

PARALLELIZATION STUDY ON THE CLUSTERING
TECHNIQUE TO MINE LARGE DATASETS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ÇANKAYA UNIVERSITY

BY

AHMET ARTU YILDIRIM

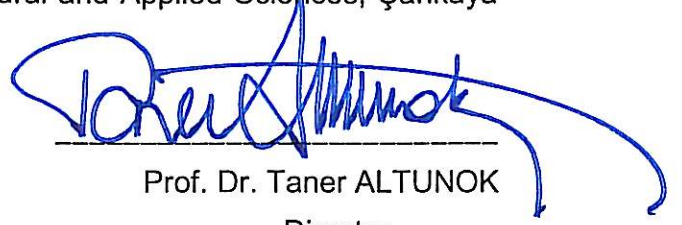
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2011

Title of the Thesis : **Parallelization Study on the Clustering Technique to Mine Large Datasets**

Submitted by **Ahmet Artu YILDIRIM**

Approval of the Graduate School of Natural and Applied Sciences, Çankaya University.



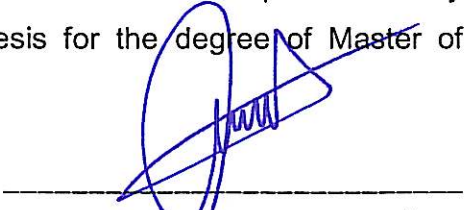
Prof. Dr. Taner ALTUNOK
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.



Prof. Dr. Mehmet Reşit TOLUN
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.



Assoc. Prof. Dr. Cem ÖZDOĞAN
Supervisor

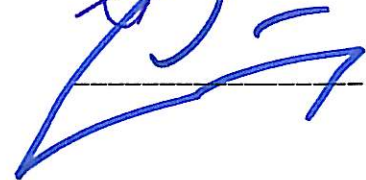
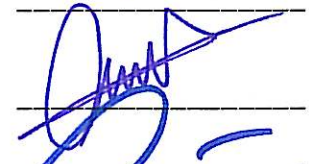
Examination Date : 27.01.2011

Examining Committee Members :

Prof. Dr. Mehmet Reşit TOLUN (Çankaya Univ.)

Assoc. Prof. Dr. Cem ÖZDOĞAN (Çankaya Univ.)


Dr. Ersin ELBAŞI (TUBITAK)



STATEMENT OF NON-PLAGIARISM

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name : Ahmet Artu YILDIRIM

Signature : 

Date : 06.06.2011

ABSTRACT

PARALLELIZATION STUDY ON THE CLUSTERING TECHNIQUE TO MINE LARGE DATASETS

YILDIRIM, Ahmet Artu

M.Sc., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Cem ÖZDOĞAN

January 2011, 55 pages

Parallel clustering algorithm implementations concerning message passing interface (MPI) and compute unified device architecture (CUDA) model with their applications to very large datasets have been presented in the thesis. WaveCluster is a novel clustering approach based on wavelet transforms. Despite it's novelty, it requires considerable amount of time to collect results for large sizes of multidimensional datasets. In the MPI algorithm; divide and conquer approach has been followed and communication among processors are kept at minimum to achieve high efficiency. Developed parallel WaveCluster algorithm exposes high speedup and scales linearly with the increasing number of processors. Parallel behavior of WaveCluster approach has been also investigated by executing the algorithm on graphical processing unit (GPU). High speedup values have been obtained in the computation of wavelet transform and connected component labeling algorithms in the GPUs with respect to the sequential algorithms running on the CPU.

Keywords : Cluster Analysis, WaveCluster Approach, Parallel WaveCluster

ÖZ

GENİŞ VERİ KÜMELERİNİ İŞLEME AMACIYLA ÖBEKLEME TEKNİĞİ ÜZERİNE PARALELLEŞTİRME ÇALIŞMASI

YILDIRIM, Ahmet Artu

Yüksek Lisans, Bilgisayar Mühendisliği Anabilim Dalı

Tez Yöneticisi : Doçent Dr. Cem ÖZDOĞAN

Ocak 2011, 55 sayfa

Bu tezde, mesaj geçirme arayüzü (MPI) ve birleşik aygıt mimarisi hesaplaması (CUDA) modelini uygulayarak geliştirilen paralel öbekleme algoritmaları, çok geniş veri kümeleri üzerindeki uygulamaları ile birlikte tanıtılmıştır. WaveCluster, wavelet dönüşümü tabanlı yenilikçi bir öbekleme analizi yaklaşımıdır. Bu yaklaşımın etkinliğine rağmen, çok boyutlu geniş veri kümeleri üzerinde çalıştırıldığında çalışma zamanı fazla olmaktadır. Geliştirilen MPI algoritmasında; yüksek verimlilik değerlerini elde etmek için işlemciler arasındaki haberleşme en az seviyede tutulmuştur. Yapılan deneysel çalışmalarda, MPI algoritması yüksek hızlanma değerleri vermiştir ve ayrıca artan işlemci sayısı ile birlikte doğrusal bir çalışma karakteristiği göstermiştir. WaveCluster yaklaşımı ayrıca grafik işlemci ünitesi (GPU) üzerinde CUDA modeli uygulanarak paralelleştirilmiştir. Geliştirilen CUDA algoritmasında, wavelet dönüşümü ve bağlı parçaları işaretleme algoritmaları geliştirilmiştir. CPU üzerinde sıralı çalışan WaveCluster yaklaşımına kıyasla CUDA algoritmalarında yüksek hızlanma değerleri elde edilmiştir.

Anahtar Kelimeler : Öbekleme Analizi, WaveCluster Yaklaşımı, Paralel WaveCluster

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisor Assoc. Prof. Dr. Cem ÖZDOĞAN for his professional guidance and continuous encouragement throughout my thesis study. His dedication and enthusiasm for scientific research is unsurpassed.

I also like to thank my wife Şule YILDIRIM for her support during the thesis study.

Finally, I have great pleasure to express my gratitude to Çankaya University in which the experiments were carried out on the computer cluster system of Boron.

TABLE OF CONTENTS

STATEMENT OF NON-PLAGIARISM	III
ABSTRACT.....	IV
ÖZ.....	V
ACKNOWLEDGMENTS	VI
TABLE OF CONTENTS.....	VII
LIST OF TABLES.....	IX
LIST OF FIGURES	X
CHAPTERS:	
1. INTRODUCTION	1
2. DATA MINING	5
2.1. Data Mining Methods	7
3. PARALLEL COMPUTING	9
3.1. Message Passing Interface.....	12
3.2. CUDA Programming Model.....	13
3.3. Parallel Data Mining Approaches.....	16
4. CLUSTER ANALYSIS.....	19
4.1. Sequential Cluster Analysis Algorithms.....	19

4.2. Parallel Cluster Analysis Algorithms.....	21
5. WAVECLUSTER APPROACH.....	24
6. PARALLEL WAVECLUSTER ALGORITHMS	28
6.1. Parallel WaveCluster Algorithm on Shared Memory Architecture	
Using CUDA.....	28
6.1.1. Implementation of low-frequency component extraction of	
the signal.....	28
6.1.2. Implementation of connected component labeling	30
6.2 Parallel WaveCluster Algorithm on Distributed Memory	
Architecture Using MPI	32
7. EXPERIMENTAL RESULTS OF THE CUDA ALGORITHMS	38
8. EXPERIMENTAL RESULTS OF THE MPI ALGORITHM	41
9. CONCLUSION	53
REFERENCES	R1
APPENDICES:	
A. MPI CODE OF PARALLEL WAVECLUSTER ALGORITHMS	A1
B. CURRICULUM VITAE	A32

LIST OF TABLES

Table 1	Performance Results of CUDA and CPU Versions of Low-Frequency Component Extraction for $\rho = 1$ (Times in Microseconds).....	39
Table 2	Performance Results of CUDA and CPU Versions of Connected Component Labeling (CCL) for $\rho = 1$ (Times in Microseconds) ...	39
Table 3	Aggregate Speedup Results of CUDA Algorithms for $\rho = 1$ (Times in Microseconds)	40
Table 4	Execution Times (in Seconds) with and without Consideration of I/O Times for SD1 with Varying Dataset Sizes and Number of Processors (np) for $\rho = 3$ Using 1 Gbit and 100 Mbit Ethernet Controller Cards.....	43
Table 5	Efficiencies for SD1 (Upper) and SD2 (Lower) with Varying Dataset Sizes (DS8, DS16, DS32 and DS65) and Number of Processors (1, 2, 4, 8, 16, 32; 1 Gbit Ethernet) at Scale Level, $\rho = 3$	49

LIST OF FIGURES

Figure 1	Knowledge Discovery Steps	6
Figure 2	Shared Memory Model for m Number of Memory (M) and n Number of Procesor (P) and Cache (C)	10
Figure 3	Distributed Memory Model for n Number of Memory (M), Procesor (P) and Cache (C).....	11
Figure 4	Automatic Scalability of CUDA Program	14
Figure 5	Two-scale Discrete Wavelet Transform Filter Bank	25
Figure 6	WaveCluster Algorithm Multi-Resolution Property. (a) Original Source Dataset. (b) $\rho = 2$ and 6 Clusters are Detected. (c) $\rho = 3$ and 3 clusters are Detected. (Where ρ is the Scale Level)	26
Figure 7	Iteration Demonstration of Connected Component Labeling CUDA Algorithm.....	31
Figure 8	Decomposition of Large Dataset with 2 Dimensions for Varying Number of Processors (np). (a) $np = 16$ (b) $np = 8$	33
Figure 9	Demonstration of Parallel Wavecluster Algorithm Based for Distributed Memory Architecture	37
Figure 10	Sample Datasets Used in Experiments (a) SourceDataset1 (SD1)	

	(b) SourceDataset2 (SD2).....	42
Figure 11	Execution Times with I/O per Object per Processors for SD1 with Respect to Dataset Sizes for 100 Mb (Upper Data Line) and 1Gb (Lower Data Line) Ethernet at Scale Level, $\rho = 3$ vs Number of Processors (np). Inset: That of np=8, 16, 32	45
Figure 12	Execution Times for SD1 with Varying Dataset Sizes (DS8, DS16, DS32 and DS65) and Number of Processors (1, 2, 4, 8, 16, 32; 1 Gbit Ethernet) at Different Scale Levels (ρ); (a) $\rho = 1$, (b) $\rho = 3$ and (c) $\rho = 5$	46
Figure 13	Speedup Plots for SD1 (Upper Plots) and SD2 (Lower Plots) with Varying Dataset Sizes (DS8, DS16, DS32 and DS65) and Number of Processors (1, 2, 4, 8, 16, 32; 1 Gbit Ethernet) at Different Scale Levels; (a, d) $\rho = 1$, (b, e) $\rho = 3$ and (c, f) $\rho = 5$..	48
Figure 14	Speedups with Varying Number of Objects in Dataset and Number of Processors (1, 2, 4, 8, 16, 32; 1 Gbit Ethernet) at Scale Level $\rho = 3$. (a) SD1 (b) SD2.....	51

CHAPTER 1

INTRODUCTION

There has been an explosive growth of very large datasets or databases in scientific and commercial domains with the recent progress in data storage technology. On the other hand, decision makers have rather scarce time to make up the right strategy which have critical importance to survive in the competitive market. This phenomenon is also applicable to researchers who need to make efficient analysis of current and past datasets in order to develop novel ideas. To cope with this problem, data mining techniques have been highly utilized in numerous fields in order to be able to extract useful information as finding patterns, trends, rules, etc.

Clustering is a common data mining technique used for information retrieval by grouping similar objects located in the dataset into disjoint classes or clusters [12]. Since cluster analysis has become crucial task for mining of the data, considerable amount of researches are carried out in developing sequential clustering analysis methods such as K-means [24], BIRCH [43], DBSCAN [11], STING [39] and WaveCluster [34]. Clustering algorithms have been used in various fields including satellite image segmentation [26], unsupervised document clustering [37] and clustering of bioinformatics data [25].

Despite to substantial improvements on processor technology in terms of speed, sequential clustering algorithms may still not complete the required task in a reasonable amount of time at very large datasets. Besides, there may not have enough amounts of available main memory resources to hold all the data on a single computer. One of the possible solutions to overcome those issues

and efficiently mine huge datasets is to make utilization of parallel algorithms [20]. Parallel processing approach is highly used as dividing the task into smaller subtasks and executing them simultaneously to cope with memory limits and to decrease the execution time of the sequential task.

Parallel computer architecture is classified according to the memory sharing model. One of the parallel architectures is distributed memory architecture which consists of processors or nodes connected with each other by means of network infrastructure. In this model, each node has its own memory. As a traditional way, message passing interface (MPI) [18] is extensively used to achieve the communication among the processors by simply sending and receiving messages. The other parallel architecture is shared memory architecture. As its name implies, each node has shared memory dedicated to the computer. Processors access the memory simultaneously by means of data bus. In shared memory systems, OpenMP [8], as an application programming interface (API), provides the programmer flexibility, ease of use and transparency in the creation of the threads running on the central processing unit (CPU). Furthermore, with the advent of many-core graphical processing units (GPUs), GPUs could be used for general purpose computation as a coprocessor of the CPU in addition to its traditional usage of accelerating the graphic rendering. There has been a substantial interest to speed up the CPU-intensive tasks in a parallel manner on GPUs to utilize its enormous computational performance for shared memory system. As an API to do GPU computation, NVIDIA introduced CUDA (Compute Unified Device Architecture) in November 2006 [29] to enable data-parallel general purpose computations on NVIDIA GPUs in an efficient way. Last but not least, hybrid model is a parallel computer architecture with the combination of the shared memory model and distributed memory model.

WaveCluster approach is a novel clustering approach designed for large

spatial datasets. The algorithm has the ability of detecting arbitrary shape clusters at different scales by taking advantage of wavelet transform and can handle noises in an appropriate way. Wavelet transform is a mathematical tool to transform the signal into low and high frequency components (transformed feature space). WaveCluster algorithm detects the connected components in the transformed feature space by applying connected component labeling algorithm and then maps the objects in the transformed feature space to the objects in the original feature space. Wavelet transform could be applied to the feature space multiple times or recursively. Each transform yields coarser representation of the signal. So, the algorithm detects the clusters in different accuracy levels from fine to coarse. By this way, WaveCluster approach gains multi-resolution feature with wavelet transform.

WaveCluster algorithm is an effective algorithm. On the other hand, execution time of the algorithm has become a serious concern when dataset size is very large or huge. In this thesis, parallel WaveCluster algorithms have been presented with respect to message passing model and CUDA model. In the MPI algorithm, master-slave and divide and conquer approach have been followed by distributing the sub datasets to processors and employing the processors to apply WaveCluster approach on its sub dataset. In the final phase, merging operation is required to make the detected clusters globally valid. The algorithm has a time complexity of $O(N)$. This means that the algorithm scales linearly with the increasing number of objects (N) in the dataset. These results show that it will be possible to mine huge datasets with this algorithm, depending on the available hardware, without having restrictions such as dataset size and other relevant criteria. Several test studies are performed and obtained results show that the algorithm possesses high and linear speedup aspects with minimum communication requirements. In the CUDA algorithms, WaveCluster

algorithm has been investigated to achieve GPU level parallelization. The CUDA implementations of two main sub-algorithms of WaveCluster approach; namely extraction of low-frequency component from the signal using wavelet transform and connected component labeling are presented. Divide and conquer approach is followed on the implementation of wavelet transform and multi-pass sliding window approach on the implementation of connected component labeling. Then, the corresponding performance evaluations are reported for each sub-algorithm with respect to increasing dataset size.

The rest of the thesis is organized as follows. Data mining is discussed in section 2. Parallel computing is introduced in section 3. Section 4 presents cluster analysis with the algorithms of sequential and parallel cluster analysis. WaveCluster approach is explained in section 5. The CUDA and MPI level parallel WaveCluster algorithms are presented in section 6. The experimental results of the CUDA algorithm and MPI algorithm are given in section 7 and 8, respectively. Finally, I conclude the thesis in section 9.

CHAPTER 2

DATA MINING

Data mining is widely used to detect the patterns and extract the meaningful information from the stored dataset. The main purpose to employ the data mining algorithm is to make estimation for future as using the past data and defining the class of each element of the dataset [12]. Today, data mining is extensively utilized to discriminate star-galaxy using neural network in the field of astronomy [30], to find association rules with respect to customer and sale information in the marketing [2], to detect the credit card fraud in the financial sector [7], to detect to predict crystal structure in the field of quantum chemistry [13] and the like.

The term of *Knowledge Discovery from Data (KDD)* is often used as a synonym of data mining. The process of knowledge discovery typically consists of seven steps [19], as it follows:

1. **Data Cleaning:** In this phase, the noise and inconsistent data are removed.
2. **Data Integration:** In case of that there are multiple data sources scattered physically, the data could be required to be combined in the data warehouse to make the data mining process more effective.
3. **Data Selection:** Relevant data are retrieved from the database for the task of analysis.
4. **Data Transformation:** The data could be transformed or consolidated into forms appropriate for mining by performing summary or aggregation operations.

5. **Data Mining:** As an essential process, intelligent methods or algorithms are applied in order to extract data patterns.
6. **Pattern Evaluation:** Truly interesting patterns representing knowledge are identified according to the interestingness measures.
7. **Knowledge Presentation:** Visualization and knowledge representation techniques are employed to present the mined knowledge to the user.

The steps of data cleaning, data integration, data selection and data transformation are regarded as a type of data preprocessing in which the data are prepared for mining. Although data mining is the step of the process of knowledge discovery, the term of data mining is used in place of knowledge discovery among scientific and commercial fields.

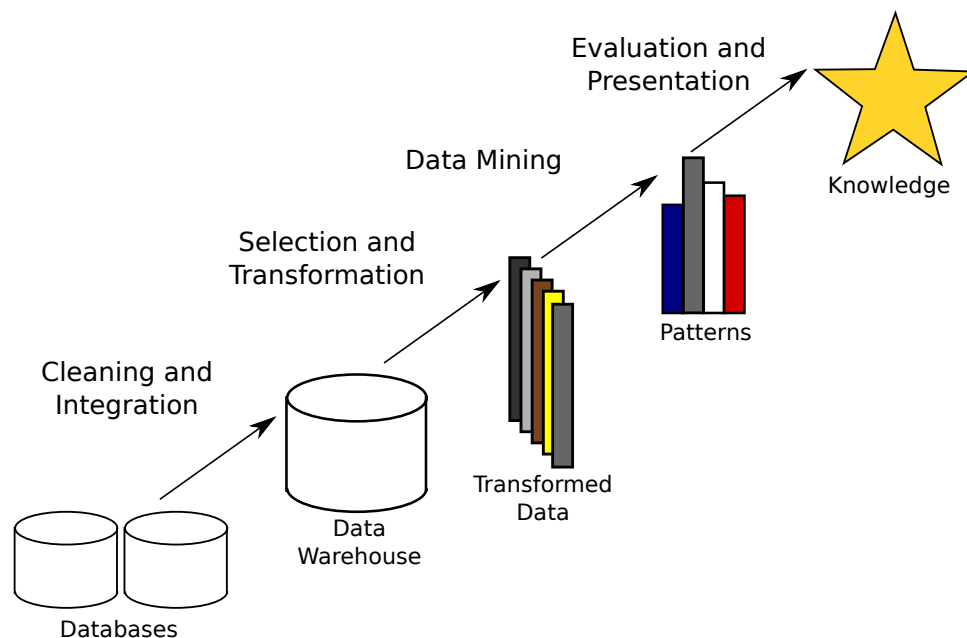


Figure 1: Knowledge Discovery Steps

2.1. Data Mining Methods

In general, data mining tasks can be classified into two categories which are descriptive and predictive mining task [19]. Descriptive mining tasks, as its name implies, characterize the general properties of the data. Whereas predictive mining task infers from the current data in order to make prediction. On the other hand, data mining algorithms can be grouped into four main groups: Classification, cluster analysis, association rule mining and regression analysis.

The purpose of classification is to determine the class of the element in the dataset. The process of classification consists of two main steps. In the first step, also known as learning or training step, analysis is made over a *training set* by means of classification algorithm. Training set has number of elements called as tuples. A tuple X is represented by an $n - dimensional$ *attribute vector* associated class labels where $X = (x_1, x_2, \dots, x_n)$ [19]. The size of *training set* is rather important to provide the accuracy of a classifier. So, in case insufficient number of elements are analysed to model the characteristics of the dataset, a known phenomena named overfitting occurs where not only the general patterns are extracted but also patterns of noisy data are incorporated into the model. In the second step, the elements are classified based on the extracted pattern information from the training step.

If the class of elements are determined in advance before performing classification process, It is called supervised learning. Decision trees and neural networks are the most common known data mining methods in supervised learning. Contrary to supervised learning, if the classes are not known in advance, the analysis process is defined as cluster analysis. Cluster analysis is one of the most widely used common technique for grouping a set of objects into classes of similar objects or clusters.

In association rule mining, the relationships are sought between the elements in the dataset and the found relationship is named as association rule. The most encountered task in association rule mining is to detect the purchasing trend of the customers in market basket analysis. By finding the rules using the past sales information, one can predict the future customer demands and construct an inventory strategy to satisfy the customer's needs, coincided with gaining more profit. The association rule should be based on a minimum support threshold and a minimum confidence value in order to accept the rule valid. One of the most well-known association rule mining algorithm is the algorithm of Apriori which was developed by Agrawal et. al. [2].

Regression analysis is used to make numerical prediction based on past data. The main aim is to find the general formula which models the relationship between one or more independent variables (predictor) and dependent (response) variable. The features of the dataset are listed in the side of independent variables and response variable is calculated to predict future value with respect to predictor variables.

CHAPTER 3

PARALLEL COMPUTING

Parallel processing approach is highly used as dividing the task into smaller subtasks and running them simultaneously to cope with memory limits and to decrease the execution time of the sequential task. In addition, larger problems, such as simulation of climate change, weather prediction and the like, can be solved in a fast and efficient manner by means of parallel computation.

Before delving into parallel computing, Flynn's taxonomy [14] will be presented in order to be able to differentiate the type of computer architecture. Flynn's taxonomy is the basic classification of computer architectures based on instruction and data stream, types of which are SISD (single instruction, single data stream), SIMD (single instruction, multiple data streams), MISD (multiple instruction, single data stream) and finally MIMD (multiple instruction, multiple data streams). As the descriptions of the classifications follow [10]:

- * **SISD**: It defines the computer which runs in a serial manner. The instructions are fetched and executed sequentially without parallelism. The computers with one processor are classified into this group.
- * **SIMD**: Multiple processors execute the same instruction on different data in a parallel manner. Graphical processing units (GPUs) and array processors are the examples of this classification.
- * **MISD**: Multiple processors execute different instructions on a single datum. It is deemed impractical in terms of possibility.

* **MIMD**: Multiple processors execute diverse instructions on diverse data. Parallel systems based on distributed memory model are classified under this model.

The classification of parallel computer architecture is based on memory sharing paradigm which are *shared memory model*, *distributed memory model* and *hybrid model*.

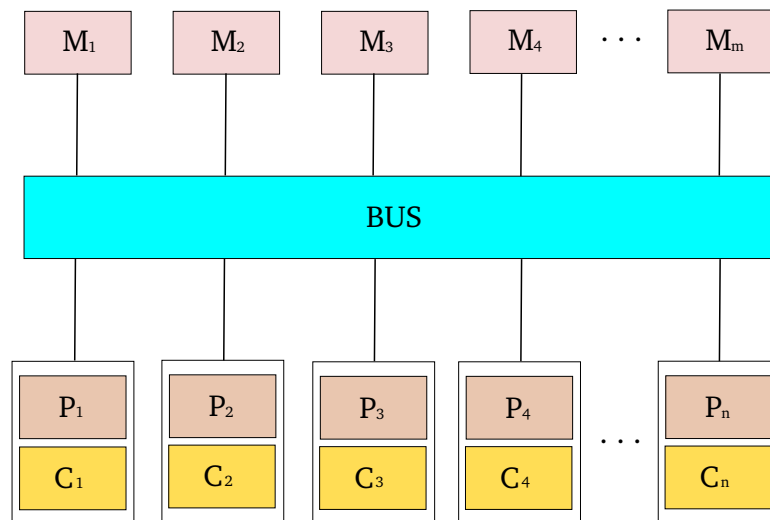


Figure 2: Shared Memory Model for m number of memory (M) and n number of procesor (P) and cache (C)

In the shared memory model, the system has its own dedicated memory which is simultaneously accessed and shared among the processors. One of the advantage of this model is that there is no extra data copy operation among diverse memories which fosters the parallel computation. Processors are connected to shared memory by shared bus. Shared memory model could be divided into Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA). In uniform memory access, each processor could access the memory in an equal time regardless of its distance to memory. The most encountered

example of this model is represented by Symmetric Multiprocessors (SMP). Nevertheless, in non-uniform model, processors have not equal access time to memory modules. Graphical processing unit (GPU) model and thread model could be regarded as shared memory model.

Distributed memory model is highly utilized in the parallel computation field. In this model, each processor are connected to each other through computer network. As its name implies, the memory modules are distributed and each processor can execute its task independently. Processors access its own local memory over local memory bus and the other processor's memory over the network infrastructure. The communication overhead and synchronization among processors are the major concerns in this model.

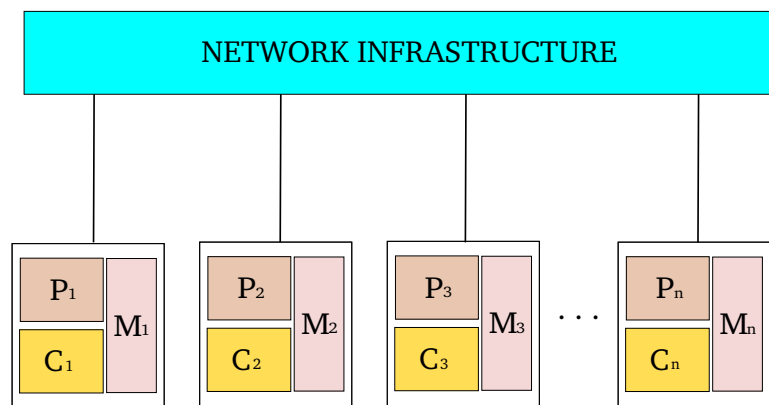


Figure 3: Distributed Memory Model for n number of memory (M), processor (P) and cache (C)

Hybrid model is the combination of the shared memory model and distributed memory model. It has been employed increasingly in lots of computer systems. According to the TOP500 [38] list released on November 2010, the hybrid architecture model based on the CPU and GPU has been employed on the most powerful computer system in the world named Tianhe-1A in China where each

computer has NVIDIA Tesla M2050 GPUs.

3.1. Message Passing Interface

Message Passing Interface (MPI) is a specification which provides mechanism among the processors to communicate with each other by simply sending and receiving messages. Message passing is a powerful and very general method of expressing parallelism [31]. Message-passing programs can be used to create extremely efficient parallel programs, and currently message passing is the most widely used method of programming many types of parallel programs [31]. To achieve this sort of parallelism, Message-Passing interface (MPI) forum consisting of numerous individuals and groups has released a specification named Message Passing Interface (MPI) in May 1994 to overcome the problem of non-standardization and portability. Although MPI specification is designed to be used for distributed memory systems, it is possible to utilize its functions for a system with shared memory model. Message Passing Interface has been implemented by means of library calls using the programming languages of C, fortran and etc.

In MPI model, there are two type of communication routines which are point-to-point communication and collective communication. MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation [4]. In point-to-point communication, processes could communicate with each other in a blocking and non-blocking way. Blocking communication means process could use the corresponding buffer safely after MPI send function *MPI_Send* returns. *MPI_Send* routine does not imply that counter receiving process has received all the message. Because

many MPI implementations use system buffer apart from the buffer of sending parameter to improve program performance by copying the buffer passed into *MPI_Send* function into system buffer. To achieve synchronous blocking message passing, *MPI_Ssend* is used. On the other hand, MPI receive function *MPI_Recv* always returns after all the buffer is filled and message is obtained totally, so it is synchronous blocking routine in nature. Communication routines occur in the context of a communicator. A communicator defines the set of processes allowed to communicate with each other belong to the same group. There is a pre-defined communicator named *MPI_COMM_WORLD* which consists of all the processes. For message identification, *tag* value is used in point-to-point communications. To receive any message regardless of the *tag*, pre-defined parameter of *MPI_ANY_TAG* should be passed to the MPI routines.

Collective communication routines are used to pass messages in a group manner and consist of the operations of broadcasting, scattering, gathering and all-to-all. Contrary to point-to-point communication, collective communication routines are not blocking and there is no message identification by means of *tag* argument. These routines have also synchronization function of *MPI_Barrier* to block all the processes until all processes reach to that routine and collective computation functions such as *MPI_Reduce* and variants to make collective computations including finding maximum, minimum, sum, product and logical operations such as operations of *AND*, *OR* and etc. on the data.

3.2. Cuda Programming Model

While message passing interface (MPI) [18] is extensively used to achieve the communication among the computers on a distributed memory system, there

has been a substantial interest to speed up the CPU-intensive tasks on many-core graphical processing units (GPUs) to utilize its enormous computational performance for shared memory system.

To do GPU computation, one remedy is to adapt the computational task to the graphics APIs such as OpenGL or DirectX, but they are not convenient for non-graphics applications and impose many hurdles to the general purpose application programmer [9]. NVIDIA introduced CUDA (Compute Unified Device Architecture) in November 2006 [29] to enable data-parallel general purpose computations on NVIDIA GPUs in an efficient way. In the CUDA programming model [29], GPU runs the computationally intensive data-parallel parts of the application as a co-processor in a SPMD (Single-Program Multiple-Data) manner while allowing the CPU to conduct concurrent tasks and sequential parts of the application. Currently, there are several but increasing number of CUDA studies which have been conducted on the field of data mining to take advantage of the high performance of GPUs [1, 23].

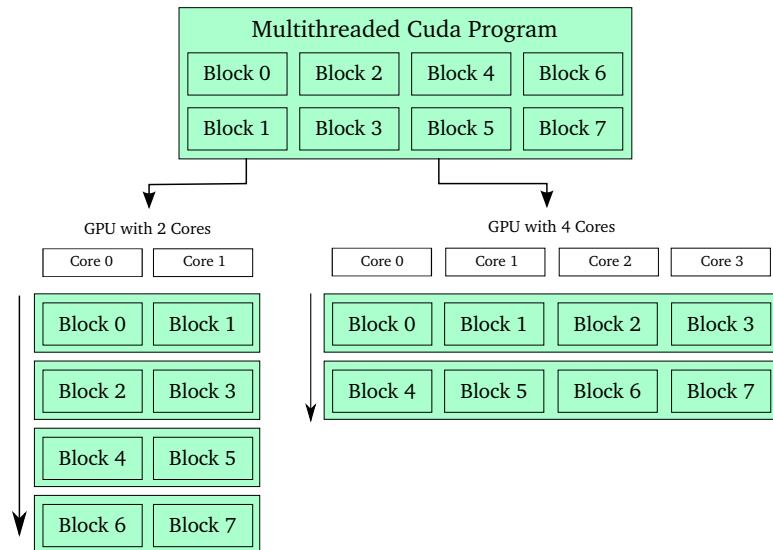


Figure 4: Automatic Scalability of CUDA Program

From the point of programmer's view, GPU is regarded as a device that runs hundreds of concurrent lightweight threads with zero scheduling overhead. In this model, all threads execute the same instruction but perform on different data concurrently. The common function, called kernel, is executed by each thread that can be programmed by ANSI C language extended with several keywords and constructs. Besides, other languages such as CUDA Fortran, OpenCL and DirectCompute, are supported by CUDA software environment [29]. When the kernel is invoked, CUDA runtime creates a grid which is composed of blocks of threads. Each thread and block is distinguished by the built-in index variables which are automatically assigned by CUDA runtime and each thread accesses its memory region using these index variables. The runtime values of grid and block sizes are specified at the kernel invocation time via language extensions.

The CUDA memory model employs three levels of memory sharing to take advantage of high memory bandwidth of the CUDA device which are local, shared and global memories [28]. Local memory provides very fast access for threads with very limited size used to store non-array local variables that are private to each thread. Shared memory is allocated for all threads within the same block which has also fast access time and is used to store frequently accessed data to save global memory bandwidth. The synchronization mechanism of shared memory is succeeded by calling `__syncthreads()` barrier function which blocks until all threads within the same block have reached this routine. There is no intrinsic synchronization mechanism among the threads in different blocks to allow thread blocks to be scheduled in any order across any number of cores automatically (automatic scalability) as shown in Figure 4 and to avoid the possibility of deadlock [28]. Finally, global memory is the slowest but has highly large memory size when compared to local and shared memories and is the only accessible memory from all threads and the host application. The contents of global memory are retained

during the lifetime of the application if not being freed intentionally.

3.3. Parallel Data Mining Approaches

There are three basic approaches for parallelization of data mining algorithms [35] , as it follows:

- * **Independent Search:** Each processor has access to the whole data set, but each mines a different part of the search space, starting from a randomly chosen initial position. The cost of an independent search strategy algorithm has the form:

$$cost_i = k_i[STEP + ACCESS] + \sigma_{pg} + \sigma \quad (3.1)$$

In Equation 3.1, k_i is the number of iterations of the whole program, $STEP$ and $ACCESS$ are the costs of the single iteration of the algorithm and its data accesses as before, σ_{pg} is the cost of sharing the answers among the processors at the end, and σ is the cost of computing the best solution.

- * **Parallelized Sequential Data Mining Algorithm:** Each processor restricts itself to generating a particular subset of the set of possible results. There are two variant: In the first, each processor generates complete results, but with restrictions on the variable values in some positions. Validating such concepts means examining only a subset of the rows of the data set. In the second, each processor generates partial results but the variable can take any values. Validating such results requires examining a subset of the columns.

The basic structure of these algorithms is first to partition and then to repeat the following steps:

- * Execute a special variant of a data mining algorithm on the entire data set, or perhaps a segment of it, and the current partial set of results to derive a new partial set of results.
- * Exchange the partial results with other processors, deleting results that are not globally correct.

The cost of a parallelized algorithm has the form:

$$cost_i = k_i[SPECIAL + ACCESS + EXCH + RES] \quad (3.2)$$

In this expression, *SPECIAL* is the complexity of a single step of the special algorithm, *EXCH* is the cost of exchanging the partial results, and *RES* is the cost of resolving the partial results or partial result set into a consistent set.

- * **Replicated Sequential Data mining Algorithm:** Each processor works on a partition of the data set and executes what is more or less the sequential algorithm. Because each processor sees only partial information, it builds entire results that are locally, but possibly not globally, correct. They are called as *approximate concepts*. Processors exchange approximate results, or facts about them, to check their global correctness.

The cost of a replicated algorithm has the form:

$$cost_i = k_i[STEP + ACCESS + rpg + RES] \quad (3.3)$$

where k_i is the number of iterations required by the parallel algorithm, *ACCESS* is the cost of data access, *rpg* is the cost of a total exchange between the processors of these approximate results and *RES* is the

computation cost of using these approximate results to compute better approximations for the next iteration.

When all these approaches are regarded, although it is not possible to say that one strategy is the best overall, there is a strong tendency for replicated strategies to be better than other parallelized strategies.

CHAPTER 4

CLUSTER ANALYSIS

Cluster analysis is the sort of data mining technique that is used to discover the similar objects based on the criterion defined by the algorithm and these objects are assigned into groups or clusters. As stated formally [17], let dataset $X \in \mathbb{R}^{m \times n}$ be the set of objects x_i , $1 \leq i \leq m$, for any dimension of n . The goal of clustering is to map group of more similar objects x_i into K nonempty clusters $C_1, C_2, C_3, \dots, C_K$. Cluster analysis is highly used in many fields such as data mining, bioinformatics and pattern recognition.

4.1. Sequential Cluster Analysis Algorithms

K-means clustering algorithm is the well-known clustering algorithm that groups the objects of the data set into k clusters (C_1, C_2, \dots, C_K) based on Euclidean distance between the data point x_i and the cluster mean μ_j . The aim of the algorithm is to minimize the variance with respect to nearest cluster centroid. The value of cluster number K is selected prior to the execution of the algorithm which is the main hurdle for the analyzer to define. The algorithm achieves the clustering by minimizing the function of sum of squares:

$$\sum_{j=1}^K \sum_{x_i \in C_j} \|x_i - \mu_j\|^2 \quad (4.1)$$

K-means algorithm has the time complexity of $O(NKT)$ where N is the number of objects, K is the cluster number and T is the iteration count [42].

The algorithm of BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [43] performs incremental clustering based on the data structure named CF (Clustering Feature) tree. A Clustering feature is a triple summarizing the information about a cluster which consists of number of data points in the cluster N , the linear sum of the N data points LS and square sum of the N data points SS . BIRCH is an incremental method that does not require the whole dataset in advance, and only scans the dataset once. After the dataset scan, it constructs the CF tree. A dense region of points is treated collectively as a single cluster. Points in sparse regions are treated as outliers and removed optionally. In the first phase of the algorithm, BIRCH loads the dataset into memory by building a CF tree. In the second phase, the condense operation is conducted by building a smaller CF tree. This operation also provides efficient memory utilization. In the phase 3, it defines the clusters and as an optional, makes cluster refining. BIRCH has the time complexity of $O(N^2)$.

CLARA (Clustering LARge Applications) is designed to handle large data sets in which performs sampling [22]. The operation of drawing an only sample of the data set is to foster the clustering analysis. It applies PAM [22] on the sample data set, and then finds the medoids of the sample. If the sample is drawn in a sufficiently random way, the medoids of the sample would approximate the medoids of the entire data set [27]. CLARA operates on multiple samples for better approximation which gives the best clustering result.

CLARANS aims to identify spatial structures that may be present in the data [27]. CLARANS uses randomized search to facilitate the clustering of a large number of objects. The algorithm studies on the efficiency and effectiveness of three different approaches to calculate the similarities between polygon objects. They are the approach that calculates the exact separation distance between two polygons, the approach that overestimates the exact distance by using the

minimum distance between vertices, and the approach that underestimates the exact distance by using the separation distance between the isothetic rectangles of the polygons. The results has shown that it is more efficient than the algorithms of PAM and CLARA and calculating the similarity between two polygons by using the separation distance between the isothetic rectangles of the polygons is the most efficient and effective approach.

DBSCAN is the clustering algorithm [11] which relies on a density-based notion of clusters. It is designed to discover clusters of arbitrary shape. DBSCAN requires only one input parameter and supports the user in determining an appropriate value for it. The key idea is that for each point of a cluster the neighborhood of a given radius has to contain at least a minimum number of points, i.e. the density in the neighborhood has to exceed some threshold. The shape of a neighborhood is determined by the choice of a distance function for two points p and q , denoted by $dist(p, q)$. For instance, when using the Manhattan distance in 2D space, the shape of the neighborhood is rectangular. Note, that this approach works with any distance function so that an appropriate function can be chosen for some given application. One drawback of this algorithm is that there are at least a minimum number ($MinPts$) of points in an Eps-neighborhood of that point which is hard to determine like k-means algorithm. The algorithm has the time complexity of $O(NlogN)$.

4.2. Parallel Cluster Analysis Algorithms

In recent years, there have been increasing amount of research in implementation of the parallel clustering algorithms to speedup the corresponding sequential clustering algorithm.

Zhang et al. [44] proposed a parallel implementation of the K-means clustering algorithm. They adopted master-slave model and introduced dynamic load balance for enhancing the efficiency of parallel K-means. At the beginning of the algorithm, the master processor divides the sample and sends them accompany with cluster center to each slave one. Each slave processor receives the dataset with size of SN where N is the entire data size and P is the slave number. Each slave processor executes clustering operation for received data and returns the clustering results to the master. The master partitions a new sub-dataset and sends it to the slave. This is continued until there are no more data in the master and all the results are returned. So far an iterative process is completed. If the value difference between twice error-squared function than ϵ where $\epsilon \leq 10^{-6}$ then the algorithm is ended. Otherwise the master processor computes the new cluster center again and a new iterative procedure begins. This algorithm has no acceleration ratio due to excessive communication time over computation time when dataset scale between 100K and 700K. After dataset scale reaches to the value of 800K, the beneficial acceleration ratio is obtained as computation time being larger than communication time.

Arlia and Coppola [3] presented their results concerning the parallelization of DBSCAN clustering algorithm based on master-slave model. In their algorithm, The Master module performs cluster assignment, while the Slave module answers neighborhood queries using the R*-Tree. Reading spatial information is decoupled from writing labels, and the Slave has a pure functional behaviour. Having restructured the algorithm to a Master-Slave cooperation, its structure must be expressed using the skeletons of language. There is pipeline parallelism between Master and Slave, and functional independent replication can be exploited among multiple Slaves. Cluster labels are kept and checked in the Master, which can quickly become a bottleneck. This algorithm gives consistently good speed-ups

but needs verification for larger input sets.

Boutsinas and Gnardellis [6] reported a clustering methodology for scaling up any clustering algorithm. The main idea of this methodology is based on running the clustering algorithm s times on subsets of the sample of data in parallel. This algorithm requires construction of a new table called *Meta-table* by merging all partial local results and applying clustering algorithm over the *Meta-table* globally. This procedure continues iteratively until certain stopping criteria are sustained.

CHAPTER 5

WAVECLUSTER APPROACH

WaveCluster algorithm is a novel, grid-based clustering algorithm based on wavelet transform [34]. WaveCluster algorithm conforms all requirements of being good clustering algorithm such as discovery of clusters with arbitrary shape, ability to handle outliers and handling of multi-dimensional datasets. The goal of this algorithm is to map group of more similar or dense spatial objects into disjoint clusters on transformed feature space using wavelet transform. The collection of objects in the feature space composes an n -dimensional signal. Examinations have shown that WaveCluster algorithm outperforms BIRCH [43] and CLARANS [27] as much as 8 to 10 times, and 20 to 30 times [34], respectively.

WaveCluster approach consists of two main sub-algorithms which are wavelet transform and connected component labeling algorithm. Wavelet transform is a mathematical transformation tool which are used to decompose the signal of $f(n)$ into average subband (approximation coefficients) and detail subbands (detail coefficients). Wavelet transform could be applied to the signal multiple times on average subbands to take advantage of its multi-resolution analysis feature. The operation of recursively decomposing the signal by applying averaging and differencing the coefficients, is called a filter bank. Each operation of wavelet transform constitutes half objects in the transformed feature space for each dimension when compared to previous feature space. For one dimensional wavelet transform, let V^0 be the average subband of the feature space at last scale and V^j be the average subband of the signal which has number of 2^j objects. Each objects in the average subband of the feature space V^j also contained in

the average subband of the feature space V^{j+1} . Hence, the feature spaces are nested [36], that is:

$$V^0 \subset V^1 \subset V^2 \subset \dots \quad (5.1)$$

In discrete wavelet transformation (DWT), the function $f(n)$ is discrete where $f(n) = f(x_0 + n\Delta x)$ for some $x_0, \Delta x$ and $n = 0, 1, 2, \dots, M - 1$ ($M = 2^J$) and the wavelet operation could be regarded as a convolution operation using filtering techniques which yields wavelet coefficients [16]. The DWT filter bank of one-dimensional signal has been depicted for two scale in Figure 5 where h_φ is low pass filter and h_ψ is high pass filter. The signal components are represented as W_φ and W_ψ for low-frequency component (average subband) and high-frequency component (detail subband), respectively.

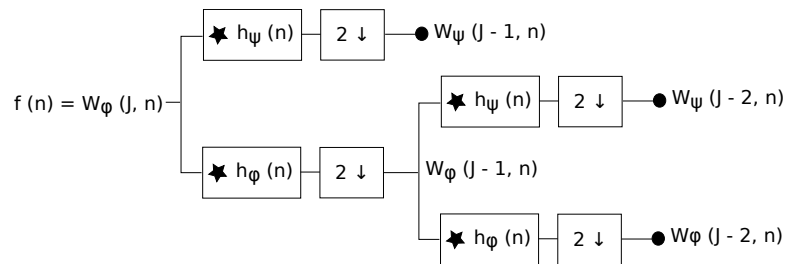


Figure 5: Two-scale Discrete Wavelet Transform Filter Bank

There are many popular wavelet algorithms including Haar wavelets, Daubechies wavelets and Mexican Hat wavelets. In the experiments, Haar wavelets [36] are selected because of its simplicity, fast and memory-efficient characteristics [21]. It should also be mentioned that it is necessary to scale source dataset linearly to power of two for each dimension, since the signal rate decreases by a factor of two at each level in the discrete wavelet transform. For Haar wavelets, the feature space are convoluted with 2×2 Haar matrix in a recursive manner, as it follows:

$$H_2 = \frac{1}{\sqrt{2}} \begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix} \quad (5.2)$$

In WaveCluster approach, cluster analysis is performed by applying connected component labeling algorithm on the objects located in the transformed feature space (low-frequency component of the signal). The low-frequency component is the product of the operation of wavelet transform, whereas the connected components are found using connected component labeling algorithm with respect to the feature of pixel connectivity for each direction. Cluster analysis is conducted recursively on the transformed feature space that provides multi-resolution feature in clustering as finding clusters from fine (low scale value) to coarse (high scale value). In Figure 6, the effect of the wavelet transformation on source dataset (see Figure 6(a)) is demonstrated.

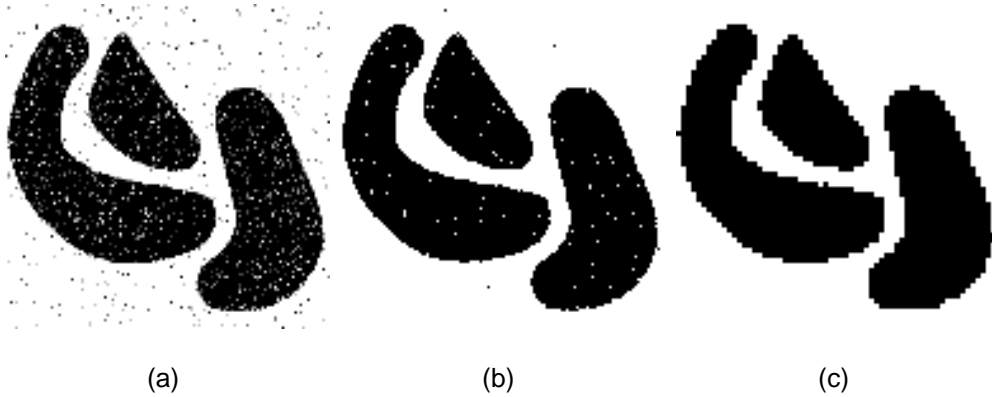


Figure 6: WaveCluster algorithm multi-resolution property. (a) Original source dataset. (b) $\rho = 2$ and 6 clusters are detected. (c) $\rho = 3$ and 3 clusters are detected. (where ρ is the scale level)

The results of WaveCluster algorithm for different values of ρ are shown in Figures 6(b) and 6(c); where ρ (scale level) represents how many times wavelet transform is applied on the feature space. There are 6 clusters detected with

$\rho = 2$ (Figure 6(b)) and 3 clusters are detected with $\rho = 3$ (Figure 6(c)). In the performed experiments, connected components are found on average subbands (feature space) using classical two-pass connected component labelling algorithm [33] at different scales.

Algorithm 1 WaveCluster Algorithm

- 1: Quantize feature space, then assign objects to the units.
 - 2: Apply wavelet transform on the feature space.
 - 3: Find the connected component components (clusters) in the subbands of transformed feature space, at different levels.
 - 4: Assign label to the units.
 - 5: Make the lookup table.
 - 6: Map the objects to the clusters.
-

WaveCluster algorithm contains three phases. In the first phase, algorithm quantizes feature space and then assigns objects to the units. This phase also affects the performance of clustering for different values of interval size. In the second phase, discrete wavelet transform is applied on the feature space multiple times. WaveCluster algorithm gains the ability to remove outliers with wavelet transform and detects the clusters at different levels of accuracy (multi-resolution property). Following the transformation, dense regions (clusters) are detected by finding connected components and labels are assigned to the units in the transformed feature space. Next, a lookup table is constructed to map the units in the transformed feature space to original feature space. In the third and last phase, WaveCluster algorithm assigns the cluster number of each object in the original feature space.

CHAPTER 6

PARALLEL WAVECLUSTER ALGORITHMS

Despite effectiveness of WaveCluster algorithm, execution time of the algorithm has become a serious concern when dataset size is large. In this section, parallel WaveCluster algorithms based on the message passing model for distributed memory multiprocessors [40] and CUDA algorithms of WaveCluster approach for shared memory system [41] are presented.

6.1. Parallel Wavecluster Algorithms on Shared Memory Architecture Using CUDA

In recent years, CUDA algorithms have been gaining interest a lot among the researchers to speed-up the sequential algorithms in a parallel manner by utilizing powerful and cutting-edge graphical processing units (GPUs). In this section, parallel WaveCluster approach based on CUDA model is presented. Since the algorithms of extraction of low-frequency component by means of wavelet transform and connected component labeling are the fundamental sub-algorithms of WaveCluster approach, these algorithms have been implemented with respect to CUDA model in the study.

6.1.1. Implementation of low-frequency component extraction of the signal

Discrete wavelet transform is the core part of the WaveCluster algorithm which takes advantage of its multi-resolution feature. As mentioned previously, since WaveCluster tries to find dense regions over low-frequency component of the signal, only the extraction phase of the low-frequency component is implemented. The algorithm is designed for 2-dimensional feature space to ease algorithm demonstration, but it can be expanded to many dimensional wavelet transform unless the limit of shared memory allocation is exceeded. The algorithm can be regarded as simple convolution operation as well.

There are many wavelet types exist such as Haar, Daubechies, Morlet and Mexican hat. In the implementation, Haar wavelet is used, because its initial window width is two which leads us to less waste of shared memory usage. The host function calls kernel function as much as the value of scale and kernel function returns the lower frequency representation (V_{j-1}) of the feature space (V_j) at each iteration.

Divide and conquer approach is followed in the CUDA implementation of this process. Each thread is responsible to calculate one approximation value extracted from local disjoint 2x2 square-shaped points of feature space. Before kernel invocation, input feature space is transferred from host memory to global memory of the device and output buffer is allocated in device memory to store transformed feature space. In the kernel, input feature space is transferred from global memory into shared memory of buffer I to make data access efficient. Each thread firstly applies one-dimensional wavelet transform to each column of local feature space and stores intermediate values in the shared memory buffer of H . The final approximation value is eventually calculated by applying second one-dimensional wavelet transform to each row of points on H . So, buffer H is used to store temporal results. If the aim is only to calculate approximation representation of the signal, the operation can be halted here. However, the implementation also

Algorithm 2 CUDA Algorithm of Low-Frequency Component Extraction

Require: $V^j, lastlevel, threshold$

Ensure: V^{j-1}

```
1: declare  $I[dim.y * 2][dim.x * 2]$  in shared memory
2: declare  $H[dim.y * 2][dim.x]$  in shared memory
3: load thread-related disjoint 2x2 points of  $V^j$  into buffer  $I$ 
4: apply one-dimensional wavelet transform to each column of points of 2x2 field
   and store values into buffer  $H$ 
5: apply one-dimensional wavelet transform to each row of points over  $H$  and
   store approximation value in local memory  $m$ 
6: if  $lastlevel = true$  then
7:   if  $val > threshold$  then
8:      $m \leftarrow MAXFLOAT$ 
9:   else
10:     $m \leftarrow threadindex$ 
11:   end if
12: end if
13:  $V^{j-1}[threadindex] \leftarrow m$ 
```

makes the data suitable for usage in the connected component labeling algorithm. For that purpose, it performs thresholding operation at the last iteration (last level of wavelet transformation) to assign maximum float value to background points and unique thread index value to foreground points with respect to threshold value. This operation also removes outliers on the transformed feature space.

6.1.2. Implementation of connected component labeling

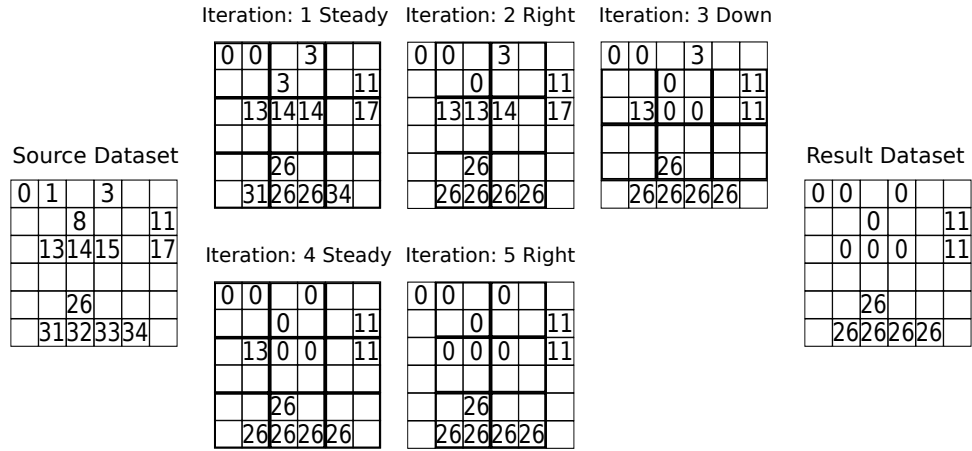


Figure 7: Iteration Demonstration of Connected Component Labeling CUDA Algorithm

In this algorithm, multi-pass CCL algorithm based on sliding window approach is presented which groups the points with respect to pixel connectivity by sliding many windows over feature space concurrently. Each thread is responsible of 2x2 square-shaped field forming a window where calculates the minimum value of these points and assigns the minimum value to its foreground points. As a prerequisite, each foreground point is expected to be assigned unique increasing value in a left-to-right and top-to-bottom manners. Maximum float number are assigned to background points to ensure consistency in the algorithm. The algorithm consists of three sub-operations. In the first sub-operation, the windows stay steady and in the second and third sub-operation, windows move to the right and down respectively. Thus, the minimum value of connected points can be propagated in all directions. The algorithm continues to iterate until no point value changes in all three sub-operations. For that reason, additional 3 iterations are executed to detect this stopping criterion. If any thread changes any value of the points, the variable of *ischanged* is assigned true value that kept in the global memory. The host function of the kernel keeps track of the *ischanged* variable

for each sub-operation and then resets its value to false before kernel invocation. The execution time of the algorithm is highly dependent to the maximum distance between two points within cluster. Since GPU runs each operation very fast in a parallel manner, the algorithm finishes the execution in a reasonable amount of computation time. The algorithm demonstration is depicted in Figure 7 where each window is surrounded with bold line.

Algorithm 3 CUDA Algorithm of Connected Component Labeling

```

1: if windowdirection = RIGHT AND last thread in row then
2:   return
3: end if
4: if windowdirection = DOWN AND last thread in column then
5:   return
6: end if
7: declare  $I[dim.y * 2][dim.x * 2]$  in shared memory
8: calculate startPositionIndex of the local buffer with respect to direction
9: load disjoint 2x2 points(window) into buffer  $I$  using startPositionIndex
10: find minimum value among points in the window stored on buffer  $I$ 
11: assign minimum value to foreground points
12: if anyvalueofpointsischanged then
13:    $ischanged \leftarrow true$ 
14: end if

```

6.2. Parallel Wavecluster Algorithm on Distributed Memory Architecture Using MPI

In this section, parallel WaveCluster algorithm for distributed memory architecture [32] is presented. The master-slave model and SPMD (Single Process,

Multiple Data) techniques are applied to the algorithm as the parallelization technique. In this approach, each copy of the single program runs on processors independently and communication is provided by the manner of sending and receiving messages among nodes. The terms of processor, core and node will be used interchangeably in the text. In the computer cluster system, each workstation is connected with an underlying Ethernet network. Datasets are stored in I/O (Input/Output) server via network file system, so that each node in the virtual system can have an access to datasets in a shared manner. Communication requirements between master and slave nodes are managed by using Message Passing Interface (MPI) [18]. MPI is a specification rather than an implementation. OpenMPI [15] implementation is employed because of its high-performance and it is one of the most widely used MPI implementation.

$D_{0,0}$	$D_{1,0}$	$D_{2,0}$	$D_{3,0}$
$D_{0,1}$	$D_{1,1}$	$D_{2,1}$	$D_{3,1}$
$D_{0,2}$	$D_{1,2}$	$D_{2,2}$	$D_{3,2}$
$D_{0,3}$	$D_{1,3}$	$D_{2,3}$	$D_{3,3}$

(a)

$D_{0,0}$	$D_{1,0}$
$D_{0,1}$	$D_{1,1}$
$D_{0,2}$	$D_{1,2}$
$D_{0,3}$	$D_{1,3}$

(b)

Figure 8: Decomposition of large dataset D with 2 dimensions for varying number of processors (np). (a) $np = 16$ (b) $np = 8$

The replicated approach [35] has been followed in the parallel implementation of WaveCluster algorithm. In this approach, each processor works on a specific partition of the dataset and executes nearly identical code segments of the algorithm. The obtained results from each processor are locally correct, but possibly not globally. Hence, processors are subject to exchange their local results and then to check the correctness. The replicated approach is not particularly

novel, but it is often the best way to increase performance in a data-mining application [35].

Algorithm 4 Parallel WaveCluster Algorithm

- 1: parent broadcasts parameters of MC and ρ
 - 2: **for** $i = 1$ to processorsize **do**
 - 3: parent sends position vector of subset to processor i
 - 4: **end for**
 - 5: **for all** processor i , in parallel **do**
 - 6: read and copy local dataset into memory using position vector
 - 7: apply d dimensional wavelet transform on a d dimensional local dataset ρ times
 - 8: find connected components on transformed local feature space
 - 9: assign cluster number c_n to the units where $(i - 1) * MC \leq c_n \leq i * MC$
 - 10: send clustering border vector of local feature space to parent
 - 11: **end for**
 - 12: **for** $i = 1$ to processorsize **do**
 - 13: parent creates merge table for processor i with respect to the adjacency relations at clustering border vectors
 - 14: parent sends merge table to processor i
 - 15: **end for**
 - 16: **for all** processor i , in parallel **do**
 - 17: update cluster numbers of units of local feature space
 - 18: make the lookup table
 - 19: map the objects to the clusters using lookup table
 - 20: write clustering results
 - 21: **end for**
-

In the employed algorithm (see Algorithm 4), dataset is partitioned evenly among processors in a grid manner as shown in Figure 8. Each subset of the dataset has a size of N/P ; where N is the total size of the dataset and P is the total number of processors. In the initialization phase, the parent node broadcasts allowed maximum total number of clusters per node (MC) and scale level (ρ) values to compute nodes (line 1). Then, parent node sends corresponding position vectors of subsets of dataset to each node although it is possible for each node to access to full dataset from I/O server over network. So that each compute node can retrieve only required data to local memory. Two benefits are acquired by this approach; reducing the bandwidth usage and overcoming the restriction of limited amount of local memory for huge datasets. This position vector information is also received by the parent node itself for admitting the parent node into parallelization stage. Such that the possible idleness behavior of parent node during calculation stages is prevented (lines 2-4). After reading and copying the local dataset into local memories, processors applies d dimensional discrete wavelet transform on a d dimensional local dataset ρ times. This transform process highly affects the total execution time of parallel WaveCluster algorithm according to the results of the experiments. Since it yields smaller size of feature space at each iteration of wavelet transform, subsequent operations works on smaller portions of data. In the next step, processors detect dense regions on transformed local feature space by finding connected components and assigning cluster number c_n to the the units where $(i - 1) * MC \leq c_n \leq i * MC$. The parameter of MC is utilized to assign uniquely cluster number across processors. For that reason, the value of MC may be selected according to the estimation of maximum total number of clusters per local dataset (lines 5-11). At the moment of last phase, detected clusters by each processor might be valid locally but may not be globally. Therefore, a merge operation is needed to detect clusters globally.

After processors send the border vector containing of cluster numbers of the units on borders for the local feature space to parent node, parent node creates a merge table for each processor with respect to the adjacency relations at the units of local feature space borders. This requirement for sustaining consistency brings a barrier primitive. Accordingly all processors wait the parent node to receive merged table (lines 12-15). Then, processors update cluster numbers of units on local feature space. Finally, processors map the objects on original feature space to the clusters using lookup table (lines 16-21). In lookup table, each entry specifies relationships of one unit in the transformed feature space to the corresponding units of the original feature space. The demonstration of the algorithm is depicted in Figure 9.

In the two-pass connected component labeling, the implementation approach of the operations of union and find is an important factor that highly affects the performance of the WaveCluster algorithm. In the implementation, the union-find data structure is utilized to perform these operations efficiently [33]. This data structure is implemented as a vector array that represents a tree structure in which first set contains the labels of the objects and the second set contains the parent (target) labels of the objects. With the union-find data structure, the find procedure runs in highly efficient manner when compared to the linked-list implementation. The vector array of union-find data structure has the size of MC elements.

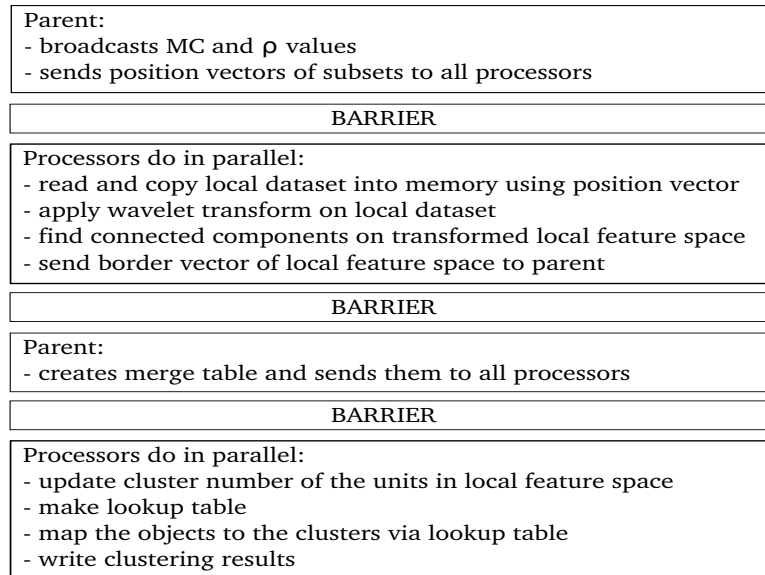


Figure 9: Demonstration of Parallel Wavecluster Algorithm based for Distributed Memory Architecture

In the constructing process of the merge tables which is performed by the parent processor for each processor, if parent processor finds different labels among the neighbor units of the clustering border vectors, the smaller of two labels is assigned to the current units. To detect complex-shaped clusters, this procedure is applied recursively for all clustering border vectors of each dimension. Each entry in the merge table data structure has the member of index label and the target label to resolve globally correct cluster number. When merging condition occurs, all occurrences of the old label of the local units are changed with the new label value by the processors in parallel.

CHAPTER 7

EXPERIMENTAL RESULTS OF THE CUDA ALGORITHMS

The speedups of CUDA algorithms have been investigated and compared to the sequential ones. The experiments were conducted on a Linux workstation with 2 GB RAM, Intel Core2Duo (2 Cores, 2.4 GHz, 4MB L2 Cache) processor and NVIDIA GTX 465 (1 GB memory, 352 computing cores, each core runs at 1.215 GHz) with compute capability 2.0 and runtime version 3.10. Two-dimensional synthetic dataset has been used in the experiments.

For the parallel experiments, the larger datasets of the source datasets are obtained by making multiple copies of the source dataset. The number of clusters with the points are increased in a linear fashion.

Execution times (in microseconds) and corresponding kernel speedup values for the low-frequency extraction algorithm are presented in Table 1. The results indicates that at most 107.10 speedup have been achieved for the CUDA algorithm and high speedups for increasing number of points.

The sequential code of Connected Component Labeling (CCL) is implemented using union-find data structure [33] which is highly fast and efficient with respect to linked-list implementation of CCL algorithm. Table 2 shows the performance results of the CCL CUDA algorithm. At most 5.56 speedup value is obtained as dataset size increases in the dataset.

In CUDA model, the input buffer is needed to be transferred from host memory to device memory before employing GPUs to execute kernel (CUDA function) and vice versa for obtaining the outputs of the kernel using CUDA functions of

Table 1: Performance results of CUDA and CPU versions of Low-Frequency Component Extraction for $\rho = 1$ (times in microseconds)

Dataset Size	Number of Points	Execution Time (CPU)	Execution Time (GPU)	Kernel Speedup
256	65536	496	33	14.17
512	262144	1987	53	37.49
1024	1048576	7868	113	69.62
2048	4194304	31045	319	97.31
4096	16777216	123279	1151	107.10

Table 2: Performance results of CUDA and CPU versions of Connected Component Labeling (CCL) for $\rho = 1$ (times in microseconds)

Dataset Size	Number of Points	Execution Time (CPU)	Execution Time (GPU)	Kernel Speedup
256	65536	1082	1242	0.87
512	262144	4133	1503	2.74
1024	1048576	15555	3891	3.99
2048	4194304	57608	9689	5.94
4096	16777216	174586	31357	5.56

*cudaMemcpy**. This extra transfer operation over data bus (such as PCI-E bus) is known as fundamental drawback of CUDA model when compared to CPU

Table 3: Aggregate Speedup Results of CUDA Algorithms for $\rho = 1$
(times in microseconds)

Dataset Size	Number of Points	PCI-E Transfer Time	Aggregate Speedup
256	65536	54522	0.02
512	262144	55306	0.10
1024	1048576	57904	0.37
2048	4194304	67182	1.14
4096	16777216	103664	2.18

execution. As far as developed parallel CUDA algorithms are concerned, this phenomenon has been also observed. As seen in Figure 3, aggregate speedups are decreased when the performance of data transfer between host and device memory is counted into speedup calculation. Aggregate speedup is calculated as seen in Formula 7.1.

$$Agg.Speedup = CPUExec.Time / (TransferTime + GPUExec.Time) \quad (7.1)$$

CHAPTER 8

EXPERIMENTAL RESULTS OF THE MPI ALGORITHM

All the experiments were performed on a cluster system having 32 cores with a 2.8 GHz clock speed computers where each compute node has 4 cores with fast Ethernet (100 Mbit/sec) and Gbit Ethernet (1 Gbit/sec) cards as underlying communication infrastructure [5]. Two source datasets with different object distribution characteristics (sparsity, see Figure 10) were used in the experiments. SourceDataset1 (SD1) was originally used in Ref. [34] in which objects are evenly distributed across the dataset. On the other hand, the SourceDataset2 (SD2) has data sparsity property. The aim of selecting SD2 is to measure the performance of the algorithm on the datasets having unevenly distributed sparse data characteristic. These datasets are also adapted to the aims by scaling linearly for obtaining very large/huge datasets. Thereafter, these datasets are called as DS(1_ or 2_)8 (with 67108864 objects), DS(1_ or 2_)16 (with 268435456 objects), DS(1_ or 2_)32 (with 1073741824 objects) and DS(1_ or 2_)65 (with 4294967296 objects). An object value occupies 4 bytes that is stored in an integer array.

The behavior and performance of WaveCluster clustering approach based on wavelet transforms with the developed parallel algorithm are investigated and obtained results are presented in terms of execution times with and without I/O costs, speed-up and efficiency for each source datasets (SD1 and SD2) with respect to varying dataset sizes (DS8, DS16, DS32, DS65). The results for datasets DS8, DS16, and DS32 were obtained with 1, 2, 4, 8, 16 and 32 core cases. Furthermore, the case as beyond the limit of dataset size that does not

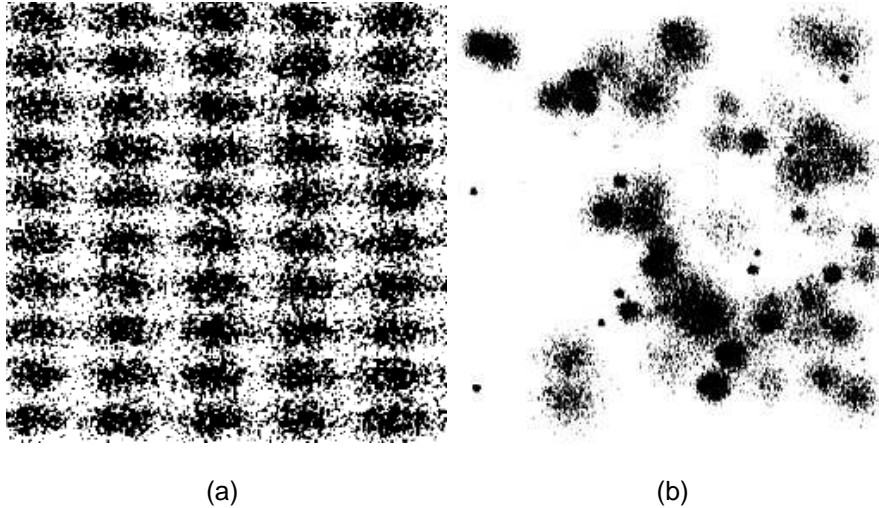


Figure 10: Sample datasets used in experiments (a)SourceDataset1 (SD1) (b) SourceDataset2 (SD2)

fit into memory of a single processor within the available hardware is studied by running huge dataset (DS65) with 8, 16 and 32 cores.

Table 4 shows the execution times of the developed algorithm with and without I/O costs for $\rho = 3$ (as scale level) for SD1 using both 100 Mbit Ethernet and 1 Gbit Ethernet devices. The reason of presenting execution times results only for $\rho = 3$ in this table is that this scale level represent the behavior of the parallel WaveCluster algorithm as most conveniently in the aspects of wavelet transforms and connected component labelling. The experiment results obtained using 100 Mbit Ethernet are given to demonstrate the performance of the algorithm on low-speed network infrastructure. Execution times are obtained by utilizing MPI routine `MPI_Wtime`.

A shortcoming of dealing with large datasets in WaveCluster clustering approach is that the required I/O operations may dominate the execution time. This situation can be seen from Table 4 for both fast and giga Ethernet cases.

Table 4: Execution times (in seconds) with and without consideration of I/O times for SD1 with varying dataset sizes and number of processors (np) for $\rho = 3$ using 1 Gbit and 100 Mbit Ethernet controller cards.

Dataset/np	1	2	4	8	16	32
DS1_65 (I/O) 1 Gb	-	-	-	807	657	574
DS1_65 (I/O)	-	-	-	5832	5760	5720
DS1_32 (I/O) 1 Gb	777	559	274	204	165	143
DS1_32 (I/O)	1720	1569	1496	1457	1439	1430
DS1_16 (I/O) 1 Gb	174	125	61	43	35	30
DS1_16 (I/O)	444	392	374	364	359	357
DS1_8 (I/O) 1 Gb	38	28	22	17	11	7
DS1_8 (I/O)	107	97	92	90	89	88
DS1_65 1 Gb	-	-	-	10.55	5.77	3.30
DS1_65	-	-	-	17.59	9.70	5.21
DS1_32 1 Gb	15.60	8.69	5.21	2.60	1.41	0.83
DS1_32	35.19	17.96	9.15	4.41	2.50	1.30
DS1_16 1 Gb	3.90	2.18	1.32	0.66	0.35	0.18
DS1_16	8.83	4.53	2.25	1.06	0.57	0.33
DS1_8 1 Gb	0.98	0.57	0.32	0.16	0.09	0.05
DS1_8	2.22	1.11	0.55	0.27	0.14	0.08

The time spent for I/O operations is mainly sourced from loading the data to local memories of compute nodes during initialization stage and also some comparatively very small time amount is spent for exchanging border vectors between nodes during computations.

It could be possible to debate that the improvement with parallelization may not be beneficial as much as expected due to the high I/O requirements during the initialization stage. However, this limitation could be turned into a gain since one may utilize two aspects that the parallelization may bring benefits for data processing.

The first benefit is obtained by applying a parallelization scheme directly to CPU-bound operations and the second one is that distributing the data for IO-bound operations and processing in the context of distributed computing. The benefit obtained from the former one in the study seems to be relatively small due to the IO-bound nature of the WaveCluster approach. But, it should be mentioned that there is a steady decrease at execution time without I/O times by increasing number of processors. The later one brings us the capability of handling larger sizes of datasets that it could not be possible to process sequentially due to memory limitations. Furthermore, the execution times are decreased considerably.

This divide-and-conquer type data distribution is achieved in the developed algorithm just by sending corresponding global data pointers to compute nodes. Each compute node loaded necessary data to local memory from file server via network file system. Consequently, the time spent for I/O requirements is divided among processors. This expected benefit is obtained from Gbit Ethernet rather than fast Ethernet. As seen from Table 4, the values for I/O times have a tendency to decrease with increasing number of processors for Gbit Ethernet due to higher bandwidth.

Execution times with I/O per object per processors for each datasets at fast and gigabit Ethernet cases are calculated and depicted in Figure 11 for varying size of processors at scale level, $\rho = 3$. By this normalization, it is observed that the time spent for reading and processing the data is independent of dataset

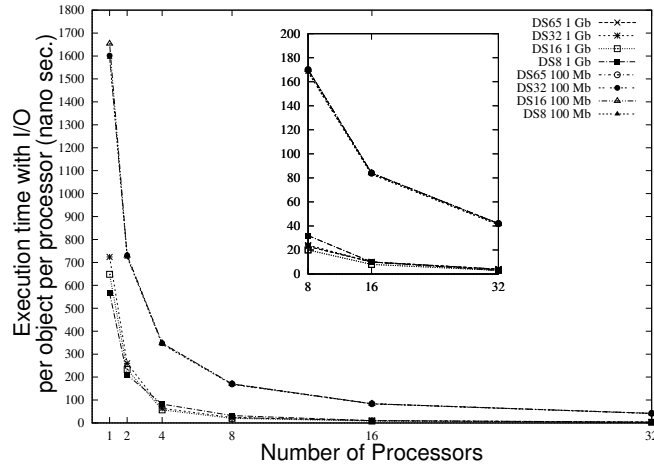


Figure 11: Execution times with I/O per object per processors for SD1 with respect to dataset sizes for 100 Mb (upper data line) and 1Gb (lower data line) Ethernet at scale level, $\rho = 3$ vs number of processors (np). Inset: That of np=8, 16, 32.

size and also indicates good load balancing feature of the developed algorithm on SD1.

Overall, the conclusion is that the total execution time is decreased considerably by parallelization of CPU-bound and/or IO-bound operations and applying a parallel algorithm to WaveCluster approach is beneficial. This conclusion is more convenient for the case of Gbit Ethernet since I/O times are highly depends on the speed of network infrastructure. As the parallel WaveCluster algorithm is executed using faster network infrastructure such as 10/100 Gbit Ethernet, Myrinet, Quadrics etc. and/or using faster storage networks, it is expected that this sort of shortcoming (I/O time limitation) could be surmounted to some extent. All the timing and performance results/figures reported from now on have been obtained by the usage of 1 Gbit Ethernet.

Although a considerable improvement with parallelization is obtained by

distribution of IO-bound operations, those I/O times are not considered for the presented figures in the rest of the text. The reason for this is that the time spent for reading of dataset and writing of result from/to the disc is highly dependent on numerous factors including read-write speed of the disc, speed of the network, format of the input/result file and the purpose of clustering process. Hence, there is only focus of the parallelization behavior of the developed parallel WaveCluster algorithm.

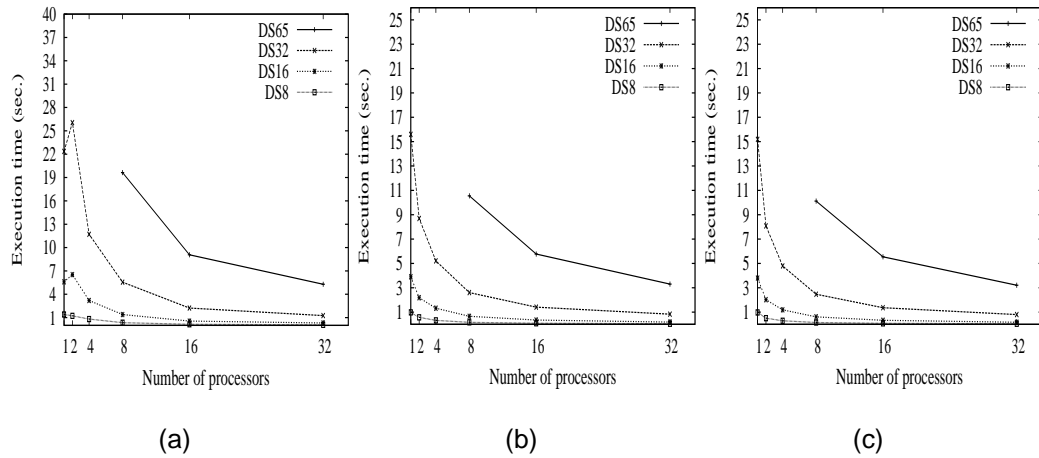


Figure 12: Execution times for SD1 with varying dataset sizes (DS8, DS16, DS32 and DS65) and number of processors (1, 2, 4, 8, 16, 32; 1 Gbit Ethernet) at different scale levels (ρ); (a) $\rho = 1$, (b) $\rho = 3$ and (c) $\rho = 5$.

The behavior of execution times by increasing number of processors for data sets DS8, DS16, DS32 and DS65 of SD1 with $\rho = 1$, $\rho = 3$ and $\rho = 5$ are plotted in Figure 12. The expectation of decrease in execution time by increasing number of processors is satisfied for almost all the cases except one specific case for SD1. In Figure 12(a) ($\rho = 1$), execution time of 2-core case is found as greater than that of single core for DS8, DS16 and DS32 datasets. This exceptional case

can be explained due to the additional communication time of sending clustering border vector of feature space and time of updating cluster number of units on local feature space and computation time of constructing merge table for each processor. But, updating cluster numbers of feature space has the main weight for this performance degradation. As the number of processor increases, less time is required in the operation of updating cluster numbers of reduced feature space per cores. Accordingly, very good timing values and linear scaling speedup performance has been obtained. Update procedure is implemented as scanning transformed feature space and replacing all occurrences of old labels with new ones. This procedure is performed in a fast manner via programming language data type of a pointer which points to cluster label of all associated units.

Speedup (S_p) is defined as the ratio of computation times of single processor and of P processors. The obtained speedup ratios for DS8, DS16, DS32 and DS65 of SD1 with $\rho = 1$, $\rho = 3$ and $\rho = 5$ are presented in Figures 13(a), (b) and (c), respectively, for varied number of processors and dataset sizes. It is clearly seen from these figures that the developed parallel WaveCluster algorithm scales almost linearly. In Figure 13(a), the speedup line for DS65 is higher than the lines for other datasets. The number of operations for that $\rho = 1$ scale level is more than $\rho = 3$ and $\rho = 5$ scale levels. This elevation of the line for DS65 dataset at most time consuming scale level value confirms the suitability of the parallel WaveCluster algorithm for very large/huge datasets.

The speedup values for the other datasets in Figure 13(a) are almost same up to 8 processors and for 32 processors. The separation of the lines at the processor size 16 can be explained as the unsustainable balance between the computed local data and communicated data. There are comparatively less amount of communication for 8 cores while less amount of calculation for 32 cores. This situation changes for the next scale level value ($\rho = 3$) as being closer at 16 and 32

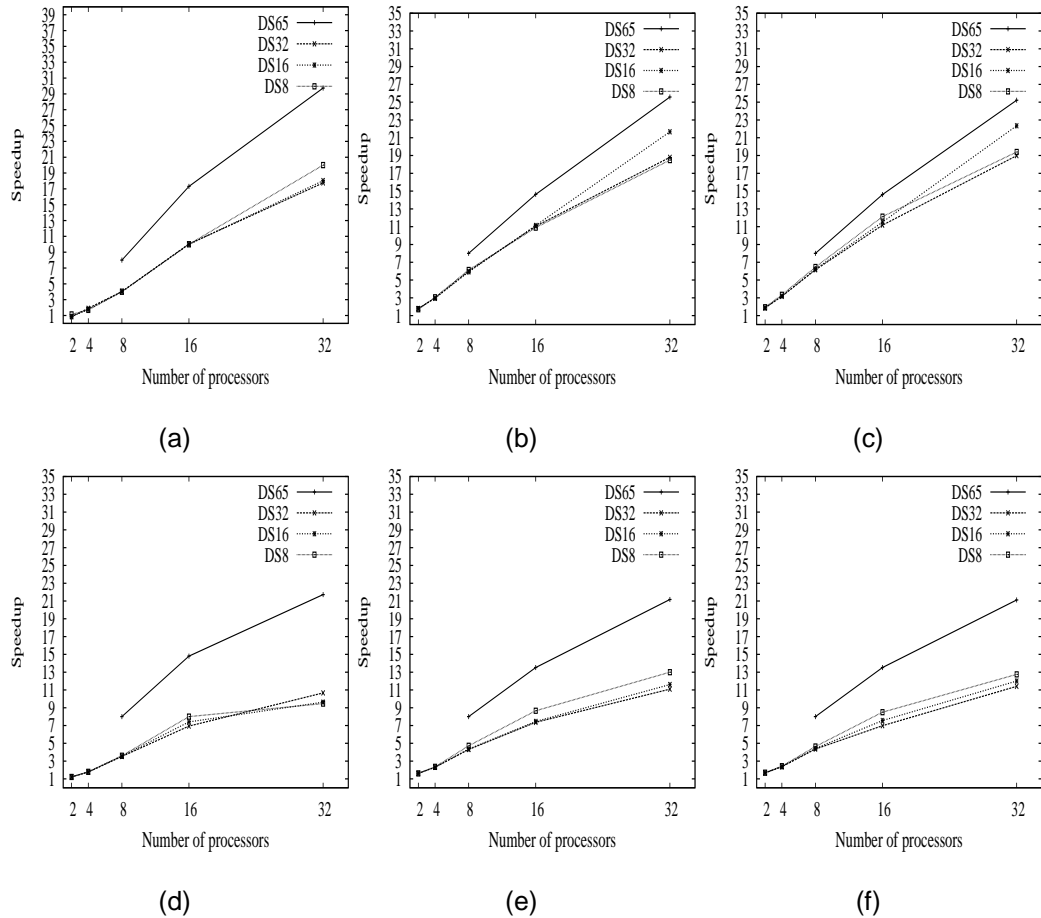


Figure 13: Speedup plots for SD1 (upper plots) and SD2 (lower plots) with varying dataset sizes (DS8, DS16, DS32 and DS65) and number of processors (1, 2, 4, 8, 16, 32; 1 Gbit Ethernet) at different scale levels; (a, d) $\rho = 1$, (b, e) $\rho = 3$ and (c, f) $\rho = 5$.

processors, but not matching exactly (see Figure 13(b)). The small deviations at that processor size are simply due to the increasing communication requirements. This effect is more apparent for the next scale level value ($\rho = 5$) at 32 processors. Due to comparatively less amount of computations at that scale level, the time spent for the communication becomes considerable.

The speedup values of SD2 are also depicted in Figure 13(d, e, f) for varying dataset sizes with different scale levels. Obtained linear scaling level behavior for the sample sparse dataset is not as good as evenly distributed data set. For all scale level values, the slope for speedup values is reduced for 32 processors regardless of the dataset size. It is expected that this shortcoming could be compensated by decomposing the dataset based on the count of foreground objects, where provides load balancing among processors coming with the burden of additional operation. As it is seen from the Figure 13, the linear behavior of the algorithm is getting improved as scale level and dataset sizes increases. Even in the present form of the algorithm, it is promising as the speedup values of SD2 approach to the values of SD1 in higher scale levels with increased dataset sizes.

Table 5: Efficiencies for SD1 (upper) and SD2 (lower) with varying dataset sizes (DS8, DS16, DS32 and DS65) and number of processors (1, 2, 4, 8, 16, 32; 1 Gbit Ethernet) at scale level, $\rho = 3$

Dataset/np	2	4	8	16	32
DS1_65	-	-	-	0.91	0.80
DS1_32	0.90	0.75	0.75	0.69	0.59
DS1_16	0.89	0.74	0.70	0.68	0.68
DS1_8	0.86	0.77	0.77	0.68	0.58
DS2_65	-	-	-	0.84	0.66
DS2_32	0.81	0.57	0.53	0.46	0.34
DS2_16	0.81	0.58	0.54	0.46	0.36
DS2_8	0.81	0.59	0.59	0.54	0.40

Table 5 shows the obtained efficiency values for SD1 and SD2 with varying

dataset sizes and number of processors at scale level, $\rho = 3$ without I/O costs. The reason of presenting efficiency results in this table only for $\rho = 3$ is that this scale level represent the behavior of the algorithm as most conveniently. Obtained efficiency values for SD1 are considerably high for $np = 2$ and $np = 16$, then exhibits a steady decrease with the increasing number of processors. This behavior is due to increasing communication requirements. Communication cost per processors increases.

The obtained results for efficiency parallel performance metric on the sparse dataset (having missing objects), SD2 are also tabulated in Table 5. When efficiency values of these two datasets (SD1 and SD2) are compared, it is observed that the efficiency values for SD2 are smaller at scale level, $\rho = 3$ and shows a fast drop after the smallest number of processors, $np = 2$ and $np = 16$. Because of the fact that processors having less objects complete their execution before others have completed. Consequently, there occurs some idling times. Due to this idling of the processor, the efficiency decreases for the sparse dataset.

On the other hand, better efficiency values have been obtained as scale level, (ρ), increases. The reason is that the execution time of wavelet transform neither depends on the cluster shape complexity nor sparsity of dataset but depends on the performance of connected component labeling algorithm. For this reason, as scale level value increases, finding the connected components on this smaller local transformed feature space is rather faster operation when compared to low scale levels. This could imply a relation between scale level and efficiency for a sparse dataset.

The parallel implementation can be further improved such that the efficiency trend might be better than the present situation. The present minor restriction is due to the fact that parent processor waits all processors before starting to create merge tables. A possible solution is broadcasting clustering border vector

of local feature space instead of implementing master-slave model and thus each processor can create its own merge table.

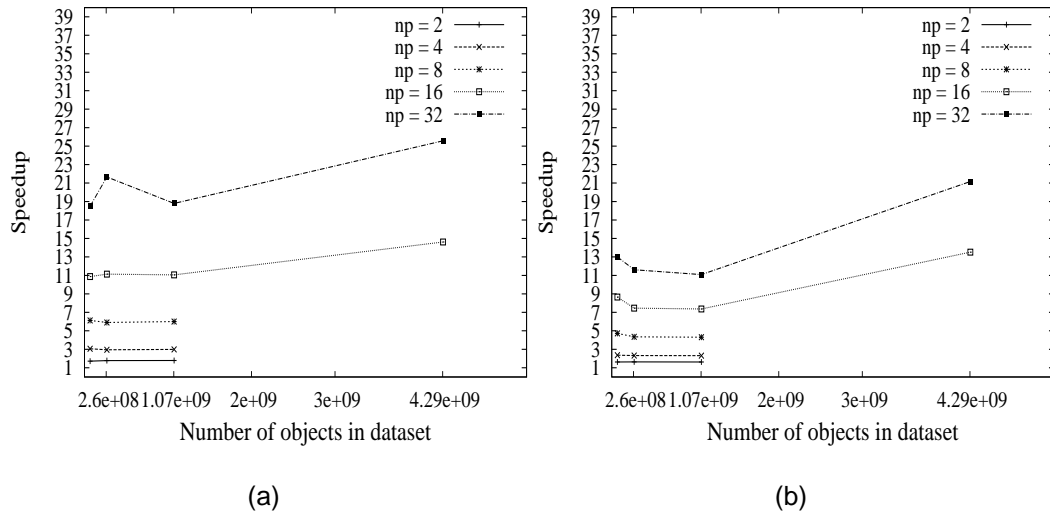


Figure 14: Speedups with varying number of objects in dataset and number of processors (1, 2, 4, 8, 16, 32; 1 Gbit Ethernet) at scale level $\rho = 3$. (a) SD1 (b) SD2

Speedup values with respect to the number of objects for SD1 and SD2 datasets are also plotted to understand the algorithmic behavior as another point of view, see Figure 14. Each symbols on the data line correspond to number of objects in datasets of DS8, DS16, DS32 and DS65, respectively. The upper limits of dataset sizes for processor sizes are apparent from the figure. Investigation of the dataset DS65 with 4294967296 objects become possible only at size of 8 processors. Figure 14 shows that speedup values are nearly independent from the number of objects in SD1 up to processor size 16. For processor size 32, there is a increase in speedup values for DS16 and DS65.

When SD2 is regarded in order to make speedup analysis as shown in Figure 14(b), the speedup values remain constant for the processor sizes of 2 and 4.

There are subtle decreases for other processors sizes up to DS32 and then increases in speedup for the largest dataset DS65. Obtained results from these experiments for the datasets SD1 and SD2 confirm that the developed parallel WaveCluster algorithm has a time complexity of $O(N)$, where N is the number of objects in the dataset. In general, the linear scaling behavior is observed up to processor size 8, but then there are small deviations from the linearity for the processor sizes of 16 and 32 mostly in positive manner. The relatively high speedup obtained at dataset DS65 for 16 and 32 processors system also supports the suitability of the algorithm for larger datasets.

CHAPTER 9

CONCLUSION

Cluster analysis has great importance for exploratory pattern analysis in many fields to group similar objects in the dataset into classes or clusters. WaveCluster approach is a novel unsupervised clustering algorithm based on wavelet transform. There are two sort of problems encountered in WaveCluster approach which studied and corresponding algorithms proposed to overcome these problems. One problem is that WaveCluster algorithm could engender performance problems when the input dataset is huge. The other problem is that it is not possible to mine huge datasets due to memory scarcity. As a solution, the parallel approach has been followed to take advantage of its simultaneous processing ability and distributed nature. In this thesis, parallel implementation of WaveCluster algorithms based on the message passing interface (MPI) for distributed memory architecture and CUDA model for shared memory architecture have been developed to study the algorithm in two distinct models.

In the MPI algorithm; obtained execution times with and without I/O times, speed-up and efficiency results have been presented for varied number of objects on a dense (evenly distributed) dataset and sparse (unevenly) dataset for different scale levels ($\rho = 1, 2, 3$) to reveal the performance of developed algorithm. Studied datasets are scaled linearly to obtain very large/huge datasets to adapt for the performance analysis aims. The highest value obtained for efficiency value (without I/O times) is 0.91 for dense dataset (DS65) containing 4294967296 objects at 1 Gbit Ethernet case.

Experiments are performed on a PC cluster of 8 compute nodes with 32 processors at total and having fast and gigabit Ethernets as underlying communication hardware. Results have shown that the parallel clustering algorithm exposes superior speed-up and linear scaling behavior (time complexity) and can be employed to overcome space complexity constraint as well due to low memory consumption.

As a parallel implementation strategy, master/slave model is adopted and replicated approach is followed to increase the performance and to reduce the amount of memory consumed at each processor for holding local datasets. This approach also brought us a considerable improvement in execution time when I/O time is considered. The communications among processors are kept as minimum to achieve high efficiency, such that each processor accesses its subset of large dataset directly in a shared manner. A minor restriction is due to the fact that parent processor waits all processors before starting to create merge tables. When this minor restriction is addressed, the presented parallel implementation can be further improved. An proposed solution is to broadcast clustering border vector of local feature space instead of implementing master-slave model and thus each processor can create its own merge table. The proposed parallel algorithm can also be applied feasibly on a shared memory architecture using thread programming technique.

In the CUDA algorithm of WaveCluster approach; the CUDA implementations of extraction of low-frequency component and connected component labeling algorithms have been presented which are essential sub-algorithms in WaveCluster algorithm. Together with these sub-algorithms, the lookup phase can be easily implemented using constant memory which is fast and cached on the device.

The reported results demonstrate that kernel algorithms expose good speedup values as dataset size increase (107.10x speedup in the kernel of low-frequency

component extraction and 5.56x in the kernel of connected component labeling). Besides, as a time complexity of the algorithms, the execution times of CCL CUDA algorithms scales nearly linear with the number of points in the used dataset. It is also observed that the data transfer time between CPU and GPU may introduce a considerable latency delay which is known as the main bottleneck on GPU computation. Spectacular increasing amount of data and high demand to process this data in a fast and efficient way makes the GPU computation as a good promised solution due to its tremendous computational power.

To sum up, promising results have been obtained in both parallel algorithms of WaveCluster approach. The results also show the usefulness of parallel computation approach to cope with algorithmic problems.

REFERENCES

- [1] **Adil, S., and Qamar, S.** (2009). Implementation of Association Rule Mining Using CUDA. In International Conference on Emerging Technologies (ICET 2009), Pages 332–336.
- [2] **Agrawal, R., Imieliński, T., and Swami, A.** (1993). Mining Association Rules Between Sets of Items in Large Databases. In SIGMOD '93: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Pages 207–216. ACM.
- [3] **Arlia, D., and Coppola, M.** (2001). Experiments in Parallel Clustering with Dbscan. In Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, Pages 326–331, London, UK. Springer-Verlag.
- [4] **Blaise Barney, L. L. N. L.** (2010). <https://computing.llnl.gov/tutorials/mpi/>.
- [5] **Boron,** (2010). <http://siber.cankaya.edu.tr/boron-ganglia/>.
- [6] **Boutsinas, B., and Gnardellis, T.** (2002). On Distributing the Clustering Process. Pattern Recogn. Lett., 23(8):999–1008.
- [7] **Brause, R., Langsdorf, T., and Hepp, M.** (1999). Neural Data Mining for Credit Card Fraud Detection. In ICTAI '99: Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence, page 103. IEEE Computer Society.
- [8] **Chapman, B., Jost, G., and Pas, R. V. D.** (2007). Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press.
- [9] **Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K.** (2008). A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. J. Parallel Distrib. Comput., 68(10):1370–1380.

- [10] **Duncan, R.** (1990). A Survey of Parallel Computer Architectures. *Computer*, 23(2):5–16.
- [11] **Ester, M., Kriegel, H.-P., Sander, J., and Xu, X.** (1996). A Density Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. of 2nd International Conference on Knowledge Discovery and Data Mining*, Pages 226–231.
- [12] **Fayyad, U., Piatetsky-shapiro, G., and Smyth, P.** (1996). From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, 17:37–54.
- [13] **Fischer, C. C., Tibbetts, K. J., Morgan, D., and Ceder, G.** (2006). Predicting Crystal Structure by Merging Data Mining with Quantum Mechanics. *Nature Materials*, 5:641–646.
- [14] **Flynn, M. J.** (1972). Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960.
- [15] **Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S.** (2004). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Pages 97–104, Budapest, Hungary.
- [16] **Gonzalez, R. C., and Woods, R. E.** (1992). *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd Edition.
- [17] **Graepel, T.** (1998). *Statistical Physics of Clustering Algorithms*. In *Diplomarbeit, Technische Universität, FB Physik, Institut für Theoretische Physik*.
- [18] **Gropp, W., Lusk, E., and Skjellum, A.** (1994). *Using MPI: Portable Parallel Programming with the Message-passing Interface*. MIT Press, Cambridge, MA, USA.
- [19] **Han, J., and Kamber, M.** (2006). *Data Mining Concept and Techniques*. Morgan Kaufman, San Francisco, USA.
- [20] **Hedberg, S. R.** (1995). Parallelism Speeds Data Mining. *IEEE Parallel Distrib. Technol.*, 3(4):3–6.
- [21] **Huffmire, T., and Sherwood, T.** (2006). Wavelet-based Phase Classification. In *PACT '06: Proceedings of the 15th International*

Conference on Parallel Architectures and Compilation Techniques, Pages 95–104, New York, NY, USA. ACM.

- [22] **Kaufman, L., and Rousseeuw, P. J.** (2005). Finding Groups in Data: An Introduction to Cluster Analysis (Wiley Series in Probability and Statistics). Wiley-Interscience.
- [23] **Kumar, N., Satoor, S., and Buck, I.** (2009). Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPU using CUDA. In 11th IEEE International Conference on High Performance Computing and Communications (HPCC 09), Pages 103–109.
- [24] **MacQueen, J. B.** (1967). Some Methods for Classification and Analysis of Multivariate Observations. In Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1, Pages 281–297. University of California Press.
- [25] **Madeira, S. C., and Oliveira, A. L.** (2004). Biclustering Algorithms for Biological Data Analysis: A Survey. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 1:24–45.
- [26] **Mukhopadhyay, A., and Maulik, U.** (2009). Unsupervised Satellite Image Segmentation by Combining SA Based Fuzzy Clustering with Support Vector Machine. Advances in Pattern Recognition, International Conference on, 0:381–384.
- [27] **Ng, R. T., and Han, J.** (2002). Clarans: A Method for Clustering Objects for Spatial Data Mining. IEEE Trans. on Knowl. and Data Eng., 14(5):1003–1016.
- [28] **Nickolls, J., Buck, I., Garland, M., and Skadron, K.** (2008). Scalable Parallel Programming with CUDA. Queue, 6:40–53.
- [29] **NVIDIA.** (2010). NVIDIA CUDA Programming Guide 3.0.
- [30] **Odewahn, S., Stockwell, E., Penning-ton, R., Humphreys, R., and Zumach, W.** (1992). Automated Star/Galaxy Discrimination with Neural Networks. Astronomical Journal, 103(1):318–331.
- [31] **Pacheco, P. S.** (1996). Parallel Programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [32] **Quammen, C.** (2005). Introduction to Programming Shared-Memory and Distributed-Memory Parallel Computers. Crossroads, 12(1):2–2.

- [33] **Shapiro, L., and Stockman, G.** (2001). Computer Vision. Prentice Hall.
- [34] **Sheikholeslami, G., Chatterjee, S., and Zhang, A.** (2000). Wavecluster: A Wavelet-based Clustering Approach for Spatial Data in Very Large Databases. The VLDB Journal, 8(3-4):289–304.
- [35] **Skillicorn, D.** (1999). Strategies for Parallel Data Mining. IEEE Concurrency, 7(4):26–35.
- [36] **Stollnitz, E. J., DeRose, T. D., and Salesin, D. H.** (1995). Wavelets for Computer Graphics: A Primer, Part 1. IEEE Comput. Graph. Appl., 15(3):76–84.
- [37] **Surdeanu, M., Turmo, J., and Ageno, A.** (2005). A Hybrid Unsupervised Approach for Document Clustering. In KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge Discovery in Data Mining, Pages 685–690, New York, NY, USA. ACM.
- [38] **TOP500**, (2010). Top 500 Supercomputing Site. <http://www.top500.org>.
- [39] **Wang, W., Yang, J., and Muntz, R. R.** (1997). Sting: A Statistical Information Grid Approach to Spatial Data Mining. In VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases, Pages 186–195, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- [40] **Yıldırım, A. A., and Özdoğan, C.** (2011). Parallel Wavecluster: A Linear Scaling Parallel Clustering Algorithm Implementation with Application to Very Large Datasets. Journal of Parallel and Distributed Computing, Elsevier.
- [41] **Yıldırım, A. A., and Özdoğan, C.** (2010). Parallel Wavelet-based Clustering Algorithm on GPUs Using CUDA. In World Conference on Information Technology (WCIT 10). Procedia-Computer Science Journal, Elsevier.
- [42] **Zamir, O., and Etzioni, O.** (1998). Web Document Clustering: A Feasibility Demonstration. In Research and Development in Information Retrieval, Pages 46–54.
- [43] **Zhang, T., Ramakrishnan, R., and Livny, M.** (1996). BIRCH: An Efficient Data Clustering Method for Very Large Databases. In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96), Pages 103–114.

- [44] **Zhang, Y., Xiong, Z., Mao, J., and Ou, L.** (2006). The Study of Parallel K-means Algorithm. In Proceedings of the 6th World Congress on Intelligent Control and Automation, Volume 2, Pages 5868–5871. IEEE Transaction on Intelligent Control and Automation.

APPENDIX A

MPI CODE OF PARALLEL WAVECLUSTER ALGORITHM

/*

The source code is only distributed to demonstrate the logic and the phases of the algorithm. If you wish to modify or distribute source code of the program, It is required to write to the author to ask for permission. The author is not responsible for damages, including any general, special, incidental or consequential damages or any data loss arising out when the full or part of the source code has been included in your software.

*/

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <glib.h>
#include <math.h>
#include <glib/gprintf.h>
#include <glib/gstring.h>
#include <glib/gslist.h>
#include <glib/gmem.h>
#include <mpi.h>
```

```
#define MAX_CLUSTER_ID 200000
#define MAX_BORDER_LUP 1000
#define LOW_R 0.7071
```



```

#define THRESHOLD 125

typedef enum _bordertype
{
    LEFT = 1, RIGHT = 2, TOP = 3, BOTTOM = 4
} bordertype;

typedef struct _border
{
    bordertype type;
    gint lefttop;
    gint rightbottom;
} border;

typedef struct _clusterborder
{
    gchar clusterid;
    GSList *borders;
} clusterborder;

typedef struct _node
{
    gint id;
    gint index_x;
    gint index_y;
    gint *borders;
    gint borderlu[MAX_BORDER_LUP][2];
    gint borderluindex;
} node;

static gint *mergetable;
static gint *borders = NULL;
static gint *connectedbufferresult = NULL;
static gint *datasetbuffer = NULL;
static GSList *nodelist = NULL;
static gint datalength;
static gint gridwidth;
static gint gridheight;
static gint basegridwidth;

```

```

static gint basegridheight;
static gint griddimensionwidth;
static gint griddimensionheight;
static gint wavelettransformationcount = 1;
static int rank = 0;
static int size = 1;

static void writepgm (gchar * path,
                    gint * buffer);
static void haartransform (gint * input,
                        gint ** lowoutput,
                        gboolean th);
static void detectclusters (gint * data);
static int
getneighbourlabel (gint * conbuffer,
                  gint index_x,
                  gint index_y,
                  gint * biggerlabel);
static gboolean hasneighbourlabel (gint *
                                data,
                                gint
                                index_x,
                                gint
                                index_y);

static void saveoutput ();
static gint *createborders ();
static node *getslavenode (int id);
static void mergeclusters ();
static void changebordervalue (gint * buf,
                              gint oldvalue,
                              gint
                              newvalue);
static void updateclusters (gint * mergedata,
                          gint
                          mergedatalength);
static void makelookup ();

int
main (int argc, char *argv[])

```

```

{
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  int i, j;

  /* initialize merge table */
  mergetable =
    (gint *) malloc (sizeof (gint) *
                     MAX_CLUSTER_ID);
  int maxcls = MAX_CLUSTER_ID;
  while (maxcls)
    {
      maxcls--;
      mergetable[maxcls] = -1;
    }

  MPI_Status status;
  gint *lowoutput = NULL;
  gboolean isparallel = TRUE;

  if (size == 1)
    {
      isparallel = FALSE;
    }

  if (rank == 0 && argc >= 2)
    wavelettransformationcount =
      g_ascii_strtod (argv[1], NULL);

  if (isparallel)
    MPI_Bcast (&wavelettransformationcount, 1,
               MPI_INT, 0, MPI_COMM_WORLD);

  /* dataset dimension size should be multiple of 2 */
  struct stat stats;
  stat ("dataset", &stats);
  datalength = sqrt (stats.st_size / 2);

```

```

if (size == 2)
{
    griddimensionwidth = 2;
    griddimensionheight = 1;
}
else if (size == 8)
{
    griddimensionwidth = 4;
    griddimensionheight = 2;
}
else if (size == 32)
{
    griddimensionwidth = 8;
    griddimensionheight = 4;
}
else
{
    griddimensionwidth = sqrt (size);
    griddimensionheight =
        griddimensionwidth;
}

```

```

basegridwidth = gridwidth =
    datalength / griddimensionwidth;
basegridheight = gridheight =
    datalength / griddimensionheight;

```

```

if (isparallel)
{
    if (rank == 0)
    {
        /* create node structures */
        int nodeindex = 0;

        for (i = 0;
            i < griddimensionheight; i++)
        {

```

```

for (j = 0;
    j < griddimensionwidth;
    j++)
{

    /* create node */
    node *slavenode =
        (node *)
        malloc (sizeof (node));
    slavenode->id = nodeindex;
    slavenode->index_x = j;
    slavenode->index_y = i;
    slavenode->borders = NULL;
    slavenode->borderluindex =
        0;

    nodelist =
        g_slist_append (nodelist,
                        slavenode);
    nodeindex++;

}

}
}

gint64 gridbuffersize =
    gridwidth * gridheight;
datasetbuffer =
    (gint *) malloc (gridbuffersize *
                    sizeof (gint));

gint gridindex_y = 0;
gint gridindex_x = 0;
/* send grid indexes to each slave node */
if (rank == 0)
{

```

```

gint *tmpgridbuf =
    (gint *) malloc (sizeof (gint) *
                    2);
for (i = 1; i < size; i++)
{
    gint tmpgridindex_y =
        i / griddimensionwidth;
    gint tmpgridindex_x =
        i -
        (tmpgridindex_y *
         griddimensionwidth);
    tmpgridbuf[0] = tmpgridindex_y;
    tmpgridbuf[1] = tmpgridindex_x;
    MPI_Send (tmpgridbuf, 2,
              MPI_INT, i, 0,
              MPI_COMM_WORLD);
}
g_free (tmpgridbuf);

}
else
{

    gint *tmpgridbuf =
        (gint *) malloc (sizeof (gint) *
                        2);
    MPI_Recv (tmpgridbuf, 2, MPI_INT,
             0, 0, MPI_COMM_WORLD,
             &status);
    gridindex_y = tmpgridbuf[0];
    gridindex_x = tmpgridbuf[1];
    g_free (tmpgridbuf);

}

/* load each buffer for node */
GIOChannel *datafile =
    g_io_channel_new_file ("dataset",

```

```

        "r", NULL);
if (datafile == NULL)
{
    g_printf
    ("error while reading file rank:%d\n",
    rank);
    return (1);
}

gint64 bufferindex = 0;
gint64 bufindex =
((gridindex_y * griddimensionwidth *
gridbuffersize) +
(gridindex_x * gridwidth)) * 2;

for (j = 0; j < gridheight; j++)
{
    g_io_channel_seek (datafile,
        bufindex,
        G_SEEK_SET);
    for (i = 0; i < gridwidth; i++)
    {

        gchar *line = NULL;
        g_io_channel_read_line
        (datafile, &line, NULL, NULL,
        NULL);
        /* read point and scale it to 255
        because bitmap has 1 bit palette! */
        datasetbuffer[bufferindex++] =
        255 *
        (gint) g_ascii_strtod (line,
        NULL);
        g_free (line);

    }

    bufindex += datalength * 2;

```

```

}

gint *lowpartbuffer = datasetbuffer;
for (i = 0;
     i < wavelettransformationcount;
     i++)
{
    gboolean th = FALSE;
    if (i ==
        wavelettransformationcount - 1)
        th = TRUE;
    haartransform (lowpartbuffer,
                  &lowoutput, th);
    if (i != 0)
        g_free (lowpartbuffer);
    lowpartbuffer = lowoutput;
}

detectclusters (lowpartbuffer);
g_free (lowpartbuffer);
borders = createborders ();

gint borderlength =
    2 * (gridwidth + gridheight);
if (rank == 0)
{
    node *slavenode = getslavenode (0);
    slavenode->borders = borders;

    gint slavecount = size - 1;
    for (i = 0; i < slavecount; i++)
    {
        gint *nodeborders =
            (gint *)
            malloc (sizeof (gint) *
                  borderlength);
        MPI_Recv (nodeborders,
                  borderlength,

```



```

        MPI_INT,
        MPI_ANY_SOURCE, 0,
        MPI_COMM_WORLD,
        &status);

    /* get slave node pointer to
       assign cluster and border list */
    slavenode =
        getslavenode (status.
                     MPI_SOURCE);
    slavenode->borders =
        nodeborders;
}

mergeclusters ();

makelookup ();

}
else
{
    /* send border data */
    MPI_Send (borders, borderlength,
              MPI_INT, 0, 0,
              MPI_COMM_WORLD);

    /* get merge result */
    gint mergedatalength = 0;
    MPI_Recv (&mergedatalength, 1,
              MPI_INT, 0, 0,
              MPI_COMM_WORLD, &status);
    if (mergedatalength != -1)
    {
        gint *mergedata =
            (gint *)
            malloc (sizeof (gint) *
                  mergedatalength);

        MPI_Recv (mergedata,

```

```

        mergedatalength,
        MPI_INT, 0, 0,
        MPI_COMM_WORLD,
        &status);
    updateclusters (mergedata,
                    mergedatalength);
}

    makelookup ();
}
}
else/* sequential execution */
{

    /* read database into memory */
    GIOChannel *datafile =
        g_io_channel_new_file ("dataset",
                               "r", NULL);
    if (datafile == NULL)
    {
        g_printf
            ("error while reading file\n");
        return (1);
    }
    gchar *line;

    /* allocate buffer to fill */
    datasetbuffer =
        (gint *) malloc (gridwidth *
                         gridheight *
                         sizeof (gint));
    gint64 bufferindex = 0;
    do
    {
        g_io_channel_read_line (datafile,
                                &line,
                                NULL, NULL,
                                NULL);
        if (line != NULL)

```

```

    {
        /* read point and scale it to
        255 because bitmap has 1 bit palette! */
        datasetbuffer[bufferindex++] =
            255 *
            (guchar)
            g_ascii_strtod (line, NULL);
        g_free (line);
    }
}
while (line != NULL);

g_io_channel_shutdown (datafile, FALSE,
                      NULL);
g_io_channel_unref (datafile);

gint *lowpartbuffer = datasetbuffer;
for (i = 0;
     i < wavelettransformationcount;
     i++)
{
    gboolean th = FALSE;
    if (i ==
        wavelettransformationcount - 1)
        th = TRUE;
    haartransform (lowpartbuffer,
                  &lowoutput, th);
    if (i != 0)
        g_free (lowpartbuffer);
    lowpartbuffer = lowoutput;
}

detectclusters (lowpartbuffer);
g_free (lowpartbuffer);

makelookup ();

saveoutput ();

```

```

    }

    if (isparallel)
    {
        saveoutput ();
        MPI_Finalize ();
        return (0);
    }

    static void
    updateclusters (gint * mergedata,
                   gint mergedatalength)
    {

        gint64 i, j;
        gint64 buflength = gridwidth * gridheight;

        for (j = 0; j < buflength; j++)
        {

            for (i = 0; i < mergedatalength / 2;
                 i++)
            {

                gint oldclusterid =
                    mergedata[i * 2];
                gint newclusterid =
                    mergedata[i * 2 + 1];

                if (connectedbufferresult[j] ==
                    oldclusterid)
                    connectedbufferresult[j] =
                        newclusterid;

            }
        }
    }
}

```

```

void
changebordervalue (gint * buf, gint oldvalue,
                  gint newvalue)
{
    gint borderlength =
        2 * (gridwidth + gridheight);
    gint i;
    for (i = 0; i < borderlength; i++)
        {
            if (buf[i] == oldvalue)
                buf[i] = newvalue;
        }
}

```

```

void
mergeclusters ()
{
    gint i, j, y, k = 0;

    /* check horizontal border neighbors */
    for (y = 0; y < griddimensionheight; y++)
        {
            for (i = 0; i < griddimensionwidth - 1;
                i++)
                {
                    node *nodeleft =
                        getslavenode (i + k);
                    node *noderight =
                        getslavenode (i + k + 1);
                    for (j = 0; j < gridheight; j++)
                        {
                            gint leftborderindex =
                                gridwidth + j;
                            gint rightborderindex =
                                (2 * gridwidth +
                                 gridheight) + j;
                            gint leftvalue =
                                nodeleft->
                                borders[leftborderindex];

```

```

gint rightvalue =
    noderight->
    borders[rightborderindex];
if (leftvalue != 0
    && rightvalue != 0
    && leftvalue != rightvalue)
{
    /* merge cluster */
    if (leftvalue < rightvalue)
    {
        changebordervalue
            (noderight->borders,
             rightvalue,
             leftvalue);
        noderight->
            borderlu[noderight->
                borderluindex]
            [0] = rightvalue;
        noderight->
            borderlu[noderight->
                borderluindex]
            [1] = leftvalue;
        noderight->
            borderluindex++;
    }
    else
    {
        changebordervalue
            (nodeleft->borders,
             leftvalue,
             rightvalue);
        nodeleft->
            borderlu[nodeleft->
                borderluindex]
            [0] = leftvalue;
        nodeleft->
            borderlu[nodeleft->
                borderluindex]
            [1] = rightvalue;
    }
}

```

```

        nodeleft->
            borderluindex++;
    }
}
}
}
k += griddimensionwidth;
}

```

```

k = 0;
for (y = 0; y < griddimensionheight - 1;
    y++)
{
    for (i = 0; i < griddimensionwidth;
        i++)
    {
        node *nodetop =
            getslavenode (i + k);
        node *nodebottom =
            getslavenode (i + k +
                griddimensionwidth);
        for (j = 0; j < gridwidth; j++)
        {
            gint topborderindex =
                gridwidth + gridheight + j;
            gint bottomborderindex = j;
            gint topvalue =
                nodetop->
                borders[topborderindex];
            gint bottomvalue =
                nodebottom->
                borders[bottomborderindex];
            if (topvalue != 0
                && bottomvalue != 0
                && topvalue != bottomvalue)
            {
                /* merge cluster */
                if (topvalue < bottomvalue)
                {

```

```

        changebordervalue
        (nodebottom->borders,
         bottomvalue,
         topvalue);
nodebottom->
borderlu[nodebottom->
borderluindex]
[0] = bottomvalue;
nodebottom->
borderlu[nodebottom->
borderluindex]
[1] = topvalue;
nodebottom->
borderluindex++;
    }
else
    {
        changebordervalue
        (nodetop->borders,
         topvalue,
         bottomvalue);
nodetop->
borderlu[nodetop->
borderluindex]
[0] = topvalue;
nodetop->
borderlu[nodetop->
borderluindex]
[1] = bottomvalue;
nodetop->
borderluindex++;
    }
}
}
}
k += griddimensionwidth;
}

```

```
node *node = NULL;
```



```

/* send merge result to slave nodes */
for (i = 1; i < size; i++)
{
    node = getslavenode (i);
    if (node->borderluindex == 0)
    {
        gint datalength = -1;
        MPI_Send (&datalength, 1, MPI_INT,
                 i, 0, MPI_COMM_WORLD);
    }
    else
    {
        gint datalength =
            node->borderluindex * 2;
        gint *mergedata =
            (gint *) malloc (sizeof (gint) *
                             datalength);
        for (j = 0;
             j < node->borderluindex; j++)
        {
            mergedata[j * 2] =
                node->borderlu[j][0];
            mergedata[j * 2 + 1] =
                node->borderlu[j][1];
        }
        MPI_Send (&datalength, 1, MPI_INT,
                 i, 0, MPI_COMM_WORLD);
        MPI_Send (mergedata, datalength,
                 MPI_INT, i, 0,
                 MPI_COMM_WORLD);
    }
}

/* update cluster numbers for master node */
node = getslavenode (0);
if (node->borderluindex != 0)
{
    gint datalength =
        node->borderluindex * 2;

```

```

gint *mergedata =
    (gint *) malloc (sizeof (gint) *
                    datalength);
for (j = 0; j < node->borderluindex;
    j++)
{
    mergedata[j * 2] =
        node->borderlu[j][0];
    mergedata[j * 2 + 1] =
        node->borderlu[j][1];
}

updateclusters (mergedata, datalength);
}
}

```

```

node *
getslavenode (int id)
{
    gint nodelength =
        g_slist_length (nodelist);
    int i;
    for (i = 0; i < nodelength; i++)
    {
        node *slavenode =
            (node *) g_slist_nth_data (nodelist,
                                       i);
        if (slavenode->id)
            return slavenode;
    }
    return NULL;
}

```

```

gint *
createborders ()
{
    gint bordersize =
        2 * (gridwidth + gridheight);
    gint *border =

```

```

(gint *) malloc (sizeof (gint) *
                 bordersize);
memset (border, 0,
        sizeof (gint) * bordersize);

/* top and bottom border */
gint i, j = 0;
for (i = 0; i < gridwidth; i++)
{
    border[j] = connectedbufferresult[i];
    border[j + gridwidth + gridheight] =
        connectedbufferresult[(gridwidth *
                                (gridheight -
                                1)) + i];
    j++;
}

/* right and left border */
j = 0;
for (i = 0; i < gridheight; i++)
{
    border[j + gridwidth] =
        connectedbufferresult[i * gridwidth +
                                (gridwidth -
                                1)];
    border[j + gridwidth + gridheight +
            gridwidth] =
        connectedbufferresult[i * gridwidth];
    j++;
}

return border;
}

void
makelookup ()
{
    gint scalex = basegridwidth / gridwidth;
    gint scaley = basegridheight / gridheight;

```

```

gint64 i, j;
for (j = 0; j < basegridheight; j++)
{
    for (i = 0; i < basegridwidth; i++)
    {
        gint pointval =
            datasetbuffer[j * basegridwidth +
                i];
        if (pointval != 255)
        {
            gint clusterval =
                connectedbufferresult[(j /
                    scaley)
                    *
                    gridwidth
                    +
                    (i /
                    scalex)];
            /* nocluster defined */
            if (clusterval == MAX_CLUSTER_ID)
                datasetbuffer[j * basegridwidth + i] = -1;
            else
                datasetbuffer[j *
                    basegridwidth +
                    i] =
                    clusterval;
        }
        else
        {
            datasetbuffer[j *
                basegridwidth +
                i] = -1;
        }
    }
}
}

void

```

```

saveoutput ()
{
    gint64 i, j;
    char buf[40];
    g_sprintf (buf, "%d_output", rank);
    GIOChannel *channel =
        g_io_channel_new_file (buf, "w", NULL);

    g_sprintf (buf, "%d %d\n", basegridwidth,
                basegridheight);
    g_io_channel_write_chars (channel, buf,
                              strlen (buf),
                              NULL, NULL);

    for (i = 0; i < basegridheight; i++)
    {
        for (j = 0; j < basegridwidth; j++)
        {
            gint value =
                datasetbuffer[i * basegridwidth +
                               j];
            if (value != -1)
            {
                g_sprintf (buf,
                            "x:%lld y:%lld c:%d\n",
                            j, i, value);
                g_io_channel_write_chars
                    (channel, buf, strlen (buf),
                     NULL, NULL);
            }
        }
    }
    g_io_channel_shutdown (channel, TRUE,
                            NULL);
    g_io_channel_unref (channel);
}

gboolean
hasneighbourlabel (gint * data, gint index_x,

```

```

        gint index_y)
{
    int i, j;
    for (i = -1; i < 2; i++)
    {
        for (j = -1; j < 2; j++)
        {
            if (i == 0 && j == 0)
                continue;
            if (index_y + i == -1
                || index_y + i == gridheight)
                continue;
            if (index_x + j == -1
                || index_x + j == gridwidth)
                continue;

            gint neighbour =
                data[((index_y +
                    i) * gridwidth) +
                    index_x + j];
            if (neighbour != 255)
                return TRUE;
        }
    }
    return FALSE;
}

gint
getneighbourlabel (gint * conbuffer,
                  gint index_x,
                  gint index_y,
                  gint * biggerlabel)
{
    gint i, j, result = MAX_CLUSTER_ID;
    if (biggerlabel)
        *biggerlabel = MAX_CLUSTER_ID;

    for (i = -1; i < 2; i++)
    {

```

```

for (j = -1; j < 2; j++)
{
    if (i == 0 && j == 0)
        continue;
    if (index_y + i == -1
        || index_y + i == gridheight)
        continue;
    if (index_x + j == -1
        || index_x + j == gridwidth)
        continue;

    gint neighbour =
        conbuffer[((index_y +
                    i) * gridwidth) +
                index_x + j];
    if (neighbour != MAX_CLUSTER_ID
        && result == MAX_CLUSTER_ID)
    {
        result = neighbour;
    }
    else if (neighbour !=
            MAX_CLUSTER_ID
            && result != neighbour)
    {
        if (neighbour < result)
        {
            if (biggerlabel)
                *biggerlabel = result;
            result = neighbour;
        }
        else
        {
            if (biggerlabel)
                *biggerlabel = neighbour;
        }
        return result;
    }
}
}

```

```

    return result;
}

void
detectclusters (gint * data)
{
    connectedbufferresult =
        (gint *) malloc (gridwidth * gridheight *
                        sizeof (gint));
    gint j = 0, i = gridwidth * gridheight;
    while (i)
    {
        i--;
        connectedbufferresult[i] =
            MAX_CLUSTER_ID;
    }
    const gint baselabel =
        MAX_CLUSTER_ID * rank + 1;

    gint labelindex = 1;
    gint biggerlabel;

    /* first pass */
    for (j = 0; j < gridheight; j++)
    {
        for (i = 0; i < gridwidth; i++)
        {
            gchar point =
                data[j * gridwidth + i];

            /* element is not background */
            if (point != 255)
            {
                gboolean hasneighbour =
                    hasneighbourlabel (data, i,
                                        j);
                if (hasneighbour == TRUE)
                {
                    gint neighbourlabel =

```



```

getneighbourlabel
  (connectedbufferresult,
   i, j,
   &biggerlabel);
if (neighbourlabel ==
    MAX_CLUSTER_ID)
{

    /* create new cluster and add
    point to this cluster */

    connectedbufferresult[j
        *
        gridwidth
        +
        i]
        = labelindex;
    labelindex++;
}
else
{

    /* assign min label to point and
    set other cluster with this label */
    connectedbufferresult[j
        *
        gridwidth
        +
        i]
        = neighbourlabel;
    if (biggerlabel !=
        MAX_CLUSTER_ID)
    {
        if (mergetable
            [biggerlabel]
            == -1)
            mergetable
            [biggerlabel] =
            neighbourlabel;
    }
}

```

```

        }
    }
}
else
{
    /* create new cluster and add
    point to this cluster */
    connectedbufferresult[j *
        gridwidth
        +
        i] =
        labelindex;
    labelindex++;
}
}
}
}

/* second pass */
gint sizeofconbuf = gridwidth * gridheight;
for (i = 0; i < sizeofconbuf; i++)
{
    if (connectedbufferresult[i] ==
        MAX_CLUSTER_ID)
        continue;

    int clsresult =
        connectedbufferresult[i];
    while (clsresult != -1)
    {
        if (mergetable[clsresult] == -1)
            break;

        connectedbufferresult[i] =
            mergetable[clsresult];
        clsresult = mergetable[clsresult];
    }
}

```

```

        connectedbufferresult[i] += baselabel;
    }
}

void
writepgm (gchar * path, gint * buffer)
{
    GIOChannel *channel =
        g_io_channel_new_file (path, "w", NULL);
    char buf[10];
    g_sprintf (buf, "%d %d\n", gridwidth,
                gridheight);
    g_io_channel_write_chars (channel, "P2\n",
                              3, NULL, NULL);
    g_io_channel_write_chars (channel,
                              "# wavelet output\n",
                              17, NULL, NULL);
    g_io_channel_write_chars (channel, buf,
                              strlen (buf),
                              NULL, NULL);
    g_io_channel_write_chars (channel, "255\n",
                              4, NULL, NULL);

    int i, j;
    for (i = 0; i < gridheight; i++)
    {
        for (j = 0; j < gridwidth; j++)
        {
            gint value =
                buffer[i * gridwidth + j];
            if (value == MAX_CLUSTER_ID)
                value = 255;
            else
                value = (value + 20) % 200;

            g_sprintf (buf, "%d\n", value);
            g_io_channel_write_chars (channel,
                                      buf,
                                      strlen
                                      (buf),

```

```

        NULL,
        NULL);
    }
    g_io_channel_write_chars (channel,
        "\n", 1,
        NULL, NULL);
}
g_io_channel_shutdown (channel, TRUE,
    NULL);
g_io_channel_unref (channel);
}

```

```

void
haartransform (gint * input,
    gint ** lowoutput,
    gboolean th)
{
    gint halfresx = gridwidth / 2;
    gint halfresy = gridheight / 2;
    gint *lowdata =
        (gint *) malloc (halfresx * halfresy *
            sizeof (gint));
    int i, j, k, t;
    gint *tmplowodd =
        (gint *) malloc (halfresx *
            sizeof (gint));
    gint *tmploweven =
        (gint *) malloc (halfresx *
            sizeof (gint));
    gint *tmplow = NULL;

    t = 0;
    for (i = 0; i < halfresy * 2; i++)
    {
        if (i % 2 == 1)
        {
            tmplow = tmplowodd;
        }
        else

```

```

    {
        tmpalow = tmpaloweven;
    }

for (j = 0; j < halfresx; j++)
{
    gint ind =
        i * 2 * halfresx + j * 2;
    gint indnext =
        i * 2 * halfresx + j * 2 + 1;
    gint sum =
        (input[ind] +
         input[indnext]) * LOW_R;
    tmpalow[j] = sum;
}
if (i % 2 == 1)
{
    for (k = 0; k < halfresx; k++)
    {
        int sum =
            (tmpaloweven[k] +
             tmpalowodd[k]) * LOW_R;
        if (th)
        {
            if (sum > THRESHOLD)
                sum = 255;
            else
                sum = 0;
        }
        lowdata[t] = sum;
        t++;
    }
}
}
g_free (tmpalowodd);
g_free (tmpaloweven);

gridwidth = halfresx;
gridheight = halfresy;

```

```
*lowoutput = lowdata;  
}
```

APPENDIX B

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Yıldırım, Ahmet Artu

Nationality: Turkish (TC)

Date and Place of Birth: 23 July 1980 , Ankara

Marital Status: Married

Phone: +90 505 5881163

email: artu@computer.org

EDUCATION

Degree	Institution	Year of Graduation
MS	Çankaya University Computer Engineering	2011
BS	Çankaya University Industrial Engineering	2003
High School	İzzettin Çalışlar High School, Uşak	1998

WORK EXPERIENCE

Year	Place	Enrollment
2006-2009	Polar Information Technologies	Lead Software Developer

2004-2006

Polar Information
Technologies

Software Developer

PUBLICATIONS

1. Ahmet Artu Yıldırım, Cem Özdoğan. Parallel WaveCluster: A Linear Scaling Parallel Clustering Algorithm Implementation with Application to Very Large Datasets, Journal Of Parallel and Distributed Computing, Elsevier, 2011.
2. Ahmet Artu Yıldırım, Cem Özdoğan. Parallel Wavelet-Based Clustering Algorithm using CUDA, Published in Procedia-Computer Science Journal, Elsevier, 2010; Presented at World Conference on Information Technology, Bahçeşehir University, 2010.
3. Ahmet Artu Yıldırım, Cem Özdoğan. Geniş Veri Kümeleri Üzerinde Paralel Öbekleme Uygulaması: Paralel WaveCluster, II. Ulusal Yüksek Başarımlı ve Grid Hesaplama Konferansı, İstanbul Technical University, 2010.
4. Ahmet Artu Yıldırım, Cem Özdoğan. Geniş Veri Kümeleri Üzerinde Paralel Veri Madenciliği Yaklaşımları: WaveCluster Yöntemi ile Öbekleme Uygulaması, 3. Mühendislik ve Teknoloji Sempozyumu, Çankaya University, 2010.