined as:

$$A = argmin \sum_{i,j \in (V \setminus A)} u_{i,j}(G(V \setminus A)) : |A| \leq k \qquad (1)$$

where $G(V \setminus A)$ is a subgraph induced by removing A from V, and

$$u_{i,j} := \begin{cases} 1, & \text{if i and j are in the same component of } G(V \setminus A), \\ 0, & \text{otherwise.} \end{cases} \qquad (2)$$

The function in (2) is a connectivity measure for pairs of nodes i and j, while the sum in (1) can be rewritten as:

$$\sum_{h \in M} \frac{\sigma_h * (\sigma_h - 1)}{2} \qquad (3)$$

where M is the set of all connected components in $G(V \setminus A)$, and σ_h is the size of component h.

Approximated solutions of the CNDP have many real-world applications in various fields. For example, knowledge about critical nodes can be used to identify most influential actors in social graphs, and to design strategies for vehicle routing, communication breakdowns, or network immunization. Many sequential algorithms for approximated solutions of different CNDP versions have been developed [10], [1], [8], [11], [11], [4], but, to the best of our knowledge, there are no parallel algorithms for solving this NP-complete problem available. On the other hand, parallel computing on GPUs with CUDA (Compute Unified Device Architecture) and in clouds with MapReduce programs became quite popular in the past decade in general, and is quite attractive for the development of fast, approximated CNDP solutions in particular.

To this end, we present the first parallel algorithms for approximated CNDP solving on GPU and in the cloud. Both parallel solutions are inspired by the currently best performing state-of-the-art sequential algorithm CND for heuristic-based solving of the general CNDP in [3]. In particular, we significantly reduced the computational complexity of the CND algorithm, used this new version CND* as basis for the development of parallel CND algorithms in CUDA and MapReduce, and tested their performance on GPU and cloud based on a CNDP test set of Watts-Strogatz graphs and Barabasi-Albert graphs.

The paper is structured as follows: The sequential CND* algorithm is presented in Sect. 2, while its parallized versions in CUDA and MapReduce are described in Sect. 3 and 4, respectively. Results of our experimental performance evaluation are shown in Sect. 5 before we conclude in Sect.6

II. Sequential CND* Algorithm

In this section, we describe our critical node detection (CND*) algorithm for a fast, sequential computation of an approximated solution of the general CNDP. It is an improved version of the currently best performing state-of-the-art algorithm (CND) [3] (cf. Figure 1) for approximated solutions of the general CNDP, and served as a basis for the development of parallel CND algorithms which we describe in subsequent sections.

The CND heuristically computes the maximal independent set (MIS) for a given graph G. Intuitively, the MIS by its nature induces an empty subgraph such that the removal of nodes that are *not* in the MIS of G can lead to a disconnected or empty subgraph, which is what characterizes critical nodes of G. In particular, the CND determines the MIS in a greedy way, and as long as |MIS| ! = $|V| - k$, it greedily selects nodes $v \in V \setminus$MIS for their inclusion in MIS according to the objective function mentioned above, which induces the minimal connectivity score. Since the result is only optimal in cases where the cardinality of the initial MIS is greater or equal to k, a respective optimization procedure "LocalSearch" is subsequentially performed. The overall runtime complexity of the CND algorithm is O(V^2(V+ E)). We modified this CND algorithm into a significantly faster algorithm CND* (cf. Figure 2) with runtime O(V(V+E)) by use of an appropriate Union-Find data structure to speed-up the MIS generation by the original CND algorithm. This data structure supports (a) find-type operations for locating a set corresponding to an ele-

[1]NP-completeness of CNDP has been proven by its reduction to the Vertex Cover problem, or the Independent Set problem.

```
procedure CriticalNode(G,k)
1   MIS ← MaximalIndepSet(G)
2   while (|MIS| ≠ |V| − k) do
3      i ← argmin{ Σ_{h∈M_j} σ_h(σ_h−1)/2 : j ∈ V \ MIS }
4      MIS ← MIS ∪ {i}
5   end while
6   return V \ MIS    /* set of k nodes to delete */
end procedure CriticalNode
```

Fig. 1. Basic CND procedure [3]

ment x and (b) union-type operations for merging two subsets corresponding to x and y into one set. The find-type operations are used by the CND* to keep track in *component*$[node]$ and *sizes*$[component]$ on which component a node belongs to and how many nodes are in a given component, respectively. A component search is performed by the CND* each time it iterates over nodes $v_i \in$ V\MIS as candidates for MIS and uses the component sizes to determine the connectivity score for a candidate node's addition to MIS. Further, the union-type operation *unite()* is performed to merge two components into one with reassigning component ids of the affected nodes in components only after a candidate node has been found; this operation is performed at most O(V - k) times. The CND* uses an implementation of the data structure with O(1) for find-type operations and O(n) for union-type operation.

The MIS is determined by the CND* in a greedy way like by the CND (line 9), i.e. the first node is chosen arbitrarily and added to an empty set, then the CND* keeps adding nodes that are non-adjacent to any of the nodes in the set. In the rest of the paper, we will call the nodes selected to be in the MIS the *selected nodes* (including those added later to the initial MIS) and nodes in the complimentary set the *forbidden (critical) nodes*. All nodes in the initial MIS get unique component ids, their components' sizes are set to one and the number of the forbidden nodes is counted (lines 12-20). Then the CND* compares the number of the forbidden nodes with the given integer k. If there are exactly k forbidden nodes, it returns $V \setminus MIS$ as set of critical nodes. If the number of the forbidden nodes is smaller than k, then most of the nodes are already in MIS and the CND* arbitrarily chooses k - |forbidden nodes| nodes from the already selected ones, and adds them to the set of forbidden nodes (lines 21-28). While the number of forbidden nodes is greater than k, a candidate node is selected from the forbidden nodes that minimizes the connectivity score and added to the set of selected nodes (line 29-35). The while-loop execution has the following invariant:

- For any selected node x : *component[x] != 0*
- For any forbidden node y: *component[y] = 0*
- Let *x* be from selected nodes and $n \in N$ be a number of nodes *z*, such that *component [z] = component[x]*. Then, for any node *x* in set of selected nodes it holds that *sizes [component[x]] = n*.

We provide details of the CND* algorithm in the following. The *next_candidate* procedure of CND* searches for the next candidate node to add to the MIS (Figure 3): It precomputes the connectivity score for the selected nodes without the addition of a new node (lines 6-9). In lines 10-18 it iterates through all nodes and considers each node that has a component id equal to zero (i.e. forbidden nodes). For each such node x the procedure computes the score this node would have if added to selected nodes. It is done with the $score\_with\_node$ (see

```
INPUT: a graph G = (V, E), an integer k
OUTPUT: a set of critical nodes A ⊆ V, such that |A| = k
 1: procedure CND*: CRITICAL NODE DETECTION
 2:     for i =1 to G.size() do                          ▷ Initialization
 3:         component[i] ← 0
 4:     end for
 5:     max_component ← G.size
 6:     for i = 1 to max_component do
 7:         sizes[i] ← 0
 8:     end for
 9:     MIS ← find_mis(G)
10:     component_id ← 1
11:     forbidden_count ← 0
12:     for  i = 1 to G.size() do
13:         if i ∈ MIS then
14:             component[i] ← component_id
15:             sizes[component_id] ← 1
16:             component_id ++
17:         else
18:             forbidden_count ++
19:         end if
20:     end for
21:     if forbidden_count < k then                      ▷ Trivial Case
22:         X ← choose arbitrary k - forbidden_count
23:         nodes ∈ MIS
24:         for (each x ∈ X) do
25:             sizes[component[x]] ← 0
26:             component[x] ← 0
27:         end for
28:     end if
29:     while  forbidden_count > k  do                   ▷ Main Case
30:      cand_node ← next_candidate(G, component, sizes)
31:      united_comp ← any_neighbour_component(G,
32:         cand_node, component)
33:      unite(G, component, sizes, cand_node, united_comp)
34:     forbidden_count −−
35:     end while
36:     return V \ MIS
37: end procedure
```

Fig. 2. Sequential critical node detection algorithm CND*

Figure 4). If a candidate node is added to the set, then all components reachable from it become one *united* component. For computing a new score, the procedure subtracts old score contributions from all of the reachable components, computes a score produced by the *united* component and adds it to the total score. In order to avoid processing the same component twice the procedure marks already processed components. Once the score induced by a candidate node is computed, the *next_candidate* procedure compares it with the current minimal score (line 12, Figure 3) and saves the candidate node and its score only if it is smaller (lines 13-16).

Once the node for addition to MIS is found, the CND* greedily searches for the component id that will be assigned to components that are reachable from it. For this purpose, it iterates over neighbor nodes, checks their component ids and returns the first component id, which is not equal to zero (line 31, Figure 2). Then the found component id is used to unite all affected components and adjust their sizes which is is done by the $unite$ procedure (Figure 5). Let the node

```
INPUT: graph G, component[], sizes[]
OUTPUT: a node that induces a minimal connectivity score for addition to MIS
 1: procedure NEXT CANDIDATE
 2:     min_score ← ∞
 3:     candidate ← -1
 4:     total_score ← 0
 5:     max_component ← G.max_node()
 6:     for c = 1 to max_component do
 7:         total_score ← total_score + sizes[c]∗
 8:                 (sizes[c] − 1)/2
 9:     end for
10:     for node = 1 to G.size() do
11:         if component[node] = 0 then
12:             score ← score_with_node(node, total_score)
13:             if score < min_score then
14:                 min_score ← score
15:                 candidate ← node
16:             end if
17:         end if
18:     end for
19:     return candidate
20: end procedure
```

Fig. 3. Searching for candidate nodes for MIS

```
INPUT: graph G, node x, total_score, sizes[], component[]
OUTPUT: connectivity score for selected_nodes ∪ {x}
 1: procedure SCORE WITH NODE
 2:     new_size ← 1
 3:     for i = 0 to G.neighbours[x].size() do
 4:         neighbour ← G.neigbours[x][i]
 5:         comp ← component[neighbour]
 6:         if comp ≠ 0 and marked[comp] ≠ true then
 7:             marked[comp] ← true
 8:             comp_score ← sizes[comp]∗
 9:                 (sizes[comp] − 1)/2
10:             total_score ← total_score − comp_score
11:             new_size ← new_size + sizes[comp]
12:         end if
13:     end for
14:     result ← new_size ∗ (new_size − 1)/2 + total_score
15:     for i = 1 to G.neighbours[x].size() do
16:         marked[component[G.neighbour[x][i]]] ← false
17:     end for
18:     return result
19: end procedure
```

Fig. 4. Computing of connectivity scores for selected nodes

that was chosen for addition to MIS be the node x: In the initialization phase, procedure *unite* sets the component id of x to be *united_component* and the size of $united\_component$ to one. In order to avoid counting the same component twice, the procedure marks the components affected by the addition of node x (lines 4-11). With such markings done, the procedure iterates over nodes and assigns the affected nodes to the new united component and adjusts sizes of the affected components (lines 12-20).

Lemma 1: Runtime of CND*. The runtime of the CND* is determined by the runtime costs of its while-loop, thus (a) the $next\_candidate$ procedure which is O(V + E); (b) the searching for the component to unite next which takes O(V); and (c) the $unite$ procedure which is also O(V). With at most

```
INPUT: node id x, component id united_component, sizes[], component[], graph G
OUTPUT: adjusts component ids of the affected nodes and sizes of the affected components
 1: procedure UNITE
 2:     component[x] ← united_component
 3:     sizes[united_component] ++
                                          ▷ set markings
 4:     for i = 0; i < G.neighbours.size(); i++ do
 5:         neighbour ← G.neighbours[i]
 6:         component ← component[neighbour]
 7:         if component ≠ 0 ∧ marked[component] ≠ 1
 8:         ∧ component ≠ united_component then
 9:             marked[component] ← 1
10:         end if
11:     end for
                  ▷ adjust component ids of nodes and sizes of components
12:     for node = 1; node ≤ G.size(); node++ do
13:         comp = component[node]
14:         if comp ≠ 0 ∧ marked[comp] = 1
15:         ∧ comp ≠ united_component then
16:             component[node] ← united_component
17:             sizes[comp] − −
18:             sizes[united_component] ++
19:         end if
20:     end for
21:     clear_markings()
22: end procedure
```

Fig. 5. Uniting affected components after node addition to MIS

O(V) times entering the loop, the overal runtime of CND* is O(V(O(V+E) + O(V)+O(V)) = O(V(V+E)).

III. PARALLEL CND* ALGORITHM FOR GPU

Our first parallel version of the CND* algorithm, called pCND*(GPU), makes use of GPU computing under the CUDA model. To realize instruction-level parallelism on many-core GPU, CUDA programs effectively exploit the global and shared memory of thousands of GPU registers which can be partitioned among threads of execution organized and concurrently running within blocks, several caches, and execution cores [12], [6]. A CUDA program consists of instructions written for the host (CPU) and the device (GPU): The host part is typically responsible for allocating/freeing memory on the device and calling a kernel, a GPU-executable program. Once the necessary data has been copied to the device memory, a kernel is launched as a grid of thread blocks. The number of threads per block and the number blocks is predetermined in the host program. The basic strategy of the CUDA program pCND*(GPU) is as follows:

- The given graph G is stored in the global memory and is used with the read-only access by B blocks of T threads. [2]
- Each block gets its own MIS version and its own copies of *component* and *sizes* arrays, which are stored

[2] G is represented as a compressed adjacency list with two arrays: The *to* array stores all lists of adjacent nodes for each node (one after another), and the *start* array which for a given node id shows where its own adjacency list starts. Besides, the *component* array maps each node to its component id, and the *sizes* array stores the size of each component.

in the shared memory.

- Each block employs T number of threads and splits different parallelizable parts of the algorithm between the threads.
- At the end each block will return its own found solution and the algorithm will compare and select the best one from all of the found solutions.

```
INPUT: integer k, graph, component array, sizes array, scores array
OUTPUT: a set of k found critical nodes
 1: procedure pCND*(GPU): CND ON GPU
                                                    ▷ Initialization
 2:     block_id ← current_block.block_id
 3:     component[]← block_id.component
 4:     sizes[]← block_id.sizes
 5:     selected_count ← count_selected_nodes()
 6:     forbidden_count ← |V|- selected_count
 7:     total_score ← compute_total_score()
 8:     if forbidden_count < k then                  ▷ Trivial Case
 9:      remove any (k − forbidden_count) nodes from MIS
10:     end if
11:     while |forbidden_nodes| > k do               ▷ Main Case
12:         next_candidate(total_score, graph, component[],
13:             sizes[], data[], score[] )
14:         sync_threads()
15:         next_node ← minarg(data, score)
16:         total_score ← unite(next_node, total_score,
17:         graph, component[], sizes[], data[],score[])
18:         sync_threads()
19:         forbidden_count ← forbidden_count - 1
20:     end while
21:     return result
22: end procedure
```

Fig. 6. Parallel critical node detection algorithm pCND*(GPU)

We describe details of the pCND*(GPU) algorithm in the following. The host (sequential) part of the algorithm prepares arrays while the pseudocode for the device part (cf. Figure 6) runs concurrently in B blocks of T threads. In lines 2-4 a thread saves a block id, to which it belongs to, and links to the *component* and *sizes* arrays that are shared among threads within the same block of threads. Number of the selected nodes is counted with a helper-function (line5). Each thread iterates over nodes in the graph with an interval of size T. If a node has a component id greater than zero means that it is in MIS and a thread increments its counter. Once a thread is done with its subset of nodes, it writes its result in the thread's field of the shared array. The connectivity score for the set of selected nodes is computed in parallel by T threads that iterate with interval of size T over $sizes[\ ]$. All threads are synchronized and their results are summarized by the aggregate *sum* procedure in a parallel way (cf. Figure 7).

As long as the number of forbidden nodes is bigger than k, the algorithm selects a node from the set of forbidden nodes to be added to the selected ones. For this purpose (a) the set of forbidden_nodes is partitioned into T subsets each assigned to a different thread; (b) each thread i then finds the best candidate node from its subset and writes it together with its score into the i-th elements of two shared arrays (*data* and *score*); and (c) the shared array is reduced by use of the *minarg* procedure to find the best node from the found options.

```
INPUT: a shared array data[] of size T that contains integers to be summed up
OUTPUT: a sum of all integers in data[]
 1: procedure SUM
 2:     thread_id ← threadIdx.x
 3:     synchronize_threads()
 4:     t ← T
 5:     h ← t /2
 6:     while t > 1 do
 7:         if thread_id + h < t then
 8:             data[thread_id] ← data[thread_id] +
 9:                 data[thread_id + h]
10:         end if
11:         t ← t /2
12:         h ← h /2
13:         synchronize_threads()
14:     end while
15:     return data[0]
16: end procedure
```

Fig. 7. Aggregation of results of threads

```
INPUT: a graph, total_score, component[], sizes[], data[], score[]
OUTPUT: a node for addition to selected_nodes with its induced score
 1: procedure NEXT_CANDIDATE
 2:     thread_id ← thread_Idx.x
 3:     min_score ← Integer.Max_Value
 4:     for i = 1 + thread_id; i ≤ graph.size(); i += T do
 5:         node_id ← graph[i]
 6:         if component[node_id] = 0 then
 7:             score ← score_with_node(node_id, total_score,
 8:                 graph, sizes[ ], component[ ])
 9:             if score < min_score then
10:                 min_score ← score
11:                 candidate ← node_id
12:             end if
13:         end if
14:     end for
15:     data[thread_id] ← candidate
16:     score[thread_id] ← min_score
17: end procedure
```

Fig. 8. Selecting next node for addition

For the first two steps, the *next_candidate* procedure (cf. Figure 8) is executed in parallel with T threads and only one best solution is chosen from a block of threads. Each thread iterates over nodes in a graph array with an interval of T. Among those nodes a thread finds those that are in the forbidden nodes (i.e. with $component[i]$ = 0) and for each such node it computes the corresponding score if the node would be added to selected nodes with *score_with_node*. If the found score is smaller than the current value of *min_score*, then the procedure saves the node that produced the score and the score itself. The connectivity score of a node x is computed by *score_with_node* (Figure 9) which iterates over its neighbor nodes and applies a heuristic, that is, if x has a high number of neighbors then it its connectivity score is high. For each adjacent node of x, the procedure checks its component (lines 11-17) and, if the component id has not been processed before, subtracts the component's score contribution from the total score and adds the component's size to the

new_size (lines 20-21). Eventually, the score contributions of all reachable components are subtracted from the *total_score* and the contribution of a new component is added (line 25).

```
INPUT: node x, total_score, a graph, component [], sizes []
OUTPUT: a connectivity score that x induces if added to selected nodes
 1: procedure SCORE_WITH_NODE
 2:     beg ← graph.start[x]
 3:     end ← graph.start[x + 1]
 4:     if (end - beg) > threshold then
 5:         return ∞
 6:     end if
 7:     new_size ← 1
 8:     for i = beg; i < end; i++ do
                                   ▷ Get the adjacent component
 9:         c ← component[graph.to[i]]
10:         if (c != 0) then
              ▷ Check if the component was already processed
11:             boolean was ← false
12:             for j = beg; j < i; j++ do
13:                 if component[graph.to[j]] = c then
14:                     was ← true
15:                     break
16:                 end if
17:             end for
18:             if (!was) then
19:                 s ← sizes[c]
20:                 total_score ← total_score - s * (s −1)/2
21:                 new_size ← new_size + s
22:             end if
23:         end if
24:     end for
25:     result ← total_score+new_size*(new_size −1)/2
26:     return result
27: end procedure
```

Fig. 9. Computing scores for selected nodes

The *unite* procedure (Figure 10) finds and marks those adjacent components that are affected by the addition of a new node by iterating over adjacent nodes with T threads in a parallel manner as before, using the shared *data* and *score* arrays. Each thread uses its *thread_id* to save the sum of sizes of all components that it has processed in the *data* array and the sum of all scores of all components it has processed in the *score* array. Eventually, the sizes of all components affected by the addition of x are summed up in parallel and adds the node x to MIS. The procedure also adds one to the sum's result to reflect the addition of the node x. The result is a size of the newly united component (line 8). The scores of affected components are summed up in parallel (line 9). Then threads iterate over nodes and for each node that belongs to the component that needs to be united, threads reassign the node to the $united\_component$ id (line 10). The components that need to be united are recognized by checking their size, since the affected components have been marked by resetting their sizes to zero. Once all threads have finished their work (line 11), the thread with the *thread_id* of "0" updates the *sizes* array to include a new component with the *united_component* id and with a new size (line 13). It also updates the component id of the newly added node (line 14). After uniting the affected components into one, the procedure returns the updated *total_score* for selected nodes (line 17).

```
INPUT: node x, total_score, a graph, component[], sizes[], data[], score[]
OUTPUT: an updated total_score after the affected components are united
 1: procedure UNITE
 2:     thread_id ← threadIdx.x
 3:     united_component ← any_neighbor_component(x,
 4:         graph, component)
 5:     synchronize_threads()
 6:     reset_sizes_of_neighbor_components(x, graph,
            component, sizes, data, score)
 7:     synchronize_threads()
 8:     new_sizes ← sum(data) +1;
 9:     removed_scores = sum(score)
10:     reassign_components(x, graph.max_node(),
            united_component, component, sizes)
11:     synchronize_threads()
12:     if thread_id = 0 then
13:         sizes[united_component] ← new_sizes
14:         component[x] ← united_component
15:     end if
16:     total_score ← total_score - removed_scores +
            new_sizes ∗ (new_sizes − 1)/2
17:     return total_score
18: end procedure
```

Fig. 10. Uniting of components

Lemma 2: Runtime of pCND*(GPU). Runtime costs of the pCND*(GPU) are dominated by its while-loop in lines 11-18. Since this loop is entered at most V times, the cost is: *O (V · (next_candidate+ unite)) = O (V · (V/ T · max_degree^2 + V · max_degree^2/ T + log(T))) = O (V^2 / T· max_degree^2 + V· log(T))*

IV. Parallel CND* Algorithm for Cloud

Our second parallel version of the CND* algorithm, called pCND*(Cloud), exploits cloud computing [5] with the Apache Hadoop MapReduce model [2]. In the following, we briefly describe how the MapReduce program of pCND*(Cloud) works, and then compare its performance with that of the CND* as baseline and the CUDA version of CND* in the subsequent section.

Mappers. Each of the mappers (cf. Figure 11) executes the main logic of the CND* independently from others. A mapper reads a graph structure and a random seed number from an input file by itself, generates its own MIS using the seed number and runs the CND* algorithm to find its own best solution.

In the initialization phase, each pCND*(Cloud) mapper prepares (a) its own copy of a graph and MIS based on a given random seed from the input file, (b) computes *component* and *sizes* arrays based on its MIS [3], and (c) counts the number forbidden nodes. Then the mapper proceeds with finding critical nodes as in the CND* algorithm. Having found the necessary number of critical nodes, the mapper calculates the connectivity score and outputs its result as a pair, where the first element is this score, and the second element is the array of critical nodes that induce the score.

Reducer. The results from all mappers are accumulated and submitted to the reducer task of the pCND*(Cloud). The

[3]The *component* and *sizes* arrays are used as explained in Sect.2

```
INPUT: graph, integer k and random integer seed for MIS
OUTPUT: (score, k critical nodes)
 1: procedure MAPPER( )                                  ▷ Initialization
 2:     graph ← read_graph()
 3:     random_seed ← read_seed()
 4:     MIS ← generate_MIS(graph, random_seed)
 5:     component[], sizes[], forbidden_count ← set_up()
                                                         ▷ Main Part
 6:     if forbidden_count < k then
 7:       remove any (k − forbidden_count) nodes from MIS
 8:     end if
 9:     while |forbidden_nodes| > k do
10:         next_node ← next_candidate(graph,
11:             component[], sizes[])
12:         united_comp ← any_neighbor_component (
13:                 graph, next_node, component)
14:         unite(next_node, united_component,
15:                 graph, component, sizes)
16:         forbidden_count ← forbidden_count - 1
17:     end while
18:     critical_nodes[] ← V \ selected_nodes
19:     score ← find_score()
20:     return (score, critical_nodes[])
21: end procedure
```

Fig. 11. Mapper function of pCND*(Cloud)

reducer iterates over the first elements of input pairs and searches for the minimal score. Once the reducer has found the minimal score, it outputs the second element that corresponds to the found score; this second element is an array with found critical nodes.

V. Performance Evaluation

We have tested the runtime performance of our sequential and parallel CND* algorithms on CPU, respectively, on GPU and in the cloud with a CNDP test suite of 50 graphs with real-world properties, in particular 25 Barabasi-Albert (BA) and 25 Watts-Strogatz (WS) graph instances. While the BA graph model is used for generating random scale-free networks using a preferential attachment mechanism, the WS model produces graphs with a small-world property and a high clustering coefficient [7]. The sizes of graph instances in the test suite varied from small (100, 500 nodes) to medium (2.000 nodes) and large (10.000, 20.000 nodes), while the number k of critical nodes to find varied from 5 to 5.000. As evaluation baseline, we used the performance of our improved version CND* of the currently best performing state-of-the-art solution CND [3]. The CND* and pCND*(GPU) were tested on a massmarket laptop with an Athlon II X2 240 (2x 2800 MHz, 8GB) CPU and a GeForce GTX 750 GPU (512 cores, 1GB); the pCND*(GPU) run on 8 blocks with 1024 threads in each block. The pCND*(Cloud) was executed with 60 mappers on the Amazon Elastic Cloud (EC2) MapReduce service with one c1.xlarge (8 vCPUs,7GB) namenode and two c3.8xlarge (64 vCPUs,60GB) datanodes. In Figs. 12 and 13, the average runtime performance ratios for GPU- and cloud-based CND* versions with the sequential CPU-based CND* as baseline for WS and BA graph instances (over all k) are summarized. More detailed results are available at http://cndalgorithm.weebly.com/. The basic CND* algorithm appears to be best suited for small-sized BA or WS graphs; in these cases, its parallel version for GPU required more time for memory de/allocation, data tranfer (host/device) and global/shared memory access by threads. For mid-sized graphs, the GPU-based version significantly outperformed the other versions, and was outrun by the cloud-based one first for large WS graphs with 10.000 nodes where full MIS of graphs did not fit into the shared memory of GPU. Finally, for very large graphs of both models with 20.000 nodes, the cloud-based version most significantly outrun its competitor on GPU (up to 14 times faster).

Alg. \ Size	100 node instances	500-node instances	2000-node instances	10000-node instances	20000-node instances
CPU	fastest	fastest	middle	slowest	slowest
GPU	up to 30 times slower	up to 4 times slower	up to 2 times faster	up to 7 times faster	up to 8.5 times faster
Cloud	up to 1192 times slower	up to 97 times slower	up to 7 times slower	up to 5 times faster	up to 14 times faster

Fig. 12. Average runtime ratios for BA graphs with CPU as baseline

Alg. \ Size	100 node instances	500-node instances	2000-node instances	10000-node instances	20000-node instances
CPU	fastest	fastest	middle	slowest	slowest
GPU	up to 15.5 times slower	up to 2 times slower	up to 3.9 times faster	up to 6.9 times faster	up to 8 times faster
Cloud	up to 546 times slower	up to 41 times slower	up to 2.5 times slower	up to 10.7 times faster	up to 22 times faster

Fig. 13. Average runtime ratios for WS graphs with CPU as baseline

VI. Conclusions

We presented the first parallel algorithms for approximated solving of the NP-complete CND problem in general, and for CNDP solving on GPU and in the cloud in particular. Experiments revealed that the cloud-based solution can significantly outperform the currently best sequential, approximated solution [3] of the general CNDP for large-sized graphs, while the GPU-based solution achieves that already for mid-sized graphs. Both parallel versions fail to beat the origin on small graph instances. **Acknowledgment:** This research was supported by the German Ministry for Education and Research (BMBF) in the project INVERSIV under grant number 01IW14004.

References

[1] R. Albert, H. Jeong, A.-L. Barabasi. Error and attack tolerance of complex networks. *Nature*, 406(6794), 2000

[2] Apache Software Foundation. MapReduce tutorial. Website, 2014. hadoop.apache.org/docs/r1.2.1/mapred-tutorial.html

[3] A. Arulselvan, et al.. Detecting critical nodes in sparse graphs. *Computers and Operations Research*, 36(7), 2009

[4] S.P. Borgatti. Identifying sets of key players in a social network. *Computational and Mathematical Organization Theory*, 12(1), 2006

[5] T. Erl, R. Puttini, Z. Mahmood. *Cloud Computing*. Prentice Hall, 2013

[6] R. Farber. *CUDA Application Design and Development*. Elsevier, 2011

[7] M. Edalatmanesh. Heuristics for the Critical Node Detection Problem in Large Complex Networks. MSC. thesis, Brock University, CDN, 2013

[8] J.B. Orlin, et al.. *Network Flows*. Prentice-Hall, 1993

[9] P. Pardalos. Critical Node Detection Problem. Website, 2008 supernet.isenberg.umass.edu/hlogistics/slides/Pardalos-Bellago-Nagurney.pdf

[10] M. Ventresca, D. Aleman. A derandomized approximation algorithm for the critical node detection problem. *Computers and Operations Research*, 43, Elsevier, 2014

[11] J.L. Walteros, P. M. Pardalos. *Applications of Mathematics and Informatics in Military Science*. Springer, 2012

[12] N.Wilt. *The CUDA Handbook*. Pearson Education Inc., USA, 2012