INTERNATIONAL JOURNAL OF COMPUTERS
Issue 1, Volume 5, 2011

# Faster Facility Location and Hierarchical Clustering

J. Skála* I. Kolingerová*

Department of Computer Science and Engineering, Faculty of Applied Sciences,
University of West Bohemia, Univerzitní 22, 306 14 Pilsen, Czech Republic

## Abstract

We propose several methods to speed up the facility location, and the single link and the complete link clustering algorithms. The local search algorithm for the facility location is accelerated by introducing several space partitioning methods and a parallelisation on the CPU of a standard desktop computer. The influence of the cluster size on the speedup is documented. The paper further presents the computation of the single link and the complete link clustering on the GPU using the CUDA architecture.

## 1 Introduction

The facility location problem is a clustering task generally formulated as follows. Let $F$ be the set of so called *facilities*, and $C$ be the set of *clients*. Every client should be serviced by (connected to) a facility. The problem is to decide which facilities to *open*, and which clients should they service. The *facility cost* must be paid for opening a facility, and the *service cost* must be paid for connecting clients to facilities (mostly based on the distance).

The problem has a direct real life application. Imagine cities that need to be supplied with electricity, and there are several potential locations where a power plant could be built. Building a power plant everywhere would be too expensive; as well as connecting all the cities to a single one. It is to be decided where to build a power plant, and where to connect particular cities. It is necessary to find such a balance to minimise the overall costs.

Expressing the problem in a mathematical way, the task is to minimise the overall clustering cost $Q$ defined as

$$Q = \sum_{j \in F} fc + \sum_{i \in C} c_{ij} \qquad (1)$$

where $fc$ is the facility cost, and $c_{ij}$ is the distance of the client $i$ to its facility $j$. Distances are considered non-negative, symmetric, and satisfying the triangle inequality. There are generally no restrictions on the set of facilities $F$. It can be independent of $C$, a subset of $C$, or equal to $C$. There are some specialisations of the facility location problem. Facilities may have different facility costs, and may have limited capacities to service just a certain number of clients. These specialisations are not considered in this paper.

To compute an ordinary clustering of a set $P$, simply set $C = P$ and $F = P$. Unlike the $k$-means algorithm, there is no need to specify the number of clusters in advance. It is, however, necessary to choose the facility cost. Basically, it determines the cluster size. A high value will produce a low number of large clusters. Facilities are expensive, so only a few of them will be opened, and a lot of clients connected to each of them. Contrary, a low facility cost will result in many small clusters. Facilities are cheap, so a lot of them will be opened, and clients distributed among them. Recommendations on how to set the facility cost, and an experimental evaluation of the effects of the facility cost, can be found in our paper [1].

One particularly popular approach for the facility location is the local search algorithm described in Section 2.1. However, its straightforward use is rather inefficient. It is therefore necessary to employ acceleration techniques. A space partitioning, such as a quadtree, and a parallelisation are especially suitable for the use with the local search algorithm. The possible speedup achieves 80–95 %.

The facility location algorithm requires the facility cost as an input parameter. Specifying this value might be unnatural in some scenarios. Perhaps the most convenient would be to somehow specify the desired size of the clusters, e.g., their maximal diameter. The single link and especially the complete link algorithms are particularly suitable for that.

With powerful graphics cards becoming a common part of desktop computers, the research focuses on using the GPU (Graphics Processing Unit) to solve computationally intensive tasks. CUDA (Compute Unified Device Architecture) is a framework developed by NVIDIA for parallel computing on the GPU. It allows using the graphics processor as a general purpose computing unit such as for computing the fast Fourier transform (FFT) [2]. The single link and the complete link algorithms can be implemented efficiently using CUDA.

The next section presents existing methods for the facility location, and the single link and the complete link clustering. Section 3 introduces our improvements to speed up

*This work has been supported by the Czech Science Foundation under the research project 201/09/0097.

the facility location computation. Section 4 proposes the complete link computation on the GPU using the CUDA architecture. Section 5 documents the experimental evaluation of the presented algorithms.

# 2 Related work

The following sections describe selected clustering algorithms. Several different approaches to solve the facility location problem are described. A more detailed summary of the methods may be found in [3]. The single link and the complete link algorithms are described.

## 2.1 Local search algorithm

From the general point of view, the local search technique operates on a graph on the space of all feasible solutions. Two solutions are connected by an edge if one solution can be obtained from the other by a particular type of modification. The local search technique then walks in the graph along nodes with decreasing costs, and searches for a local optimum. That is such a node whose cost is not greater than the cost of any of its neighbours. Korupolu et al. [4] analysed clustering techniques based on the local search. One of the first such techniques was proposed by Charikar and Guha [5]. First, a coarse initial solution is generated. It is then iteratively refined by a series of local improvements. A single local search step can be briefly described as follows. A facility is chosen at random, and it is decided whether opening it can improve the solution. If so, nearby clients are reassigned to the new facility. Facilities with a low number of remaining clients are then closed and their clients are reassigned to the new facility too.

Describing the local search algorithm more precisely, a facility $j \in F$ is selected at random (does not matter whether it is opened or closed), and it is decided whether it can improve the current solution: If $j$ is not already opened, the facility cost would have to be paid for opening it. Next, some clients may be closer to $j$ then to their current facility. All such clients can be reassigned to $j$, decreasing the connection cost. After that, some facilities may have just a few clients. If those clients would be reassigned somewhere else, the facilities could be closed and their facility costs spared. To limit computational complexity, reassignments are allowed only to the facility $j$ which is being investigated. The reassignments will indeed increase connection costs, but the savings for closing the facilities (sparing their facility costs) could be larger. The possible improvement of the current solution is computed by the *gain* function. If $gain(j) > 0$, the facility $j$ is opened (if not already opened), and reassignments and closures are performed.

In order to obtain a constant-factor approximation, the described local search technique is repeated $N \log N$ times [5], where $N$ is the number of potential facilities. We believe that the number of iterations could be considerably reduced at the cost of slightly decreased accuracy. Detailed experiments can be found in our paper [1].

An algorithm to create the initial solution is also presented in [5]. It is for the general case when the facility cost can be different for each facility. This text assumes uniform facility costs so a different algorithm proposed by Meyerson [6] will be described. It assumes that all input points are potential facilities, i.e., $C = F$, which is quite common in general clustering problems. Points are taken in random order. A facility is always created at the first one. For every other point $i$, the distance $c_{ij}$ to the closest already open facility $j$ is measured. A new facility is opened at $i$ with probability $c_{ij}/fc$ (or one, if $c_{ij} > fc$). Otherwise, the point $i$ is assigned to the facility $j$.

## 2.2 Other facility location algorithms

Linear programming rounding was introduced by Shmoys et al. [7] based on the work by Lin and Vitter [8]. It was later extended and improved in [9, 10]. The facility location problem is formulated as an integer program. The linear relaxation of the program is solved in polynomial time. The fractional solution is then rounded to the integer solution while increasing the clustering cost by a small constant factor. The proof may be found in [7].

A related approach also based on the linear programming is the primal-dual algorithm introduced by Jain and Vazirani [11] and later addressed by Charikar and Guha [5] and Mahdian et al. [12]. The method again starts with an integer program and its linear relaxation. A dual linear program is constructed whose solution gives the solution to the original problem.

A different approach is to use genetic algorithms. Choi et al. [13] presented a heuristic using a genetic algorithm for plant-warehouse location problem. Lee at al. [14] propose an immunity based genetic algorithm to solve the quadratic assignment problem which is closely related to the facility location.

## 2.3 Single link and complete link

Single link [15] and complete link [16] are hierarchical clustering algorithms, i.e., they start with each element in a single cluster and proceed by merging similar clusters together. The algorithms differ in the way they measure the similarity between clusters. The single link algorithm defines the distance between two clusters as the *minimum* pairwise distance between the elements of the two clusters, i.e., the distance of the *most similar* elements from the two clusters. By contrast, the complete link algorithm uses the *maximum* pairwise distance, i.e., the distance of the *most dissimilar* elements, which is practically the diameter of the two cluster union. Special distance measures can be used, e.g., for clustering documents [17].

The single link algorithm is more versatile but tends to produce straggly or elongated clusters. This could be unpleasant but in some scenarios it is very useful to detect non-spherical clusters. The complete link algorithm produces compact clusters.

Let us review some terms of graph theory for the following paragraph. A *connected graph* is a graph where there is a path connecting each pair of points. A *connected component* of a graph is a maximal set of connected points such that there is a path connecting each pair. A *clique* in a graph is a set of points that are completely linked together.

Both the single link and the complete link clustering can be constructed by similar algorithms. The complete link algorithm can be summarised as follows:

1. Start with each element in a distinct cluster
2. Compute distances between all pairs of elements
3. Take the distances in an ascending order. For each such distance $d$, create a graph where all pairs of elements closer than $d$ are connected by an edge. When all the elements form a single clique, stop.
4. The result is a hierarchy of graphs where an arbitrary similarity level can be selected. The clusters are determined by the *maximal cliques* of the appropriate graph.

The single link algorithm basically works the same way. The difference is that the second phase is terminated when all the elements form a connected graph. When a graph is selected from the hierarchy, the clusters are determined by the *connected components* of the graph.

The algorithm above is presented in the terms of graph theory. A more practical notation can be found in [18] where it was used to implement the single link and the complete link clustering on a concurrent supercomputer.

1. Start with each vertex in a distinct cluster
2. Compute the mutual distances between the clusters
3. Find the closest clusters and merge them
4. Update the distances of all other clusters to the newly merged cluster
5. If not all clusters have been merged, go to 3

# 3 Speeding up the facility location

The local search algorithm used directly as it is theoretically presented is too slow. This section presents two acceleration techniques to make the algorithm more efficient – a space partitioning and a parallelisation which are especially suitable for the local search.

The local search algorithm for facility location basically consists of three operations. First the generation of the initial solution, then repeatedly computing the gain, and eventually performing reassignments.

Generating the initial solution is done only once at the beginning, and takes only about 2 % of the computing time. The iterative local search algorithm remains, consisting of repetitive gain computation and eventual reassignments. Most of the time, about 96 %, is spent evaluating the gain function; see Section 5.1. The reassignments take significantly less time and the nature of the operation (reassigning client indices from one array to another) does not give much space for improvements. The effort to speed up the computation was therefore focused on the gain computation.

## 3.1 Space partitioning

To compute the gain of a potential new facility $f$, the clients must be inspected to decide whether $f$ would be closer to them than their current facilities. However, not all the clients need to be inspected. Many of them are just too far from $f$, so that it is sure their current facility is closer. Eliminating such inperspective clients would be a great benefit. The core idea is therefore to inspect only those clients that can actually contribute to the gain.

We introduce the term of *the longest connection*. Let $C'$ be a subset of clients connected to some facilities. The longest connection $c_{max}$ is the maximal distance of any client $i \in C'$ to its facility $j$.

$$c_{max} = \max_{i \in C'} c_{ij} \tag{2}$$

The longest connection is the upper bound on the distance where any client from $C'$ can be reassigned without increasing the connection cost.

To derive an upper bound for the whole set $C'$, consider the worst case – a client on the $C'$ boundary. The client can be reassigned at most $c_{max}$ far away from the $C'$ boundary[1]. Therefore, if the distance of a facility candidate $f$ from the boundary $C'$ is greater than $c_{max}$, then no client from $C'$ can be reassigned to $f$ without increasing the connection cost.

### 3.1.1 Quadtree, octree, kD-tree

The idea is straightforward – partition the clients using a tree and then inspect only those tree nodes that contain perspective clients. A fundamental space partitioning is a simple quadtree (for 2D) or an octree (for 3D). A kD-tree is perhaps a bit more difficult to build, but it can well adapt to non-uniform distribution of input data. The kD-tree showed particularly good performance on a data in the form of a narrow rectangle.

[1]The form of the boundary depends on the implementation. It can be, e.g., a bounding box, a bounding sphere, or a convex hull.

Either of the trees is built once at the beginning of the clustering. Each tree node stores its bounding box, and the longest connection $c_{max}$ of the clients belonging to the node. Leaf nodes contain in addition a list of clients belonging to them. Figure 1 shows an example of a quadtree with the longest connections highlighted in red.

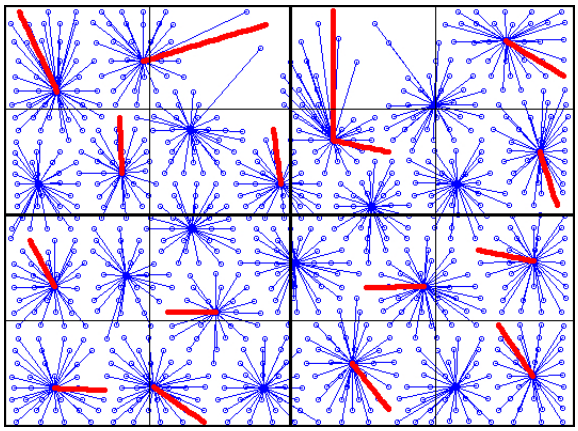

Figure 1: Example of a quadtree with the longest connections highlighted.

The gain of a facility candidate $f$ is computed by traversing the tree. The distance of $f$ to the bounding box of each node is computed. If the distance is smaller than the longest connection in the node, then the node is traversed. When the traversal gets to a leaf node, all its clients are inspected.

After considering the relevant clients for reassignment, it is necessary to consider facilities for closure. This cannot be done using the tree because closing a facility creates a *spare*, which can pay for reassigning the clients farther. Closing a facility creates a *spare* equal to its facility cost. The clients of the facility are considered one by one. Let $c$ be the distance of a client to its current facility, and let $c_f$ be the distance to the facility candidate. The distance extension $c_f - c$ is subtracted from the *spare*. As soon as the spare reaches zero, the facility is not worth closing. This is usually decided after testing just several clients.

If the gain of the facility candidate comes out positive, the reassignments and closures are actually performed. This may change the longest connection $c_{max}$ of some tree nodes. If a client with the longest connection is reassigned to a closer facility ($c_{max}$ will decrease), or if any client is reassigned to a farther facility ($c_{max}$ may increase), then the tree leaf is marked that it needs to update $c_{max}$. Once the reassignments are done, the marked tree leaves are updated. The updates propagate up to their parents.

The tree space partitioning works better for a low facility cost which yields small clusters. The longest connections are short, so only the tree nodes very close to the facility candidate are traversed. Experiments can be found in Section 5.3.

### 3.1.2 Partitioning by facilities

Why to create an artificial space partitioning when there is one already constructed? The clustering itself – although not finished yet – is a partitioning, and it perfectly corresponds to the task being solved.

Each facility (cluster centre) keeps the list of its clients, and the longest connection $c_{max}$. The cluster boundary is a sphere centred at the facility with the radius $c_{max}$. To compute the gain of a facility candidate $f$, clusters are considered one by one. It is to be decided whether $f$ is at most $c_{max}$ from the cluster boundary, that is at most $2c_{max}$ from the facility. See Figure 2 for an illustration. If $f$ lies close enough, all the clients of the facility are inspected. Otherwise, the facility is only considered for closure. The algorithm is the same as described in Section 3.1.1.

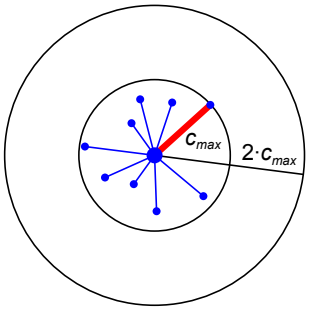


Figure 2: Facility with its boundary and the longest connection.

If the gain of the facility candidate comes out positive, the reassignments and closures are performed. As in the case of trees in Section 3.1.1, if a reassignment could change the longest connection of a facility, the facility is marked. Once the reassignments are done, the marked facilities are updated.

The cluster size also matters for the partitioning by facilities. A great facility cost yields large clusters. The longest connections are long, so even the clusters far away from the facility candidate must be inspected. A low facility cost and small clusters is also not good. There is a lot of clusters, and, although most of them are too far from the candidate, the overhead grows. Experiments can be found in Section 5.3.

## 3.2 Parallelisation

Modern computers commonly have four or more CPU cores. This gives the possibility to employ parallelism to further accelerate the computation. This section describes two possible approaches to the parallelisation. Both of them suppose the use of the space partitioning by facilities (see Section 3.1.2) because the parallelisation is straightforward.

#### 3.2.1 Single gain computed in parallel

Thanks to the space partitioning, the gain is computed facility by facility. The computation for each facility is independent. Therefore, the most straightforward parallelisation is to divide the set of facilities among several threads and let every thread deal with a subset of facilities. The computation can proceed independently, so no synchronisation is necessary. When all the threads finish, their particular results are easily merged – the gain values are summed up, the lists of clients to reassign are concatenated, and the lists of facilities to close are concatenated.

#### 3.2.2 Parallel computations of several gains

The gain often gives a positive result in the early iterations of the algorithm. With increasing number of iterations performed, positive results become less frequent. In later stages of the algorithm, gain results come out mostly negative. Again, the performance is affected by the cluster size. A detailed evaluation can be found in Section 5.3.

This gives another possibility of parallelisation – to compute the gain for several different facility candidates simultaneously. Candidates with a negative gain are useless. No reassignments are done, so the clustering is not modified. Therefore, any number of negative results can be accepted as valid iterations of the local search algorithm at the same time (yet unsuccessful to improve the solution). This means that actually several iterations of the clustering algorithm are executed in parallel. The computation pauses only when a positive result appears, and the clustering is modified.

If the gain of any of the candidates is positive, the appropriate reassignments are done. Other candidates with a negative gain can be accepted as well. But if more candidates happen to have a positive gain, the one with the greatest gain is used. The remaining positive gain results must be discarded because the clustering changes after the reassignments to the first candidate, so the other positive results are not valid anymore.

# 4 Single link and complete link on the GPU

This section presents the computation of the single link and the complete link clustering, described in Section 2.3, on a GPU using the CUDA framework.

## 4.1 The algorithm for the GPU

The complete link algorithm starts with computing the distance matrix. Each element is computed by one GPU thread. Only half of the distances are actually computed, due to the symmetry of the distance matrix. Section 4.2 describes this in detail.

Each vertex is initialised as an individual cluster, and maintains a list of assigned vertices, which at the beginning contains only the vertex itself. All the rows of the distance matrix are *active*, meaning that the cluster corresponding to the row has not been merged into some other cluster. The algorithm then proceeds as described in Section 2.3.

Sequential algorithms mostly maintain a sorted list of the closest pairs of clusters to quickly find the closest ones to merge. Our GPU implementation pre-computes the mutual distances, but does not do the sorting. It relies on the massively parallel computing power and finds the closest clusters (the minimum in a distance matrix) by brute force. Maintaining the sorted list is inherently a sequential operation, and it would not fit the parallel computation.

To find the closest pair of clusters, each active row of the distance matrix is scanned by a GPU thread to find the minimum. Again, only half of the elements are scanned, due to the symmetry. The partial minima from the rows are gathered by the CPU to find the global minimum which identifies the closest clusters $A$ and $B$.

The clusters are merged, specifically, $B$ is merged into $A$. The list of vertices already assigned to $B$ is copied to $A$. The row corresponding to cluster $B$ in the distance matrix is *deactivated*.

Now the distances to new cluster must be updated. For each active row of the distance matrix, except $A$ (and the deactivated $B$), a GPU thread compares the distance to $A$ with the distance to $B$. The greater value is kept as the distance to the merged cluster. The matrix symmetry ensures that the distances in the row corresponding to the merged cluster $A$ will be updated as well.

## 4.2 Maintaining the distance matrix

The distance matrix used in the computation is indeed symmetric. The elements on the diagonal are of no use, so only the elements above the diagonal are really needed. This is, however, not well suitable for distributing the work among the parallel threads. If the matrix size is $n$, the first row contains $n-1$ elements that need to be processed, while the before-last row contains just a single element to be processed.

A better load balancing can be elegantly achieved by a smart distribution of the matrix elements. We simply store $\left\lfloor \frac{n}{2} \right\rfloor$ elements in every row, starting from the first element to the right of the diagonal. If we get to the last column, we continue with the first column (count columns modulo $n$). The stored elements form a band above the diagonal and a triangle in the lower left corner. The idea is illustrated in Figure 3. The elements that are actually stored are marked grey. If $n$ is even, $n/2$ elements will be stored twice (the dark grey elements in the figure), but this is no problem.

The work for the parallel threads is then easily distributed as if the matrix elements would be stored in a rectangular matrix $n \times \left\lfloor \frac{n}{2} \right\rfloor$. Each thread is assigned a sin-

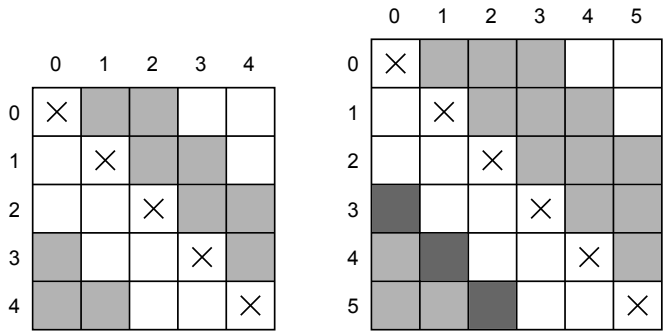

Figure 3: Scheme of the distance matrix storage for $n$ odd and for $n$ even, respectively.

gle row $i'$ and processes the elements $j' \in \{0, 1, \ldots, \lfloor \frac{n}{2} \rfloor\}$. Indices to the original $n \times n$ matrix are computed by the following indexing:

$$\begin{aligned} i &= i' \\ j &= (j' + i' + 1) \bmod n \end{aligned} \tag{3}$$

Converting the index back is more complicated:

$$\begin{aligned} &\text{if} \quad \left(0 < i - j < \frac{n+2}{2} \quad \text{OR} \quad j - i > \frac{n+2}{2}\right) \\ &\quad \text{swap}(i, j) \\ i' &= i \\ j' &= (j - i + n) \bmod n \end{aligned} \tag{4}$$

Fortunately, the backwards conversion it never needed in the algorithm.

# 5 Experiments

This section summarises the experiments done to identify the bottlenecks, find the possibilities for improvement, and to document the achieved speedup.

The program is written in C# under the .NET Framework 2.0. Experiments were performed on a PC with Core 2 Quad 2.4 GHz CPU, 4 GB RAM, running Windows XP. The parallel computations were executed in four threads.

The presented experiments were done on the following data sets: Bunny and Armadillo[2] (laser scanned 3D models), the famous Utah teapot[3] (vertices sampled from the surface), and the Crater Lake[4] (nearly 2D terrain model).

## 5.1 Facility location running time

Table 1 shows the running time of the original algorithm, and partial times for the relevant parts of the computation.

[2]Stanford 3D Scanning Repository, `http://graphics.stanford.edu/data/`

[3]Original 3D model by M. Newell, 1975.

[4]M. Garland, `http://mgarland.org/dist/models/`

Evaluating the gain functions takes approximately 96 % of time. Generating the initial solution and performing the reassignments are negligible.

## 5.2 Facility location speedup

This section documents the speedup achieved by implementing the proposed improvements. Table 2 shows the speedup by space partitioning. The kD-tree is the best because it splits the data adaptively. The space partitioning by facilities is a small bit slower, but it is perhaps easier to implement, and the parallelisation is straightforward. The octree performs slightly worse on the Crater Lake because the data are practically flat.

Table 3 shows the speedup achieved by the parallelisation. The columns *ratio 1* show the speedup compared to the original algorithm, the columns *ratio 2* show the speedup compared to the (non-parallelised) partitioning by facilities. Although ran on a 4 core CPU, the program is not 4 times faster because only the gain evaluation runs in parallel. Eventual reassignments remain sequential. The CPU utilisation oscillated between 25 and 95 %.

## 5.3 The influence of cluster size

This section shows how the cluster size affects the efficiency of the proposed algorithms. It is to be noted that the cluster size (the facility cost) is a user specified parameter, and it is therefore unreasonable to search for an optimum.

The graph in Figure 4 shows how often the gain comes out positive for various facility cost values. This strongly influences the efficiency of the parallel computation of several gains. At the beginning, gains are all positive because the initial clustering is very coarse, and almost any facility candidate can improve it. Later, the ratio of positive gains drops, especially for great facility costs (large clusters).

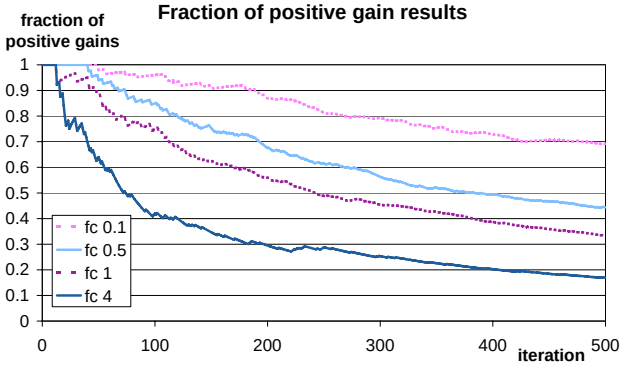


Figure 4: Fraction of positive gain results.

The graph in Figure 5 shows the fraction of *vertices* inspected for various facility cost values. A *vertex* here means either a facility or a client. In both cases, inspecting

Table 1: Total running time of the original algorithm, and partial times for the important parts.

| Data set | Number of vertices | Total time [s] | Initial solution [s] | Gains [s] | Reassignments [s] |
|---|---|---|---|---|---|
| Bunny | 35947 | 23.1 | 0.7 | 21.5 | 0.1 |
| Teapot | 80203 | 113.5 | 2.3 | 107.7 | 0.5 |
| Crater | 100001 | 175.1 | 2.8 | 167.9 | 0.5 |
| Armadillo | 172974 | 524.3 | 9.0 | 515.1 | 1.4 |

Table 2: Speedup achieved by space partitioning.

| Data set | Vertices | Original time [s] | Quadtree t. [s] | Quadtree ratio | Octree t. [s] | Octree ratio | kD-tree t. [s] | kD-tree ratio | Facility t. [s] | Facility ratio |
|---|---|---|---|---|---|---|---|---|---|---|
| Bunny | 35947 | 23.1 | 4.7 | 80 % | 4.6 | 80 % | 4.1 | 82 % | 4.3 | 81 % |
| Teapot | 80203 | 113.5 | 16.2 | 86 % | 15.1 | 87 % | 13.4 | 88 % | 16.1 | 86 % |
| Crater | 100001 | 175.1 | 18.0 | 90 % | 21.7 | 88 % | 17.3 | 90 % | 19.7 | 89 % |
| Armadillo | 172974 | 524.3 | 65.7 | 87 % | 62.1 | 88 % | 54.9 | 90 % | 60.2 | 89 % |

means computing the distance to the facility candidate. If a facility is close enough, all its clients are inspected. The notion of *vertices* was introduced to overcome the issue described at the end of Section 3.1.2. The *vertices* represent all the facilities and clients we effectively have to deal with. The ratio of inspected vertices is greater at the beginning because of the coarse initial clustering. The ratio is lower for a small facility cost (small clusters).

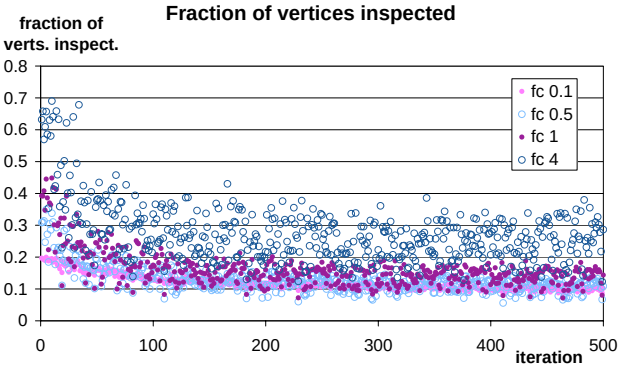


Figure 5: Fraction of vertices inspected.

### 5.4 Complete link on the GPU

This section documents the experiments with the complete link clustering on the GPU using CUDA. The measurements were performed on a PC with Pentium 4 3.6 GHz CPU and an NVIDIA GeForce 8800 GTX graphics card. The GPU algorithm was compared with a similar implementation written in C running on the CPU. Several 3D surface models were used as test data.

The results are summarised in Table 4. The overall speedup ranges from 15 to 40 % which is not as great as expected. It is probably caused by fragments of work still being done on the CPU. Transferring the data between the main memory and the graphics card memory causes delays. The computation of the distance matrix is very fast already. The following steps of the complete link algorithm can be further optimised.

## 6 Conclusion

The paper presented various techniques to accelerate the facility location, and the single link and the complete link clustering algorithms. Several space partitioning methods were introduced for the facility location. Further speedup was achieved by proposing two parallelisation schemes. The performance of the suggested methods was evaluated. The best one brings a speedup of up to 95%. The influence of the cluster size on the efficiency of the proposed methods was documented.

The single link and the complete link clustering algorithms were implemented on the GPU using the CUDA architecture. The speedup for the complete link algorithm ranges from 15 to 40 %. In the future work, we would like to further improve the proposed methods, and especially optimise the complete link clustering on the GPU to achieve a better performance.

## References

[1] J. Skála and I. Kolingerová, "Clustering geometric data streams," in *SIGRAD 2007*, pp. 17–23, 2007.

[2] S. Romero, M. A. Trenas, E. Gutierrez, and E. L. Zapata, "Locality-improved FFT implementation on a graphics processor," in *Proceedings of the 7th WSEAS International Conference on Signal Processing, Computational Geometry & Artificial Vision*, (Stevens Point, Wisconsin, USA), pp. 58–63, World Scientific and Engineering Academy and Society (WSEAS), 2007.

Table 3: Speedup achieved by parallelising the gain computation by facilities.

| | | Facility | Parallel single gain | | | Parallel multiple gains | | |
|---|---|---|---|---|---|---|---|---|
| Data set | Vertices | time [s] | t. [s] | ratio 1 | ratio 2 | t. [s] | ratio 1 | ratio 2 |
| Bunny | 35947 | 4.3 | 2.5 | 89 % | 42 % | 2.2 | 90 % | 48 % |
| Teapot | 80203 | 16.1 | 8.5 | 92 % | 47 % | 8.2 | 93 % | 49 % |
| Crater | 100001 | 19.7 | 10.3 | 94 % | 48 % | 9.4 | 95 % | 52 % |
| Armadillo | 172974 | 60.2 | 29.2 | 94 % | 52 % | 27.0 | 95 % | 55 % |

Table 4: Speedup of the CUDA implementation.

| Data set | Number of vertices | CPU time [s] | GPU time [s] | Speedup |
|---|---|---|---|---|
| Objects | 1420 | 2.855 | 2.434 | 15 % |
| Ellipsoid | 2452 | 11.985 | 7.193 | 40 % |
| Cow | 2905 | 25.578 | 19.420 | 24 % |
| Head | 4098 | 70.369 | 54.951 | 22 % |

[3] D. B. Shmoys, "Approximation algorithms for facility location problems," in *APPROX '00: Approximation Algorithms for Combinatorial Optimization*, vol. 1913 of *Lecture Notes in Computer Science*, (London, UK), pp. 27–33, Springer-Verlag, 2000.

[4] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman, "Analysis of a local search heuristic for facility location problems," in *SODA: ACM-SIAM Symposium on Discrete algorithms*, (Philadelphia, PA, USA), pp. 1–10, Society for Industrial and Applied Mathematics, 1998.

[5] M. Charikar and S. Guha, "Improved combinatorial algorithms for the facility location and k-median problems," in *IEEE Symposium on Foundations of Computer Science*, pp. 378–388, 1999.

[6] A. Meyerson, "Online facility location," in *FOCS '01: IEEE Symposium on Foundations of Computer Science*, (Washington, DC, USA), pp. 426–431, IEEE Computer Society, 2001.

[7] D. B. Shmoys, É. Tardos, and K. Aardal, "Approximation algorithms for facility location problems (extended abstract)," in *ACM Symposium on Theory of Computing*, pp. 265–274, 1997.

[8] J.-H. Lin and J. S. Vitter, "Approximation algorithms for geometric median problems," *Information Processing Letters*, vol. 44, pp. 245–249, 1992.

[9] S. Guha and S. Khuller, "Greedy strikes back: Improved facility location algorithms," in *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pp. 649–657, 1998.

[10] F. A. Chudak, "Improved approximation algorithms for uncapacitated facility location," *Lecture Notes in Computer Science*, vol. 1412, pp. 180–194, 1998.

[11] K. Jain and V. V. Vazirani, "Primal-dual approximation algorithms for metric facility location and k-median problems," in *IEEE Symposium on Foundations of Computer Science*, pp. 2–13, 1999.

[12] M. Mahdian, E. Markakis, A. Saberi, and V. Vazirani, "A greedy facility location algorithm analyzed using dual fitting," *Lecture Notes in Computer Science*, vol. 2129, pp. 127–137, 2001.

[13] S.-K. Choi, T. Lee, and J. Kim, "The genetic heuristics for the plant and warehouse location problem," *WSEAS Transactions on Circuits and Systems*, vol. 2, no. 4, pp. 704–709, 2003.

[14] C.-Y. Lee, Z.-J. Lee, and S.-F. Su, "Immunity based genetic algorithm for solving quadratic assignment problem (qap)," in *Proceedings of the 2nd WSEAS International Conference on Electronics, Control and Signal Processing*, pp. 1–9, 2003.

[15] P. H. A. Sneath and R. R. Sokal, *Numerical taxonomy: The principles and practice of numerical classification.* San Francisco: W.H. Freeman, 1973.

[16] B. King, "Step-wise clustering procedures," *Journal of the American Statistical Association*, vol. 62, no. 317, pp. 86–101, 1967.

[17] A. Jalali, F. Oroumchian, and M. R. Hejazi, "Comparison of different distance measures on hierarchical document clustering in 2–pass retrieval," *WSEAS Transactions on Computers*, vol. 3, no. 3, pp. 725–731, 2004.

[18] S. Arumugavelu and N. Ranganathan, "Simd algorithms for single link and complete link pattern clustering," in *Proceedings of the International Conference on Pattern Recognition*, (Washington, DC, USA), pp. 625–629, IEEE Computer Society, 1996.