

# Accepted Manuscript

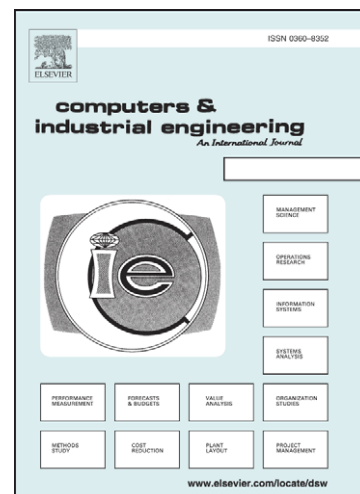
GPU-based Parallel Vertex Substitution Algorithm for the  $p$ -Median Problem

Gino Lim, Likang Ma

PII: S0360-8352(12)00270-7  
DOI: <http://dx.doi.org/10.1016/j.cie.2012.10.008>  
Reference: CAIE 3339

To appear in: *Computers & Industrial Engineering*

Received Date: 13 February 2012  
Revised Date: 30 September 2012  
Accepted Date: 1 October 2012



Please cite this article as: Lim, G., Ma, L., GPU-based Parallel Vertex Substitution Algorithm for the  $p$ -Median Problem, *Computers & Industrial Engineering* (2012), doi: <http://dx.doi.org/10.1016/j.cie.2012.10.008>

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

# GPU-based Parallel Vertex Substitution Algorithm for the $p$ -Median Problem

Gino Lim<sup>1</sup>

Department of Industrial Engineering  
The University of Houston  
Houston, TX, 77204  
713-743-4194  
ginolim@uh.edu

Likang Ma

Department of Industrial Engineering  
The University of Houston  
Houston, TX, 77204  
leondr@gmail.com

---

<sup>1</sup>Corresponding author

# GPU-based Parallel Vertex Substitution Algorithm for the $p$ -Median Problem

---

## Abstract

We introduce a GPU-based parallel Vertex Substitution (pVS) algorithm for the  $p$ -median problem using the CUDA architecture by NVIDIA. pVS is developed based on the best profit search algorithm, an implementation of Vertex Substitution (VS), that is shown to produce reliable solutions for  $p$ -median problems. In our approach, each candidate solution in the entire search space is allocated to a separate thread, rather than dividing the search space into parallel subsets. This strategy maximizes the usage of GPU parallel architecture and results in a significant speedup and robust solution quality. Computationally, pVS reduces the worst case complexity from sequential VS's  $O(p \cdot n^2)$  to  $O(p \cdot (n - p))$  on each thread by parallelizing computational tasks on GPU implementation. We tested the performance of pVS on two sets of numerous test cases (including 40 network instances from OR-lib) and compared the results against a CPU-based sequential VS implementation. Our results show that pVS achieved a speed gain ranging from 10 to 57 times over the traditional VS in all tested network instances.

*Keywords:* Parallel computing, Vertex Substitution,  $p$ -median problem, GPU computation

---

## 1. Introduction

High-performance many-core Graphics Processing Units (GPU) are capable of handling very intensive computation and data throughput. Modern GPUs have achieved higher floating point operation capacity and memory bandwidth when compared to current CPUs at an equivalent level. By taking advantage of CUDA's parallel architecture, remarkable performance improvements have been reported in GPU-based parallel algorithms in various areas of science and engineering [1, 2, 3].

In this paper, we introduce a GPU-based parallel Vertex Substitution algorithm for the unca-

pacitated  $p$ -median problem using the CUDA architecture by NVIDIA. The  $p$ -median problem is well studied in the field of discrete location theory, which includes  $p$ -median problem,  $p$ -center problem, the uncapacitated facility location problem (UFLP) and the quadratic assignment problem (QAP) [4]. The  $p$ -median problem is NP-hard for general  $p$  [5, 6]. Although many heuristic algorithms for solving  $p$ -median problems have been frequently discussed in the literature, computationally efficient GPU-based parallel algorithms are scarce [7, 8]. Most of the reported parallel algorithms achieved parallelism based on two main ideas [9, 10, 11]. The first is to divide the search space into subsets and process each subset simultaneously on each parallel processor to find a solution. For the algorithms whose convergence is sensitive to the initial starting point, the second idea is to generate multiple starting points first and run the algorithm multiple times simultaneously for each starting point. When all solutions are returned, a solution with the best objective value is selected as the optimal solution. Most of those algorithms, however, have been implemented on multi-core Central Processing Unit (CPU). Therefore, the primary goal of this paper is to introduce a computationally efficient GPU-based parallel vertex substitution algorithm to achieve a significant performance gain compared with a well-known CPU-based sequential implementation of Vertex Substitution [12]. In our approach, each candidate solution in the entire search space is allocated to a separate thread, rather than dividing the search space into parallel subsets. This strategy maximizes the usage of GPU parallel architecture and results in a significant speedup and robust solution quality. Computationally,  $p$ VS reduces the worst case complexity from sequential vertex substitution's  $O(p \cdot n^2)$  to  $O(p \cdot (n - p))$  on each thread by parallelizing computational tasks on GPU implementation, where  $n$  is the total number of vertices and  $p$  is the number of medians.

The rest of the paper is organized as follows. In Section 2, we provide an overview of the  $p$ -median problem and its existing solution techniques. Our parallel vertex substitution algorithm is discussed in Section 3 followed by its implementation on NVIDIA CUDA GPU in Section 4. Computational results are discussed in Section 5 followed by the conclusion in Section 6. More computational results are shown in the Appendix section.

## 2. Problem Description and Existing Solution Methods

### 2.1. The $p$ -median Problem

The primary objective of the  $p$ -median problem is to select locations of facilities on a network so that the sum of the weighted distance from demand points to their nearest facility is minimized. The original problem in this class dates back to the 17<sup>th</sup> century [13]. The problem discussed a method to find a median point in a triangle on a two-dimensional plane, which minimizes the sum of distances from each corner point of that triangle to the median point. This problem was extended in the early 20<sup>th</sup> century by adding weights to each of the corner points of a triangle to simulate customer demand. This problem was introduced by Alfred Weber [14] and it is acknowledged as the first location-allocation problem. Generalized problems were developed to find more than one median among more than three points on the plane.

The Weber problem was introduced into a network of graph theory in the Euclidean plane in early 1960s by Hakimi [15, 16]. Hakimi's problem is similar to Weber's weighted problem with the medians being on the network. Hakimi later proved that the optimal median is always located at a vertex of the graph albeit this point is allowed to lie along the graph's edge. This result provided a discrete representation of a continuous problem.

Let  $(\mathbb{V}, \mathbb{E})$  be a connected, undirected network with vertex set  $\mathbb{V} = \{v_1, \dots, v_n\}$  and nonnegative vertex and edge weights  $w_i$  and  $l_{ij}$ , respectively. Let  $\gamma(v_i, v_j)$  be the length (distance) of the shortest path between vertices  $v_i$  and  $v_j$  with respect to the edge weights  $w_i$ , and  $d(v_i, v_j) = w_i \gamma(v_i, v_j)$  be the weighted length (distance) of the corresponding shortest path. Notice  $d$  is not a metric since  $w_i$  is not necessarily  $w_j$ ,  $i \neq j$ . This notation is extended so that for any set of vertices  $X$ , we have  $\gamma(v_i, X) = \min\{\gamma(v_i, x) : x \in X\}$  and  $d(v_i, X) = \min\{d(v_i, x) : x \in X\}$ . With this notation, the  $p$ -median problem can be expressed as

$$\min \left\{ \sum_{v_i \in \mathbb{V}} d(v_i, X) : X \subseteq \mathbb{V}, |X| = p \right\}. \quad (1)$$

The  $p$ -median problem is commonly stated as a binary optimization problem [17]. By defining the

decision variable:

$$\xi_{ij} = \begin{cases} 1 & \text{if vertex } v_i \text{ is allocated to vertex } v_j \\ 0 & \text{otherwise,} \end{cases}$$

a binary formulation is

$$\begin{aligned} z = \min & \quad \sum_{ij} d(v_i, v_j) \xi_{ij} \\ \text{subject to} & \quad \sum_j \xi_{ij} = 1, \text{ for } i = 1, \dots, n, \\ & \quad \sum_j \xi_{jj} = p, \\ & \quad \xi_{jj} \geq \xi_{ij}, \text{ for } i, j = 1, \dots, n, i \neq j, \\ & \quad \xi_{ij} \in \{0, 1\}, \end{aligned} \tag{2}$$

where the objective is to minimize the total travel distance. Three sets of constraints ensure that each vertex is allocated to one and only one median, and that there are  $p$  medians.

## 2.2. Solution Techniques of $p$ -median Problem

Due to the computational complexity, heuristic methods have been a popular choice of solution methods for the  $p$ -median problem [18, 19, 20]. Therefore, we do not attempt to discuss all solution methods, only a few articles that are most relevant to our paper. Recently, CPU-based parallel implementations have been reported in the literature. López *et al* [10] introduced a parallel variable neighborhood search algorithm. Crainic *et al* [9] improved López's method by adding a cooperative feature among parallel computing jobs. López *et al* [11] also proposed a parallel scatter search algorithm. Their implementations usually achieved speedups of 10 or less with eight CPU processors and it is still time consuming when solving problems with more than 1000 vertices and  $p$  larger than 100.

One important and most common solution method for the  $p$ -median problem is Vertex Substitution (VS), which was first developed by Teitz and Bart [12]. The basic idea of VS is to find one vertex which is not in the solution set  $S$  and replace it with one vertex in the solution set to improve the objective value. Different variants of VS have been reported after Teitz and Bart's initial work.

Each method is differentiated by the rule of selecting an entering vertex and a leaving vertex or calculating the gain and loss of swapping two vertices. Lim *et al* [21] show that VS can usually return stable and robust solutions for the  $p$ -median problem. However, it typically takes longer time to converge than other heuristic algorithms such as Discrete Lloyd Algorithm (DLA).

### 2.3. Vertex Substitution

We now briefly describe the Vertex Substitution algorithm that motivated this study. A pseudocode is presented in Algorithm 1 [21]. Let  $\mathcal{S}$  be the candidate solution set and  $N = V \setminus \mathcal{S}$  be the remaining vertices. We consider every vertex  $v_i$  in  $N$  as a candidate median and insert  $v_i$  into  $\mathcal{S}$  to construct a candidate solution set with  $p + 1$  medians. With  $\mathcal{S}_0$  being the initial solution, the gain associated with inserting  $v_i$  into  $\mathcal{S}$  is calculated followed by the respective loss of removing a median vertex  $v_r$  from  $\{\mathcal{S} \cup \{v_i\}\}$ . As a result, a new candidate solution set with  $p$  medians is obtained. The profit  $\pi(v_r, v_i)$  is defined as the gain minus loss for each pair of  $(v_r, v_i)$ . A new solution set is obtained by swapping vertex  $v_r$  with another vertex  $v_i$  based on the profit evaluation approaches. There are two common implementations for evaluating the profit: first profit approach [22] and best profit approach [20]. The first profit approach selects  $(v_r, v_i)$  once the first positive profit is found. The best profit approach evaluates all possible profits and selects  $(v_r, v_i)$  which has the highest profit.

---

**Algorithm 1** Pseudocode for a Vertex Substitution [23]

---

```

1: procedure VERTEXSUBSTITUTION( $\mathcal{S}_0$ )
2:    $q \leftarrow 1$ 
3:    $\mathcal{S}_q \leftarrow \mathcal{S}_0$ 
4:   repeat
5:      $\pi_q^* \leftarrow \max\{\pi(v_r, v_i) : v_r \in \mathcal{S}_q, v_i \notin \mathcal{S}_q\}$ 
6:     if  $\pi_q^* > 0$  then
7:        $\mathcal{S}_{q+1} \leftarrow \{\mathcal{S}_q \cup \{v_i^*\}\} \setminus \{v_r^*\}$ , where  $(v_r^*, v_i^*)$  is a solution to Line 5
8:        $q \leftarrow q + 1$ 
9:     end if
10:  until  $\pi_q^* \leq 0$ 
11: end procedure

```

---

#### 2.4. Parallel Computing on GPUs with NVIDIA CUDA

GPU is the microprocessor which was first designed to render and accelerate 2D and 3D image processing on computers. Since year 2000, the applications of GPU computing occurred in different areas of science and engineering, including finite-element simulation [2], iterative clustering [24], sparse matrix solvers [25] and dense linear system solver [26]. There are several GPU computing architectures available to the public such as NVIDIA CUDA, OpenCL, Direct Computing and AMD Stream. NVIDIA CUDA is a programming architecture for general purpose computation on GPUs with different programming language interfaces, including CUDA C and CUDA Fortran. Currently, CUDA is the main stream framework for GPU-based parallel computing.

CUDA threads are the smallest hardware abstraction that enables developers to control the GPUs multiprocessors. A group of threads is called a block. There is a limit on the number of threads in each block. This is because all the threads in one block are expected to run on the same processor core. Blocks are organized into grids. The number of blocks in a grid is usually determined by the total number of threads or the size of data which needs to be processed. CUDA kernel, which is similar to a function in C programming language, is a group of logic statements. A CUDA kernel can be executed  $m$  times by  $m$  different CUDA threads simultaneously, where  $m$  is the number of parallel tasks to be performed.

There are three levels of memory structure in the CUDA architecture: local, shared, and global. CUDA threads can access data from different memory spaces during execution. Each thread has private local memory which can be accessed by the corresponding thread only. Within each thread block, there is a shared memory which is accessible to all threads in that block and the data in the shared memory has the same lifetime as the block. Global memory is accessible to all threads during the program's lifetime. Shared memory and local memory have ultra fast bandwidth and lowest latency but the size is very limited. Only 16KB is typically available for each block for current NVIDIA GPUs. Different configurations of block size and shared memory usage often result in difference performance due to the hardware execution schedule [27, 28].



### 3. Design of Parallel Vertex Substitution

Our parallel Vertex Substitution method is based on Lim *et al* [21], which is a modified version of Hansen's implementation [20]. The purpose of parallel Vertex Substitution is to fully utilize the power of GPU's many-core architecture and to reduce the solution time required by VS without compromising solution quality. Thus, only the best profit approach is considered in our pVS implementation. The best profit approach evaluates all possible swap pairs before selecting the best pair to swap. The evaluations can be performed independently among swap pairs, which contributes to one of the major sources of task parallelism. When implementing VS with the best profit approach, the algorithm evaluates the total distance between each vertex to its nearest median among all candidate swap pairs  $(v_r, v_i)$  and its worst case complexity is  $O(p \cdot n^2)$ . In order to reduce the solution time, pVS takes advantage of independence of the candidate solution evaluations. Within each iteration, new candidate solutions are obtained by swapping two vertices ( $v_r \in \mathcal{S}$  with another vertex  $v_i \in N$ ) for all possible combinations and associated objective values of new candidate solutions can be evaluated in parallel. The main idea of pVS is to map the evaluation job of each candidate solution to each virtual thread on CUDA GPU and all evaluation jobs are processed on individual threads.

A pseudocode of our parallel Vertex Substitution method is presented in Algorithm 2. Parallel

---

#### Algorithm 2 Pseudocode for Parallel Vertex Substitution

---

```

1: procedure PARALLELVERTICESUBSTITUTION( $\mathcal{S}_0$ )
2:    $q \leftarrow 1$ 
3:    $\mathcal{S}_q \leftarrow \mathcal{S}_0$ 
4:   repeat
5:     Activate thread( $v_r, v_i$ )
6:     thread( $v_r, v_i$ ) evaluates objective value of  $\mathcal{S}_q^{new} = \{\mathcal{S}_q \cup \{v_i^*\}\} \setminus \{v_r^*\}$ 
7:      $\pi_q^* \leftarrow \max\{\pi(v_r, v_i) : v_r \in \mathcal{S}_q, v_i \notin \mathcal{S}_q\}$  by parallel reduction
8:     if  $\pi_q^* > 0$  then
9:        $\mathcal{S}_{q+1} \leftarrow \mathcal{S}_q^{new}$ , where  $(v_r^*, v_i^*)$  is a solution to Line 7
10:     $q \leftarrow q + 1$ 
11:   end if
12:   until  $\pi_q^* \leq 0$ 
13: end procedure

```

---

Vertex Substitution begins with an initial solution  $\mathcal{S}_0$  that was randomly generated. A swap pair

consists of a vertex  $v_i \in N$  and a vertex  $v_r \in \mathcal{S}$  and this pair will be assigned to one thread,  $thread(v_r, v_i)$ . Each  $thread(v_r, v_i)$  then establishes a new solution set  $\mathcal{S}_q^{new} = \{\mathcal{S}_q \cup \{v_i\}\} \setminus \{v_r\}$  and the objective value of each  $\mathcal{S}_q^{new}$  is evaluated on this thread. Profit  $\pi(v_r, v_i)$  is obtained as the objective value of  $\mathcal{S}_q$  minus the objective value of  $\mathcal{S}_q^{new}$ ,  $\forall (v_i, v_r)$ . The best solution is found based on the highest profit by using the parallel reduction method [29]. Parallel reduction is an iterative method which utilizes parallel threads to perform comparisons on elements in one array to find a minimum or maximum value. For example, in maximizing reduction, each thread compares two elements and selects the larger one. Thus, in each iteration, the number of candidate maximizers will be reduced to half of its number in the previous iteration until it narrows down to the last single element.

**Theorem 1.** *The complexity of pVS is  $O(p * (n - p))$ .*

**Proof** Proof of the complexity of pVS consists of three parts. The first step is to group the non-solution set  $N$  into  $p$  clusters. This requires  $p$  comparisons for each vertex and it can be done in  $O(p \cdot (n - p))$ . The next step is to evaluate the objective value of solution candidate on each thread which is obtained by the summation of all distances among those  $n - p$  vertices in set  $N$  with  $O(n - p)$  operations. Third, pVS performs parallel reduction to find the best solution among  $p * (n - p)$  candidate solution sets and it can be done in  $O(\log p * (n - p))$  [29]. Therefore its worst case complexity is  $O(p \cdot (n - p))$ .  $\square$

#### 4. GPU implementation of pVS

The implementation of pVS is achieved by a GPU-CPU cooperation procedure as illustrated in Figure 1. The procedure begins with CPU operations which prepare the distance matrix  $D = \{d(v_i, v_j) \text{ for all } v_i, v_j \in \mathbb{V}\}$  and the initial solution information  $\mathcal{S}_0$ . GPU then executes the main pVS procedures. The results are copied back to CPU for checking the termination condition.

As an initialization step, a pre-generated random solution set  $\mathcal{S}_0$  and the distance matrix  $D$  are loaded into CPU's memory. We assume a strongly connected network with a symmetric distance

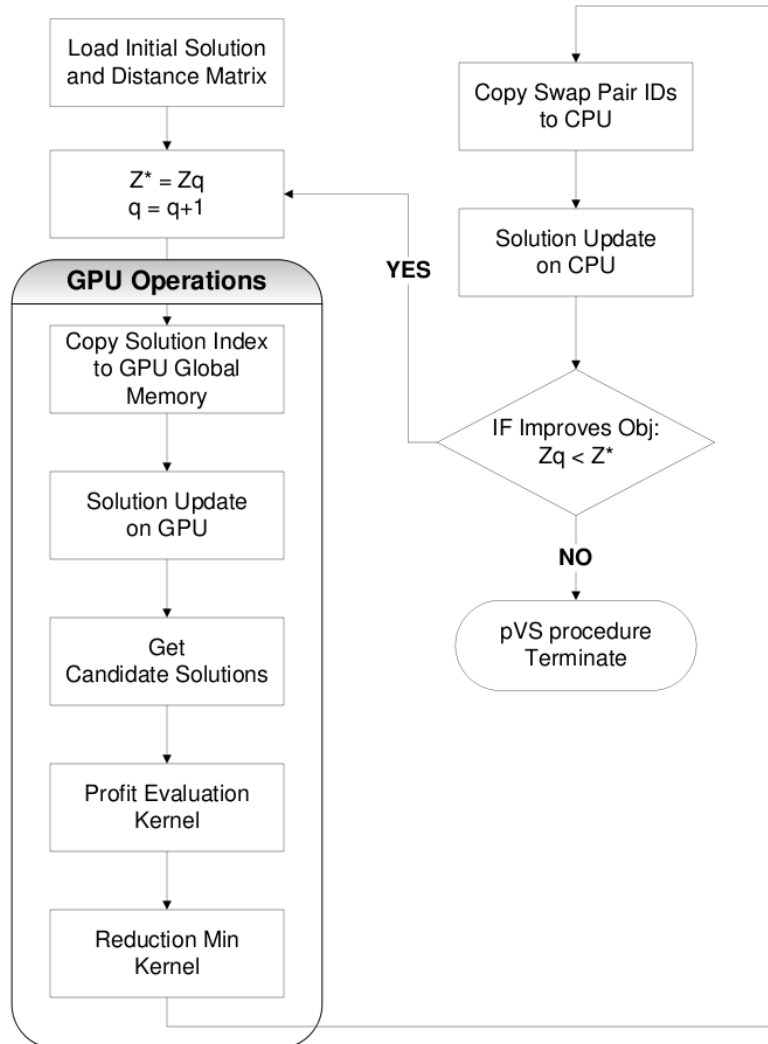


Abbildung 1: pVS GPU implementation flowchart

matrix. The distance matrix is compressed by only storing its upper triangular matrix into a linear memory location. This compressed distance matrix and the pre-generated random solution are then copied to the GPU's global memory.

Most of the computations are carried out by kernels in the GPU Operations module, which includes updating the solution sets on the GPU global memory, getting candidate solution sets, profit evaluation and finding the best pair to swap. Once the best profitable swap pair is found in each iteration, the swap pair of vertices and new objective value are copied back to CPU memory so that the solution information gets updated. The solution update operation is performed on CPU

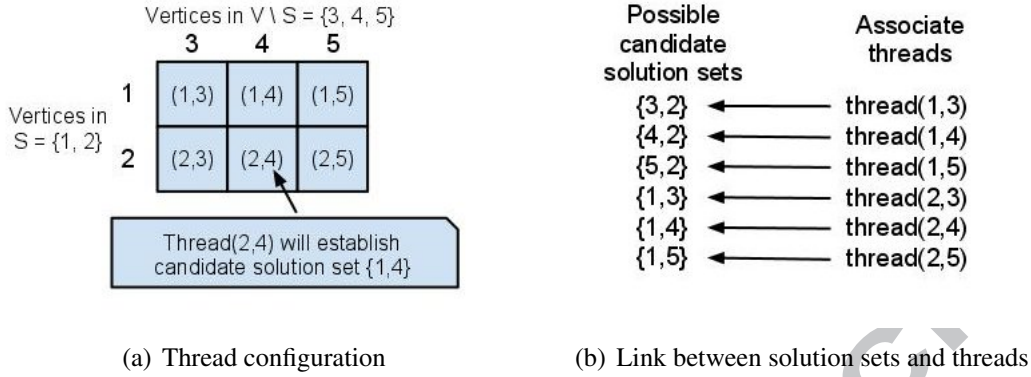


Abbildung 2: pVS representation of candidate solution set

ensuring that the time to copy data between the host and the device memory will be minimized. We now explain the three main GPU operations below.

**Get Candidate Solutions:** The candidate solution sets are generated by using a two dimensional CUDA block on GPU. pVS uses one dimension to represent the set of candidate vertices  $N = V \setminus S$ , and another dimension to represent the current solution set  $S$ . A thread can be identified by thread ID which is associated with a vertex pair  $(v_r, v_i)$ . By swapping the vertices  $(v_r$  and  $v_i)$ , a new candidate solution set is obtained. Since storing a new solution set  $S_{q+1}$  for each thread on the GPU global memory is expensive, the new solution sets are never explicitly generated on the GPU memory. Instead, by referring to the thread's ID, each  $thread(v_r, v_i)$  acquires the information on which vertex should be removed from  $S_q$  and which should be inserted. Thus a candidate solution set can be obtained without storing extra data on the GPU memory and avoid consuming a large amount of memory space. As an illustration, suppose we have a network with five vertices, two medians, its current solution set  $S$  contains vertices 1 and 2, and the remaining set  $N$  contains  $\{3, 4, 5\}$ . In Figure 2,  $thread(2, 4)$  indicates that this thread will replace vertex 2 in  $S$  by vertex 4 in  $N$  to obtain a new candidate solution set  $\{1, 4\}$ . Note that all possible candidate solution sets are established on parallel threads. Once the new candidate solutions are established, pVS evaluates objective values of those candidate solutions using Profit Evaluation Kernel.

**Profit Evaluation Kernel:** The threads and blocks structure of this kernel is inherited from *Get Candidate Solutions*. In this kernel,  $thread(v_r, v_i)$  will loop over the vertices in the entire

network. For each vertex, a nearest median in set  $\mathcal{S}_q^{new}$  and the distance between each vertex to its median is returned. Since no negative path is allowed in the network, if a vertex itself is in the set  $\mathcal{S}_q^{new}$ , it will select itself as the median with zero distance, and thus has no effect to the objective value. The total sum of distances is returned by each  $thread(v_r, v_i)$  and it is the objective value for candidate solution  $\mathcal{S}_q^{new}$ . The maximum profitable (minimum objective value) pair  $(v_r, v_i)$  will be selected by Reduction Min Kernel.

**Reduction Min Kernel:** We now have the objective values for all possible candidate solution on GPU. In order to find a solution with minimum objective value, pVS performs the parallel reduction method that returns the minimum objective value and its solution set. The index of the resulting pair  $(v_r, v_i)$  will be sent back to CPU for updating the new solution set.

The loop in Figure 1 continues until the maximum profit is less than or equal to zero.

## 5. Computational Results

### 5.1. Experiments Setup

In this section, we benchmark the performance of pVS against its CPU counterpart. We denote pVS as the GPU-based parallel VS algorithm and VS as the CPU-based VS algorithm. The specification of CPU we use is an Intel Core2 Quad 3.00 GHz with 4GB RAM and GPU is NVIDIA GTX 285, which has 240 cores with 1.47 GHz and 1GB onboard global memory. CPU Operating System is Ubuntu 10.04LTS and GPU parallel computing framework is NVIDIA CUDA 3.2.

Two sets of problem instances are used to evaluate the performance of our GPU-based parallel algorithms:

(1) OR-Lib  $p$ -median test problems. These test problems were first used by Beasley [30]. The OR-Lib test set includes 40 different undirected networks. As a pre-processing, we generated the shortest path for each pair of vertices using the Dijkstra algorithm [31].

(2) Randomly generated networks. These networks are generated based on the OR-lib test set. Two key network parameters (the number of nodes and the number of medians) are obtained from a network instance from the OR-Lib test set. Given these values, ten new strongly connected network

Tabelle 1: pVS performance on OR-lib  $p$ -median test problems  $p = 5$ 

Problem ID	$n$	Edge	$p$	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
1	100	200	5	5718	5718	0.000	0.000	N/A
6	200	800	5	7527	7527	0.010	0.000	N/A
11	300	1800	5	7578	7578	0.040	0.000	N/A
16	400	3200	5	7829	7829	0.040	0.000	N/A
21	500	5000	5	9123	9123	0.070	0.000	N/A
26	600	7200	5	9809	9809	0.120	0.010	12.0
31	700	9800	5	10006	10006	0.140	0.010	14.0
35	800	12800	5	10306	10306	0.240	0.010	24.0
38	900	16200	5	10939	10939	0.310	0.010	31.0

Tabelle 2: pVS performance on OR-lib  $p$ -median test problems  $p = 10$ 

Problem ID	$n$	Edge	$p$	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
2	100	200	10	4069	4069	0.010	0.000	N/A
3	100	200	10	4250	4250	0.010	0.000	N/A
7	200	800	10	5490	5490	0.050	0.000	N/A
12	300	1800	10	6533	6533	0.090	0.000	N/A
17	400	3200	10	6980	6980	0.180	0.010	18.0
22	500	5000	10	8464	8464	0.370	0.020	18.5
27	600	7200	10	8257	8257	0.430	0.010	43.0
32	700	9800	10	9233	9233	0.620	0.030	20.7
36	800	12800	10	9925	9925	0.730	0.020	36.5
39	900	16200	10	9352	9352	1.120	0.040	28.0

instances are generated by selecting random points on a two dimensional space. Then, the distance matrix is calculated and stored. Since the test data set does not contain large network instances, we generated additional large networks that range from 1000 to 5000 vertices with medians of 10, and 10% and 20% of the total vertices, i.e.,  $p \in \{10, 0.1n, 0.2n\}$ .

## 5.2. Numerical Results

We compare the computational performance of VS and pVS on the OR-Lib networks, and the results are tabulated in Table 1 ( $p = 5$ ) and Table 2 ( $p = 10$ ). Additional results are also provided in Table A.6 through Table A.8 in Appendix. In all tables, *ProblemIDs* are identical to the OR-

lib test problems,  $n$  denotes the number of total vertices in the network,  $Edge$  is the number of edges in the networks, and  $p$  is the number of medians. Objective value is the summation of total distances returned by each algorithm. Solution time records CPU and GPU computation time in seconds. Speedups are calculated as CPU time divided by GPU time,  $speedup = \frac{CPUtime}{GPUtime}$ , and N/A in the speedup column means either time is too short to measure or the iterations did not terminate after five hours.

For all problem instances tested, pVS obtained the same objective values as those of VS, which validates that pVS indeed obtains the same solution quality as VS. Furthermore, pVS outperformed VS in solution time on all problem instances. Table 1 and Table 2 show that when the number of medians is small, pVS works 10 to 40 times faster than VS. As the number of medians increases, pVS takes greater advantage of its parallel design and implementation, and gains substantial speedups as seen in Table A.6 – Table A.8. More results are tabulated in Appendix (Table A.9 – Table A.13) to show the average time consumption on 10 different network instances of each size (combinations of number of  $n$  and  $p$ ).

Figure 3(a) through Figure 3(e) illustrate the time consumption for both pVS and VS. VS run time exhibits an exponential increase as the size of the network increases whereas the increase in run time of pVS is linear with a very small slope as expected. Figure 3(f) displays the average speedups of pVS over VS on different network sizes and different number of medians. The trend of speedup unveils that pVS achieves higher speedups when the problem size and the number of medians are larger and the results are consistent with the results seen in our previous experiments in OR-lib. VS is an efficient algorithm and it still performs reasonably fast for problem instances with less than 1000 nodes. Good solutions could be found by VS in several minutes. When the problem size is beyond 1000 nodes with a larger median value, VS failed to obtain good solutions within a reasonable time. Our experiments clearly show that pVS is a better alternative to VS as it has the ability to solve large problems much faster than VS. Table 3 through Table 5 show the results of our experiments on large problem instances. When  $p$  is small, VS still works well. However, when  $p$  is larger than 10% or 20% of the node size, VS usually could not solve the problem instances

within five hours. On the contrary, pVS is capable of solving those problem in a few minutes. In Table 4, VS solved a problem instance ( $n = 3000, p = 300$ ) in about 4.2 hours whereas pVS found the same solution in less than 5 minutes. In the largest problem we tested ( $n = 5000, p = 1000$ ), the estimated solution time of VS is roughly 5 days (based on the number of iterations evaluated in 5 hours) whereas pVS found the solution in about two hours.

Tabelle 3: pVS performance on large network problems  $p = 10$ 

Problem ID	$n$	$p$	Objective Value		Solution Time		Speedups
			VS	pVS	VS	pVS	
0	1000	10	364.80	364.80	2.82	0.10	28.2
3	2000	10	754.68	754.68	20.58	0.58	35.5
6	3000	10	1090.53	1090.53	77.92	1.85	42.1
9	4000	10	1475.80	1475.80	89.04	2.09	42.6
12	5000	10	1749.92	1749.92	130.74	2.98	43.9

Tabelle 4: Average pVS performance on large network problems,  $p = 0.1 * n$ 

Problem ID	$n$	$p$	Objective Value		Solution Time		Speedups
			VS	pVS	VS	pVS	
1	1000	100	23.28	23.28	155.72	3.20	48.7
4	2000	200	23.46	23.46	2905.56	53.28	54.5
7	3000	300	23.68	23.68	15351.30	270.94	56.7
10	4000	400	N/A	23.08	> 5 hours	822.38	N/A
13	5000	500	N/A	22.34	> 5 hours	2103.91	N/A

Tabelle 5: Average pVS performance on large network problems,  $p = 0.2 * n$ 

Problem ID	$n$	$p$	Objective Value		Solution Time		Speedups
			VS	pVS	VS	pVS	
2	1000	200	7.72	7.72	492.61	10.10	48.8
5	2000	400	8.13	8.13	9036.87	159.33	56.7
8	3000	600	N/A	7.77	> 5 hours	837.12	N/A
11	4000	800	N/A	7.92	> 5 hours	2606.40	N/A
14	5000	1000	N/A	35.78	> 5 hours	7405.54	N/A

Overall, the performance improvements are contributed by both the parallel algorithm and the GPU device. The CPU we used (3.0 GHz with 4GB RAM) is faster per clock speed than the GPU



(1.47GHz with 1GB RAM). In general, CPU's memory can be expanded to a large size whereas GPU's memory is fixed without the ability to expand further. But the speedups of GPU come from its many-core parallel architecture (240 CUDA cores on our device) where multiple independent jobs can be processed from different threads at the same time.

The pVS algorithm is specifically designed to best use GPU's architecture by distributing fractionated and less complex tasks to independent cores in parallel. As we mentioned earlier, VS is a reliable sequential algorithm for the p-median problem, but it is a bit slow for larger problems. As an alternative, pVS is a parallel algorithm that significantly reduces the computational complexity by utilizing multiple threads per iteration.

ACCEPTED MANUSCRIPT

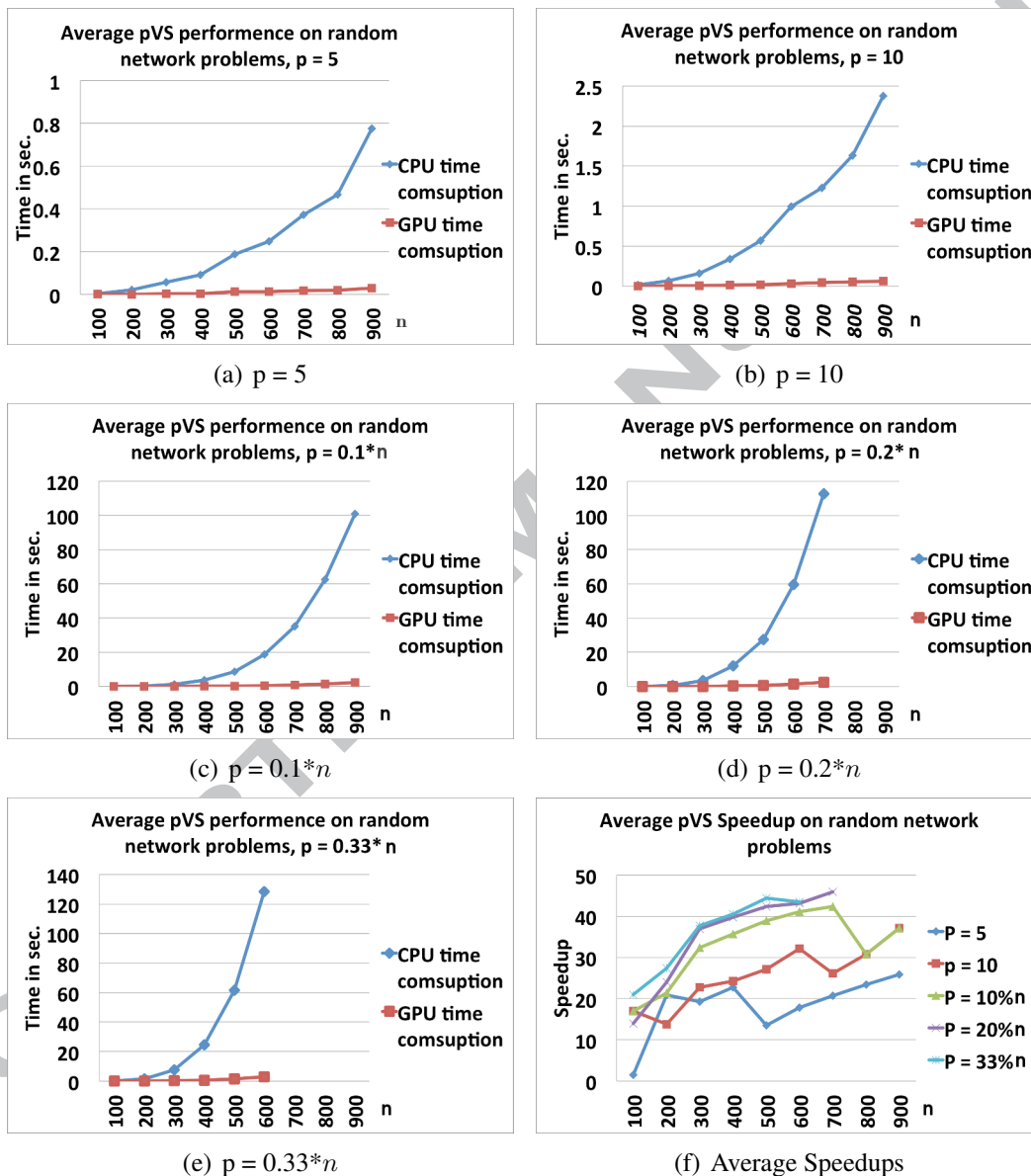


Abbildung 3: Average pVS performance on random network problems

## 6. Conclusion

We have developed a GPU-based parallel Vertex Substitution algorithm for the  $p$ -Median problem on the NVIDIA CUDA GPU architecture. We designed a GPU-CPU cooperation procedure for pVS which used GPU to compute expensive pVS operations in parallel and used CPU to coordinate the iterations and the termination of the algorithm. The worst case complexity of pVS is  $O(p \cdot (n - p))$  which is a reduction from  $O(p \cdot n^2)$  of the sequential vertex substitution. We tested the performance of pVS on two sets of test cases and compared the results with a CPU-based sequential vertex substitution implementation (best profit search). Those two test data sets include 40 different network instances from OR-lib, 400 similar randomly generated network instances, and 15 randomly generated large network instances. The pVS algorithm on GPU ran significantly faster than the CPU-based VS algorithm in all test cases. In small network instances such as OR-lib problems and 400 similar randomly generated network instances, pVS obtained 10 to 40 times speedups. The speed gain was more substantial for larger network instances having more than 1000 nodes with a larger value of median by observing a speed gain of 28 to 57 times. This is particularly important because CPU-based VS could not solve some of the large problem instances within five hours, while the GPU-based pVS solved all instances in two hours or less. A direction of future research is to use multi-start technique to further improve pVS's solution quality and to combine pVS with other heuristic algorithms to further boost the solution time.

## Literatur

- [1] M. J. Harris, W. V. Baxter, Simulation of cloud dynamics on graphics hardware., in: ACM AIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2003, pp. 92–101.
- [2] U. Diewald, T. Preußner, M. Rumpf, R. Strzodka, Diffusion models and their accelerated solution in computer vision applications, *Acta Mathematica Universitatis Comenianae* 70 (1) (2001) 15–31.
- [3] C. Men, X. Gu, D. Choi, H. Pan, A. Majumdar, S. B. Jiang, GPU-based ultra-fast dose cal-

- ulation using a finite size pencil beam model, *Physics in medicine and biology* 54 (2009) 6287–6297.
- [4] P. B. Mirchandani, R. Francis, *Discrete Location Theory*, John Wiley & Sons, New York, 1990.
- [5] M. R. Garey, D. S. Johnson., *Computers and Intractability: A guide to the theory of NP-completeness*, W. H. Freeman and Co., San Francisco, 1979.
- [6] O. Kariv, S. L. Hakimi., An algorithmic approach to network location problems, *SIAM Journal on Applied Mathematics* 37 (3) (1979) 539–560.
- [7] L. Ma, G. Lim, GPU-based parallel computational algorithms for solving  $p$ -median problem, in: *Proceedings of the IIE Annual Conference*, 2011.
- [8] L. F. M. Santos, D. Madeira, E. Clua, S. Martins, A. Plastino, A parallel GRASP resolution for a GPU architecture, in: *International Conference on Metaheuristics and Nature Inspired Computing*, 2010.
- [9] T. G. Crainic, M. Gendreau, P. Hansen, N. Mladenović, Cooperative parallel variable neighborhood search for the  $p$ -median, *Journal of Heuristics* 10 (3) (2004) 293–314.
- [10] F. García-López, B. Melián-Batista, J. A. Moreno-Pérez, J. M. Moreno-Vega, The parallel variable neighborhood search for the  $p$ -median problem, *Journal of Heuristics* 8 (2002) 375–388.
- [11] F. García-López, B. Melián-Batista, Parallelization of the scatter search for the  $p$ -median problem, *Parallel Computing* 29 (5) (2003) 575–589.
- [12] M. B. Teitz, P. Bart, Heuristic methods for estimating the generalized vertex median of a weighted graph, *Operations Research* 16 (5) (1968) 955–961.
- [13] J. Reese, Solution methods for the  $p$ -median problem: An annotated bibliography, *Networks* 48 (3) (2006) 125–142.

- [14] C. J. Friedrich, Alfred Weber's Theory of the Location of Industries, University of Chicago Press, Chicago, 1929.
- [15] S. L. Hakimi, Optimum locations of switching centers and the absolute centers and medians of a graph, *Operations Research* 12 (3) (1964) 450–459.
- [16] S. L. Hakimi, Optimum distribution of switching centers in a communication network and some related graph theoretic problems, *Operations Research* 13 (3) (1965) 462–475.
- [17] C. Revelle, R. Swain, Central facilities location, *Geographical Analysis* 2 (1970) 30–42.
- [18] F. E. Maranzana, On the location of supply points to minimize transport costs, *Operations Research Quarterly* 15 (3) (1964) 261–270.
- [19] A. W. Neebe, M. R. Rao, A subgradient approach to the  $m$ -median problem, Tech. Rep. 75-12, University of North Carolina, Chapel Hill, N.C. (1975).
- [20] P. Hansen, N. Mladenović, Variable neighborhood search for the  $p$ -median, *Location Science* 5 (4) 207–226.
- [21] G. Lim, J. Reese, A. Holder, Fast and robust techniques for the euclidean  $p$ -median problem with uniform weights, *Computers & Industrial Engineering* 57 (2009) 896–905.
- [22] R. A. Whitaker, A fast algorithm for the greedy interchange of large-scale clustering and median location problems, *INFOR* 21 (2) (1983) 95–108.
- [23] M. G. C. Resende, R. F. Werneck, On the implementation of a swap-based local search procedure for the  $p$ -median problem, in: R. Ladner (Ed.), *ALENEX '03: Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments*, SIAM, Philadelphia, 2003, pp. 119–127.
- [24] J. D. Hall, J. C. Hart, Abstract GPU acceleration of iterative clustering, *The ACM Workshop on General Purpose Computing on Graphics Processors*, and *SIGGRAPH 2004* (Aug. 2004).

- [25] J. Bolz, I. Farmer, E. Grinspun, P. Schroder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, *ACM Trans. Graph.* 22 (3) (2003) 777–786.
- [26] N. Caloppo, N. K. Govindaraju, M. Henson, D. Manocha, Efficient algorithms for solving dense linear systems on graphics hardware, in: *ACM IEEE SC 05 Conference*, 2005.
- [27] N. Bell, M. Garland, Efficient sparse matrix-vector multiplication on CUDA, Tech. Rep. NVR-2008-04, NVIDIA Corporation (2008).
- [28] R. Vuduc, A. Chandramowlishwaran, J. W. Choi, M. E. Guney, A. Shringarpure, On the limits of GPU acceleration, in: *Proc. USENIX Wkshp. Hot Topics in Parallelism (HotPar)*, Berkeley, CA, USA, 2010.
- [29] M. Harris, Optimizing parallel reduction in CUDA, Tech. rep., NVIDIA Developer Technology (2007).
- [30] J. E. Beasley, A note on solving large  $p$ -median problems, *European Journal of Operational Research* 21 (1985) 270–273.
- [31] E. W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik* 1 (1959) 269–271, 10.1007/BF01386390.  
URL <http://dx.doi.org/10.1007/BF01386390>

## Anhang A. pVS Performance Comparison Tables

Tabelle A.6: pVS performance on OR-lib  $p$ -median test problems  $p = 0.1 * n$ 

Problem ID	$n$	Edge	$p$	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
2	100	200	10	4069	4069	0.010	0.000	N/A
3	100	200	10	4250	4250	0.010	0.000	N/A
8	200	800	20	4410	4410	0.180	0.010	18.0
13	300	1800	30	4357	4357	0.690	0.020	34.5
18	400	3200	40	4643	4643	2.480	0.070	35.4
23	500	5000	50	4503	4503	5.900	0.170	34.7
28	600	7200	60	4447	4447	12.100	0.330	36.7
33	700	9800	70	4628	4628	24.580	0.680	36.1
37	800	12800	80	4986	4986	39.820	1.090	36.5
40	900	16200	90	5055	5055	0.630	0.010	63.0

Tabelle A.7: pVS performance on OR-lib  $p$ -median test problems  $p = 0.2 * n$ 

Problem ID	$n$	Edge	$p$	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
4	100	200	20	2999	2999	0.030	0.000	N/A
9	200	800	40	2709	2709	0.500	0.020	25.0
14	300	1800	60	2919	2919	2.100	0.060	35.0
19	400	3200	80	2823	2823	8.130	0.220	37.0
24	500	5000	100	2892	2892	19.270	0.510	37.8
29	600	7200	120	3006	3006	38.680	1.010	38.3
34	700	9800	140	2939	2939	65.890	1.750	37.7

Tabelle A.8: pVS performance on OR-lib  $p$ -median test problems  $p = 0.33 * n$ 

Problem ID	$n$	Edge	$p$	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
5	100	200	33	1355	1355	0.060	0.000	N/A
10	200	800	67	1247	1247	0.880	0.030	29.3
15	300	1800	60	2919	2919	2.100	0.060	35.0
20	400	3200	133	1781	1781	14.140	0.400	35.4
25	500	5000	167	1828	1828	36.460	1.000	36.5
30	600	7200	200	1966	1966	76.840	2.000	38.4

Tabelle A.9: Average pVS performance on randomly generated networks  $p = 5$ 

Problem ID	N	$p$	Objective Gap $ z_{pvs} - z_{vs} $	Average Solution Time		Speedups
				VS	pVS	
1	100	5	0	0.003	0.002	1.5
6	200	5	0	0.021	0.001	20.9
11	300	5	0	0.058	0.003	19.3
16	400	5	0	0.091	0.004	22.8
21	500	5	0	0.189	0.014	13.5
26	600	5	0	0.249	0.014	17.8
31	700	5	0	0.373	0.018	20.7
35	800	5	0	0.468	0.020	23.4
38	900	5	0	0.776	0.030	25.9

Tabelle A.10: Average pVS performance on randomly generated networks  $p = 10$ 

Problem ID	$n$	$p$	Objective Gap $ z_{pvs} - z_{vs} $	Average Solution Time		Speedups
				VS	pVS	
2	100	10	0	0.015	0.003	5.0
3	100	10	0	0.017	0.001	17.0
7	200	10	0	0.069	0.005	13.8
12	300	10	0	0.159	0.007	22.7
17	400	10	0	0.340	0.014	24.3
22	500	10	0	0.569	0.021	27.1
27	600	10	0	0.997	0.031	32.2
32	700	10	0	1.228	0.047	26.1
36	800	10	0	1.633	0.053	30.8
39	900	10	0	2.377	0.064	37.1



Tabelle A.11: Average pVS performance on randomly generated networks  $p = 0.1 * n$ 

Problem ID	$n$	$p$	Objective Gap $ z_{pvs} - z_{vs} $	Average Solution Time		Speedups
				VS	pVS	
2	100	10	0	0.015	0.003	5.0
3	100	10	0	0.017	0.001	17.0
8	200	20	0	0.214	0.010	21.4
13	300	30	0	1.167	0.036	32.4
18	400	40	0	3.637	0.102	35.7
23	500	50	0	8.681	0.223	38.9
28	600	60	0	18.672	0.454	41.1
33	700	70	0	35.017	0.827	42.3
37	800	80	0	62.946	1.426	44.1
40	900	90	0	100.915	2.223	45.4

Tabelle A.12: Average pVS performance on randomly generated networks  $p = 0.2 * n$ 

Problem ID	$n$	$p$	Objective Gap $ z_{pvs} - z_{vs} $	Average Solution Time		Speedups
				VS	pVS	
4	100	20	0	0.042	0.003	14.0
9	200	40	0	0.648	0.027	24.0
14	300	60	0	3.436	0.093	36.9
19	400	80	0	12.163	0.306	39.7
24	500	100	0	27.503	0.649	42.4
29	600	120	0	59.624	1.383	43.1
34	700	140	0	112.597	2.453	45.9

Tabelle A.13: Average pVS performance on randomly generated networks  $p = 0.33 * n$ 

Problem ID	$n$	$p$	Objective Gap $ z_{pvs} - z_{vs} $	Average Solution Time		Speedups
				VS	pVS	
5	100	33	0	0.084	0.004	21.0
10	200	67	0	1.480	0.054	27.4
15	300	100	0	7.577	0.201	37.7
20	400	133	0	24.565	0.607	40.5
25	500	167	0	61.584	1.386	44.4
30	600	200	0	128.341	2.955	43.4

- First GPU-based parallel vertex substitution algorithm for the p-median problem
- Improved the computational complexity of the algorithm
- Gained up to 57 times of speedup compared to the original vertex substitution algorithm

ACCEPTED MANUSCRIPT