Cellular matrix for parallel k-means and local search to Euclidean grid matching

Hongjian Wang

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE TECHNOLOGIE BELFORT-MONTBÉLIARD

Cellular Matrix for Parallel *K*-means and Local Search to Euclidean Grid Matching

Hongjian WANG

UNIVERSITÉ DE TECHNOLOGIE BELFORT-MONTBÉLIARD

École Doctorale SPIM
Sciences Pour l'Ingénieur et Microtechniques

THÈSE
de l'Université de Technologie de Belfort-Montbéliard

pour obtenir le grade de

DOCTEUR

Spécialité : **Informatique**

Cellular Matrix for Parallel *K*-means and Local Search to Euclidean Grid Matching

---

Official Documentation

par
Hongjian WANG

Laboratoire Systèmes et Transports (SeT)
Institut de Recherche sur les Transports, l'Énergie et la Société (IRTES)

Soutenue le 3 décembre 2015 devant le jury composé de :

Jean-François NEZAN	**Rapporteur**, Professeur des universités, *IETR, Institut National des Sciences Appliquées de Rennes*
André ROSSI	**Rapporteur**, Professeur des universités, *LERIA, Université d'Angers*
Jean-Charles CRÉPUT	**Directeur**, Maître de Conférences HDR, *IRTES-SET, Université de Technologie de Belfort-Montbéliard*
Lhassane IDOUMGHAR	**Examinateur**, Professeur des universités, *LMIA et INRIA Grand Est, Université de Haute Alsace*
Yassine RUICHEK	**Examinateur**, Professeur des universités, *IRTES-SET, Université de Technologie de Belfort-Montbéliard*
René SCHOTT	**Examinateur**, Professeur des universités, *IECN et LORIA, Université de Lorraine*
Fan YANG	**Examinatrice**, Professeur des universités, *Le2i, Aile de l'Ingénieur, Université de Bourgogne*

Abstract

In this thesis, we propose a parallel computing model, called *cellular matrix*, to provide answers to problematic issues of parallel computation when applied to Euclidean graph matching problems. These NP-hard optimization problems involve data distributed in the plane and elastic structures represented by graphs that must match the data. They include problems known under various names, such as geometric *k*-means, elastic net, topographic mapping, and elastic image matching. The Euclidean *traveling salesman problem* (TSP), the median cycle problem, and the image matching problem are also examples that can be modeled by graph matching.

The contribution presented is divided into three parts. In the first part, we present the cellular matrix model that partitions data and defines the level of granularity of parallel computation. We present a generic loop for parallel computations, and this loop models the projection between graphs and their matching. In the second part, we apply the parallel computing model to *k*-means algorithms in the plane extended with topology. The proposed algorithms are applied to the TSP, structured mesh generation, and image segmentation following the concept of superpixel. The approach is called *superpixel adaptive segmentation map* (SPASM). In the third part, we propose a parallel local search algorithm, called *distributed local search* (DLS). The solution results from the many local operations, including local evaluation, neighborhood search, and structured move, performed on the distributed data in the plane. The algorithm is applied to Euclidean graph matching problems including *stereo matching* and *optical flow*.

Résumé

Dans cette thèse, nous proposons un modèle de calcul parallèle, appelé *matrice cellulaire*, pour apporter des réponses aux problématiques de calcul parallèle appliqué à la résolution de problèmes d'appariement de graphes euclidiens. Ces problèmes d'optimisation NP-difficiles font intervenir des données réparties dans le plan et des structures élastiques représentées par des graphes qui doivent s'apparier aux données. Ils recouvrent des problèmes connus sous des appellations diverses telles que geometric *k*-means, elastic net, topographic mapping, elastic image matching. Ils permettent de modéliser par exemple le problème du voyageur de commerce euclidien, le problème du cycle médian, ainsi que des problèmes de mise en correspondance d'images.

La contribution présentée est divisée en trois parties. Dans la première partie, nous présentons le modèle de matrice cellulaire qui partitionne les données et définit le niveau de granularité du calcul parallèle. Nous présentons une boucle générique de calcul parallèle qui modélise le principe des projections de graphes et de leur appariement. Dans la deuxième partie, nous appliquons le modèle de calcul parallèle aux algorithmes de *k*-means avec topologie dans le plan. Les algorithmes proposés sont appliqués au voyageur de commerce, à la génération de maillage structuré et à la segmentation d'image suivant le concept de superpixel. L'approche est nommée *superpixel adaptive segmentation map* (SPASM). Dans la troisième partie, nous proposons un algorithme de recherche locale parallèle, appelé *distributed local search* (DLS). La solution du problème résulte des opérations locales sur les structures et les données réparties dans le plan, incluant des évaluations, des recherches de voisinage, et des mouvements structurés. L'algorithme est appliqué à des problèmes d'appariement de graphe tels que le *stéréo-matching* et le problème de *flot optique*.

Acknowledgements

I specially thank the China Scholarship Council (CSC) for providing me financial support during my PhD study.

I would like to express my deepest gratitude to Dr. Jean-Charles Créput, my advisor, for giving me the opportunity to work on this thesis, and for providing me dedicated guidance and thoughtful advice throughout the entire process. During the past three years, he has been constantly available to discuss our results and provide insightful suggestions. This thesis would never have been possible without his elaborative direction and meticulous corrections.

I would like to extend my sincere appreciation to Professor Yassine Ruichek, for regularly discussing with me and giving me valuable advice. He is also one of the examiners of this thesis.

Special gratitude is due to Dr. Naiyu Zhang, whose thesis lays foundation for a part of this work. I am thankful to Abdelkhalek Mansouri, who has helped me conduct some of the experiments. I am grateful to Xuguang Hao, who has given me tips on programming and helped me debug my programs.

I would like to express my gratitude to Professor Jean-François Nezan and Professor André Rossi, for reviewing this thesis and posing insightful questions. I would like to extend my appreciation to Professor Lhassane Idoumghar, Professor René Schott, and Professor Fan Yang, for examining this thesis.

Contents

List of Figures

List of Tables

Chapter 1

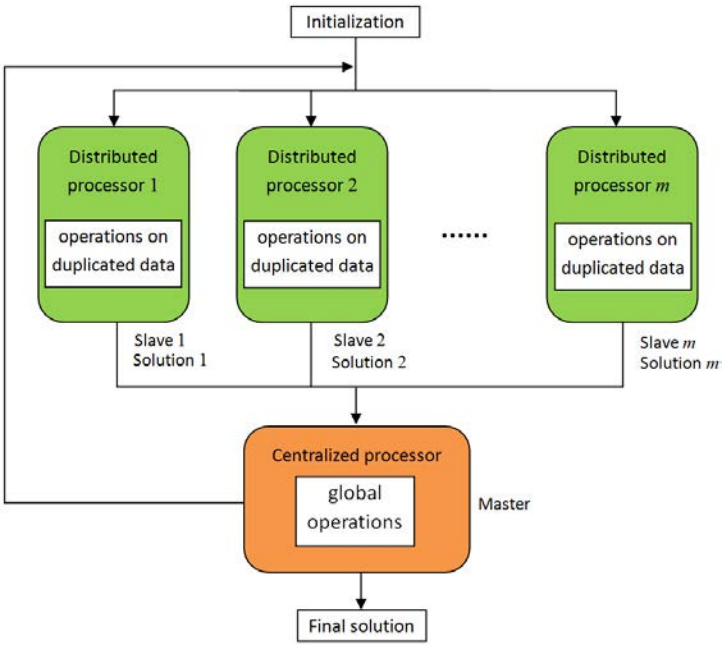
Introduction

1.1 Context

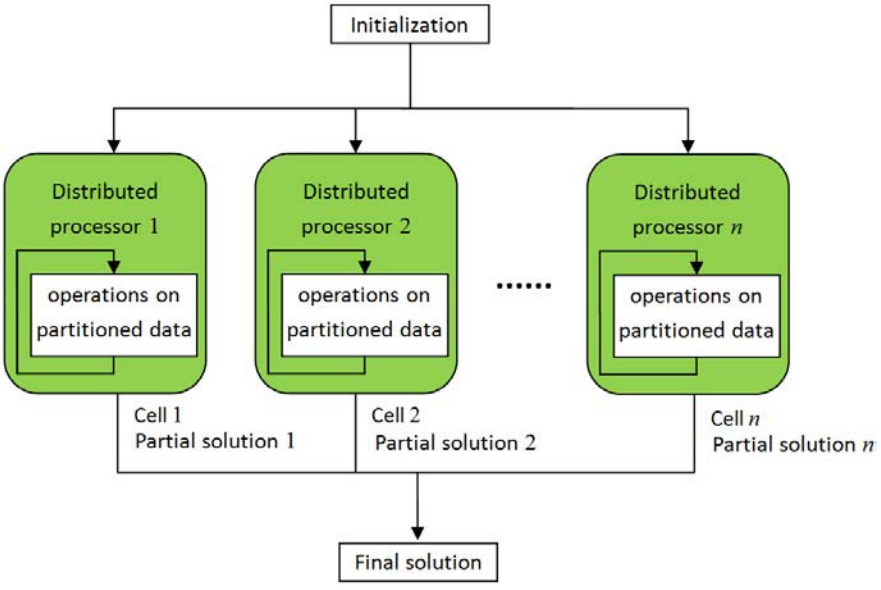
Most of the parallel implementation models for metaheuristic optimization algorithms are based on centralized control on different levels. The most common case is the so called *master-slave* model. A typical example is the well-known *genetic algorithm* (GA), where a population of candidate solutions to an optimization problem is evolved toward better solutions using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. In the master-slave parallel implementation model of GA, a *master* process manages the population of solutions, handing out solutions to a number of *slave* processes. To evolve to a new generation, the master process iteratively collects the results after the evaluation of slave processes and applies some global operations such as selection. In this case, the master process plays a central role while the slave processes act as co-processors to accelerate computation. As a result, the algorithm could not perform its function if the master process is suppressed. On the contrary, when all processes play the same role with no specific central process, the approach is called decentralized. We aim to design a conceptual model without central controller, called *cellular matrix model*, to deal with the problematic issues of decentralized control when applied to metaheuristic algorithm implementations. Two models with different control patterns are illustrated in Figure 1.1, where (a) is a master-slave model based on centralized control, and (b) is a decentralized model that we attempt to build. In the figure, the master-slave model has a centralized processor which plays the role of master process and iteratively operates during the whole algorithm. In the cellular decomposition model, there is no centralized processor, each distributed processor doing the same work during the whole algorithm. This property of decentralized control is an abstract characteristic of algorithm with the potential to build

decentralized hardware execution platform that could present robust behaviors while faced to injuries in the system. Certainly, since we are dealing with GPU (graphics processing unit) implementations, there will remain the CPU-GPU relationship, where CPU (central processing unit) is the central controller. Nevertheless, this does not preclude further exploitation of this property of decentralized control. In our work, we will try to reduce the effect of the central controller on very basic tasks as much as possible, meanwhile to augment the part of the parallel work as much as possible, following the ideal decentralized model as closely as possible.

In the literature exist attempts for designing decentralized models for parallel meta-heuristic implementations, such as *cellular genetic algorithms* [MS89, Tom99] for example, where processes are distributed on a grid, each process embedding one solution of the optimization problem. Parallel models of this type are based upon data duplication of each solution. It follows that the problem size allowed is limited by the memory size allocated to one single processor locally. Instead of data duplication, data decomposition is a way to deal with very large size problems, and our cellular matrix model is built upon data decomposition. In this case, each process embeds only a part of the input data and solution. The many processes locally interact in the plane in order to make evolve some current solution into an improved one. Given a parallel computing system with a fixed amount of memory, for example a GPU with global memory for all threads (processors), the relationship between the input problem size and the number of employed processors is different for the two computation models. Figure 1.2 gives a comparison between data duplication model and data decomposition model, in terms of the relationship between the input problem size and the number of employed processors. Here, we assume that the solution size equals the input problem size. In the data duplication model, the number of processors is decreasing along with the augment of the input problem size. This is because in such a model, the problem size allowed is limited by the memory size allocated to one single processor locally. For example, let us suppose the maximum memory is N for a given system, then, N processors can be employed at maximum if the input problem size is 1; $N/2$ processors can be employed at maximum if the input problem size is 2; and only one processor can be employed if the input problem size is N. The reciprocal relation is depicted by the data duplication curve (blue) in Figure 1.2 where $N = 10$. By contrast, in the data decomposition model, the number of employed processors is linearly increasing along with the augment of the input problem size. Let us suppose the size of the decomposed sub-problem is 1 for each processor, then, N processors are employed if the input problem size is N, as depicted by the line (red) of data decomposition in Figure 1.2. The data decomposition model has the potential to better exploit parallel computing resources when the input problem size augments. While data decomposition is not a



(a) master-slave model



(b) cellular decomposition model

FIGURE 1.1: Comparison between a master-slave model based on centralized control and data duplication, and a cellular decomposition model based on decentralized control and data decomposition.

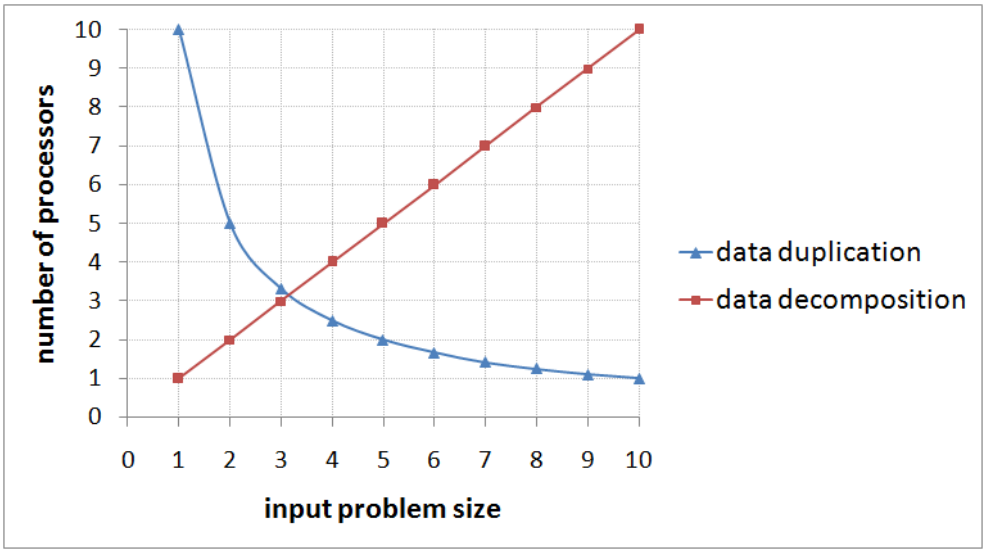


FIGURE 1.2: Comparison between data duplication model and data decomposition model. Given a parallel computing system with a fixed amount of memory, let us suppose it is 10 for example, the reciprocal relation between the input problem size and the number of employed processors, for the data duplication model, is depicted by the blue curve. Let us suppose the size of the decomposed sub-problem is 1 for each processor in the data decomposition model, then, the linearly increasing relation between the input problem size and the number of employed processors is depicted by the red line.

new way of dealing with parallelism in general, in our proposed cellular matrix model, we specifically assume a linear association from input data to processors as the problem size increases, when dealing with Euclidean optimization problems. This is the main property that we refer to as "massive parallelism", and it allows us to address large size problems. Here, by massive parallelism, we mean the theoretical and ideal possibility to execute $O(N)$ simultaneous parallel operations, where N is the input problem size.

In this thesis, we propose the cellular matrix model to deal with the problematic issues of decentralized control and data decomposition when applied to metaheuristic algorithm implementations and problems. We choose the GPU parallel computing platform to implement parallel algorithms, based on the cellular matrix model which should benefit from GPU's enormous computational power. As an essential member of the *high performance computing* (HPC) family, the GPU parallel computing is a natural choice for accelerating metaheuristic optimization algorithms which are usually time-consuming because of the hard problems, such as NP-hard problems, that they are aimed to deal with. In recent years, the performance of graphics hardware has improved rapidly and GPU vendors have made it easier for developers to harness the computation power of the GPU. Under the *compute unified device architecture* (CUDA) programming interface, a GPU works as a SIMT (*single instruction, multiple thread*) co-processor of a conventional CPU. It is based on the concept of kernels which are functions written

in C, called from CUP side, and executed by a given number of CUDA threads on GPU side. These threads will be launched onto GPU's *streaming multi-processors* (SMs) and executed in parallel [SK10].

A natural field of applications with GPU processing is image processing, which is a domain at the origin of GPU development. A lot of image processing and computer vision problems can be viewed as optimization problems in a more general way, dealing with brute data distributed in some Euclidean space and system in relation to the data. More often, these NP-hard optimization problems involve data distributed in the plane and elastic structures represented by graphs that must match the data. Such optimization problems, under various names, such as geometric *k*-means, elastic net, topographic mapping, and elastic image matching, can be stated in a generic framework of *graph matching* [Ben02, ST85, CSS07, CMC+09]. Large classes of applications are then concerned, including visual correspondence problems, and also problems in the combinatorial optimization filed such as locations of services, routing problems, and even the well-known *traveling salesman problem* (TSP), which all can be modeled as a matching process between two graphs. In this thesis, we are particularly interested in moving grids in the plane following the idea of "elastic net" matching [DW87, CHKK12], which should be applicable to a variety of matching and representational problems. The two important classes of problems that will be dealt with are *k*-means problems and elastic image matching problems. Given a set of points in some space with a distance metric, and k cluster centers, the standard *k*-means problem consists in finding the locations of cluster centers such that they minimize the sum of square distances of each data point to its closest cluster center location. We address these problems by implementation of a well-known topological map algorithm called *self-organizing map* (SOM) provided by Kohonen [Koh82]. The visual correspondence problem is to compute the pairs of pixels from two images that result from the same scene element. Its two important versions are *stereo matching* and *optical flow*, both formulated as elastic image matching problems. We address these problems by proposing a *distributed local search* (DLS) algorithm.

1.2 Objectives and Contributions

The main goal of our work can be summarized as two objectives. The first objective resides in the design of a generic parallel computation model that can partition data and define the computational level of granularity. Based on this model, a generic formulation is to be proposed, by which different Euclidean optimization problems can be formulated into the common framework of graph matching. The second objective is then related

to applying the model to different optimization algorithms in a massively parallel way, dealing with various Euclidean optimization problems under the general formulation. Because of the Euclidean nature of the problems under consideration, applications based on the model are supposed to benefit from the use of hierarchically topological data structure for data storage, search, and optimization operations, in association to parallel processors.

The contributions of this thesis are overall divided into three parts as follows.

- Firstly, we propose a computation model—cellular matrix model—that allows systematic association of the grid of processors to the massive data, such as image pixels, cities, and customers, distributed in the Euclidean plane, under different topologies of local interactions between processors. Each parallel processor works without relying on any centralized control, and it carries out simple and local operations on the topological data structure. One important property is the application of parallel *spiral search* findings and neighborhood examinations by the many processors. We assume a linear association from input data to processors and memory as the problem size increases, when dealing with Euclidean optimization problems. This property is a precondition of our work that we explicitly state in order to deal with large size problems in a massively parallel way. We provide a template parallel loop that encompasses different operations executed in the plane, allowing us to implement different Euclidean optimization algorithms in the cellular matrix model. We also propose a Euclidean grid matching problem definition that embeds both *k*-means and elastic image matching in the plane into a single and unified formulation of generic problem. This general formulation can be instantiated in different ways depending on the problems under consideration.

- Secondly, we apply the cellular matrix model to parallel implementations of *k*-means based clustering algorithms and the corresponding applications to several Euclidean grid matching problems. We provide a general energy function for topological *k*-means problems in order to highlight the generic components of the proposed parallel computation framework. We implement the parallel *self-organizing map* (SOM) algorithm which is a *k*-means based clustering algorithm with topological relationships between cluster centers. We propose a parallel image segmentation algorithm, called *superpixel adaptive segmentation map* (SPASM). It adapts the SOM for parallel execution and superpixel segmentation, using parallel on-line SOM for a fast density “projection” to deploy the cluster centers according to the distribution of some specified image attribute, before the *k*-means based segmentation by parallel batch SOM. The

goal is to produce a perceptually meaningful representation of a rigid pixel image, where the distribution (density) of superpixels coincides with the distribution of some specified attribute of the input image, such as edges, textures, and depths. Besides the SPASM, we also apply the parallel SOM algorithm to large size TSPs and the structured meshing of a disparity map.

- Thirdly, we apply the cellular matrix model to parallel implementations of local search algorithms. We propose the *distributed local search* (DLS) algorithm based on the cellular matrix model. It is a parallel formulation of a local search procedure in an attempt to follow the spirit of the standard local search metaheuristics. Starting from its location in the cellular matrix, each processor locally acts on the data. The solution results from the many local operations, including local evaluation, neighborhood search, and structured move, performed on the distributed data in the plane. Mutual exclusion is guaranteed by the cellular decomposition that allows independent moves. Classical drawbacks to address are related to cell frontier management and solution diversification. We propose the strategy of *dynamic change of cell frontier*, to eliminate conflict operations on cell frontiers. We design two classes of move operators considering small neighborhood and large neighborhood respectively, applying them in a similar way to the *variable neighborhood search* (VNS), for solution diversification. We formulate a general energy function to be equivalent to the elastic image matching problems. Example applications are visual correspondence problems including *stereo matching* and *optical flow*, which are NP-hard energy minimization problems. We apply the DLS algorithm to the two problems by minimizing corresponding energy functions.

The contributions correspond to the objectives, providing solutions to the main problems.

1.3 Outline

In Chapter 2, we provide background knowledge. We firstly give basic introductions to the two optimization algorithms that we will implement with our proposed parallel computing model, and the corresponding problems they are applied to. Then, we propose a new classification for parallel metaheuristic implementations based on three main criteria: *control*, *data*, and *memory*, and we analyze different parallel metaheuristic implementations.

In Chapter 3, we propose the cellular matrix model. We firstly present a Euclidean grid matching problem formulation which corresponds to the class of problems addressed in this thesis. Then, we detail the design of the cellular matrix model. Moreover, we provide a set of basic concepts and tools needed for the further developments of parallel computation in our proposed model.

Chapter 4 is devoted to *k*-means based algorithms and their applications. We firstly provide a general energy function for topological *k*-means problems. Then, we present the parallel SOM algorithm as a *k*-means based clustering algorithm with topological relationships between cluster centers. Afterwards, we propose the SPASM algorithm, which is a combination of parallel on-line SOM and parallel batch SOM algorithms.

Chapter 5 focuses on the experiments on our GPU implementations of the three parallel SOM applications: TSP, structured meshing, and SPASM. For each application, we perform experimental analyses and carry out comparative studies between our GPU parallel SOM approach and other approaches.

Chapter 6 is devoted to local search based algorithms. We propose the DLS algorithm and apply it to the elastic image matching problems under our generic formulation. We provide the DLS data structures and the local evaluations both on cell level and on pixel level. We propose two strategies to eliminate conflict operations on cell frontiers. We design two classes of move operators considering small neighborhood and large neighborhood respectively, and we apply different operators in the DLS algorithm, combining them under VNS framework.

Chapter 7 focuses on the experiments of the DLS algorithm applied to visual correspondence applications. First, we apply DLS to stereo matching and evaluate operators individually, with different parameter settings. We also evaluate different combinations of operators, and different initializations. Then, we turn to comparative evaluations with other energy minimization methods, performing evaluations according to the growing size of instances. Afterwards, we apply DLS to optical flow with reporting visual results and ground truth evaluations performed on standard benchmarks.

Chapter 8 concludes this thesis and provides some insights on future work.

Chapter 2

Background

2.1 Introduction

Before unveiling the main work of this thesis, we want to provide some basic concepts and definitions in this chapter, which are necessary for readers to better understand our motivations and easily catch our contributions in the latter chapters.

The main objectives of this thesis are designing a generic parallel computation model, called *cellular matrix model*, and applying the model to two classes of optimization algorithms in order to deal with various Euclidean optimization problems under a generic formulation. The first class of optimization algorithms is based on the *self-organizing map* (SOM) algorithm, which is a *k*-means based clustering algorithm with topological relationships between cluster centers. In our preliminary work, we applied SOM to the *traveling salesman problem* (TSP) [WZC13] and structured mesh generation [WZC+15]. In this thesis, we have implemented this algorithm in our parallel framework and applied it to large size instances. Therefore, in this background chapter, we firstly introduce the SOM algorithm that we employ as a heuristic to quickly generate sub-optimal TSP tour and the structured mesh of a disparity map.

The second class of Euclidean optimization algorithms we aim to implement in the cellular matrix model, is based on the local search metaheuristics. We propose the *distributed local search* (DLS) algorithm which is a parallel formulation of a local search procedure in an attempt to follow the spirit of the standard local search metaheuristics. Applications of different operators for solution diversification are possible in a similar way to the *variable neighborhood search* (VNS) algorithm. In this chapter, after the introduction of SOM, we then present a brief introduction to local search metaheuristics, including the standard local search algorithm and the VNS algorithm.

We apply the DLS algorithm to a class of Euclidean optimization problems: the visual correspondence problems. These problems including *stereo matching* and *optical flow* were formally studied by Keysers and Unger [KU03] under the appellation of *elastic image matching* problem. In the fourth section of this chapter, we provide some background knowledge about the problems. We firstly report the definition of elastic image matching, then, we present the two specific examples of visual correspondence problems. We formulate them as pixel labeling problems with energy minimization. In the next chapter, we will propose a generic formulation of grid matching problem which corresponds to the class of problems addressed in this thesis. The elastic image matching problem can then be instantiated from the formulation.

In the last part of this chapter, we focus on the investigation of different parallel metaheuristics. We propose a new classification method to parallel metaheuristic implementations, followed by a discussion of its importance. Then, we analyze different parallel metaheuristic implementations, including parallel SOM implementations, parallel local search metaheuristic implementations, and other parallel metaheuristic implementations, classifying and summarizing them with the proposed classification method. Our proposed cellular matrix model for parallel SOM and DLS should be interesting to the research field of parallel metaheuristics: decentralized control introduces potential for robustness; and data decomposition decides the linear association from input data to processors and memory needed, as the problem size increases.

2.2 Self-Organizing Map

The first class of Euclidean optimization algorithms we aim to parallelize in this thesis, is based on the *self-organizing map* (SOM) [Koh82] algorithm which we treat as a *k*-means based clustering algorithm with topological relationships between cluster centers. In this section, we briefly introduce this algorithm.

The Kohonen's SOM is a neural network approach dealing with visual patterns moving and adapting themselves to brute distributed data in space. It is often presented as a non supervised learning procedure performing a non parametric regression that reflects topological information of the input data. Meanwhile, it can also be seen as a center based clustering algorithm with topological relationships between centers. The SOM algorithm has two versions: the on-line version which consists of a stochastic regression training procedure, and the batch version which operates on all the data at the same time. We present these two versions in the following two subsections respectively.

2.2.1 On-Line SOM Algorithm

The standard SOM is a non directed graph $G = (V, E)$, called neural network, or topological grid, where each vertex $v \in V$ is a neuron, also called cluster center, having a synaptic weight vector $w_v = (x, y) \in \Re^2$. Here, $\Re^2$ is the two-dimensional Euclidean space, and the neuron network could be a one-dimensional ring or a two-dimensional regular grid deployed in the plane. Synaptic weight vector corresponds to the vertex location in the plane. The set of neurons V is provided with the d_G induced canonical metric $d_G(v, v') = 1$ if and only if $(v, v') \in E$, and with the usual Euclidean distance $d(v, v')$. The neural network is deployed on an input data set I consisting of N data points distributed in the same Euclidean space.

Algorithm 1: On-line SOM training procedure.

1: Randomly generate weight vectors;
2: **for** $iter \leftarrow 0$ to t_{max} **do**
3: Randomly extract a point p from the data set;
4: Perform competition to select the winner neuron n^* according to p;
5: Apply learning law to the neurons of a neighborhood of n^* with radius σ;
6: Slightly decrease learning rate α and radius σ of neighborhood;
7: **end for**

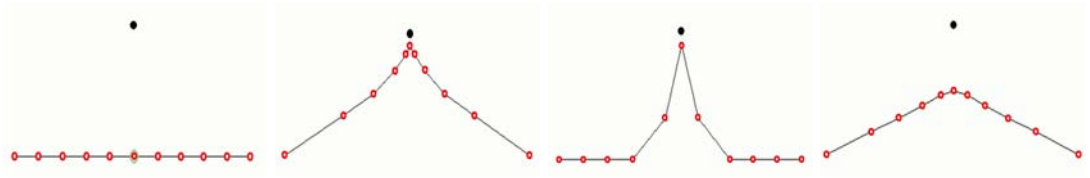
The basic incremental-learning SOM algorithm, which we call the on-line SOM, consists of a sequential training procedure given in Algorithm 1. A fixed amount of t_{max} iterations are applied to the graph (neural network), the vertex (neuron) coordinates of which being randomly initialized into an area delimiting the data set. Each iteration follows three basic steps, as indicated by lines 3—5. At each iteration t, a point $p(t) \in \Re^2$ is randomly extracted from the data set (**extraction step**). Then, a competition between neurons against the input point $p(t)$ is performed to select the winner neuron n^* (**search step**). Usually, it is the nearest neuron to $p(t)$. Finally, the learning law (**triggering step**)

$$w_n(t+1) = w_n(t) + \alpha(t) \times h_t(n^*, n) \times (p(t) - w_n(t)) \tag{2.1}$$

is applied to n^* and to the neurons within a finite neighborhood of n^* with radius σ_t, in the sense of the topological distance d_G, using learning rate $\alpha(t)$ and function profile h_t. The function profile is given by a Gaussian form

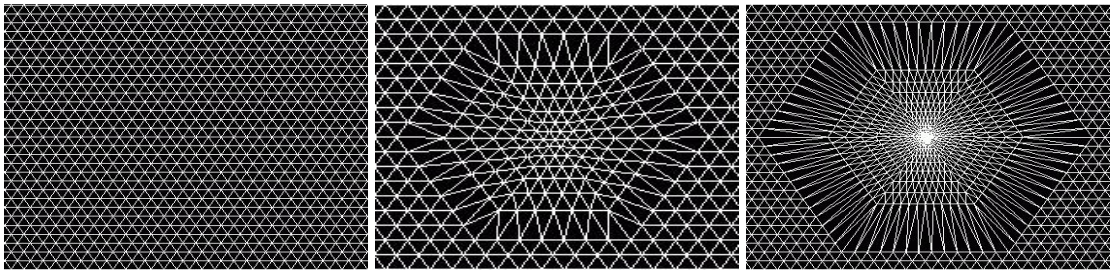
$$h_t(n^*, n) = exp(-d_G(n^*, n)^2/\sigma_t^2). \tag{2.2}$$

Here, the learning rate $\alpha(t)$ and radius σ_t are geometrically decreasing functions of time. To perform a decreasing run within t_{max} iterations, in each iteration t, the



(a) (b) (c) (d)

FIGURE 2.1: A single SOM iteration with one-dimensional neural network. The red circles represent neurons while the black dots represent input data points. (a) Initial configuration; (b) $\alpha = 0.9, \sigma = 4$; (c) $\alpha = 0.9, \sigma = 1$; (d) $\alpha = 0.5, \sigma = 4$.



(a) (b) (c)

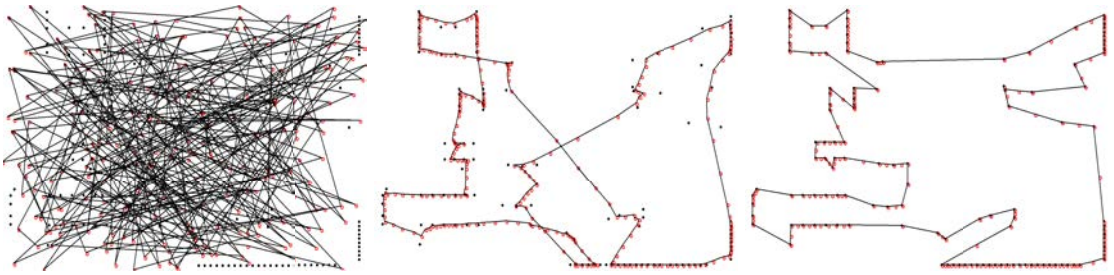
FIGURE 2.2: A single SOM iteration with two-dimensional neural network. (a) Initial configuration; (b) $\alpha = 0.5, \sigma = 6$; (c) $\alpha = 1, \sigma = 12$.

coefficients $\alpha(t)$ and σ_t are respectively multiplied by

$$exp(ln(\chi_{final}/\chi_{init})/t_{max}), \tag{2.3}$$

where $\chi = \alpha$ or $\chi = \sigma$, χ_{init} and χ_{final} being respectively the values in the starting and the final iteration. Note that a SOM simulation is characterized by the five running parameters $(\alpha_{init}, \alpha_{final}, \sigma_{init}, \sigma_{final}, t_{max})$. Examples of a basic iteration with different learning rates and neighborhood sizes are shown in Figure 2.1 and Figure 2.2. Figure 2.1 illustrates the SOM with one-dimensional neural network. Starting from the initial state of Figure 2.1 (a), results of a single SOM iteration with three different parameter settings are shown in Figure 2.1 (b), (c), and (d), respectively. Similarly, Figure 2.2 illustrates the SOM with two-dimensional neural network.

In this thesis, we employ the on-line SOM algorithm as a heuristic to quickly generate sub-optimal solutions. The applications to the Euclidean TSP and structured mesh generation have been studied in our previous work [WZC13, WZC+15], where we have used on-line SOM for the tour construction of the TSP, and for the structured mesh generation of a disparity map. In this thesis, we extend our previous work by much more experiments especially on large size instances. Now we give brief introductions to these two basic applications.



(a) (b) (c)

FIGURE 2.3: Tour construction by on-line SOM using the $pr124$ instance from TSPLIB [Rei91].

2.2.1.1 On-Line SOM to Euclidean TSP

A classical and widely studied combinatorial optimization problem is the Euclidean *traveling salesman problem* (TSP). From graph theory perspective, the TSP can be simply defined as a complete weighted graph $G = (V, E, d)$ where $V = \{1, 2, \cdots, n\}$ is a set of vertices (cities), $E = \{(i, j)|(i, j) \in V \times V\}$ is a set of edges, and d is a function assigning a weight (distance) d_{ij} to every edge (i, j). The objective is to find a minimum weight cycle in G which visits each vertex exactly once. The Euclidean TSP, or planar TSP, is the TSP where cities located in Euclidean plane and the distance between two cities is ordinary Euclidean distance. The problem is NP-complete [Pap77].

When applying on-line SOM to the TSP, we train the SOM neural network according to cities, expecting that the distribution of neurons in the Euclidean plane coincides with the distribution of cities in the same plane. Here, the input data set is the set of cities of TSP; the SOM neural network is a one-dimensional ring of neurons with the weight vector of each neuron being its correspondingly two-dimensional coordinate in the Euclidean plane. In order to generate a final admissible tour, each city has to be mapped to its nearest free neuron (the neuron that has no city assigned to it yet), two cities being assigned to different neurons. Then, the application consists of applying iterations to a ring structure with a fixed number of neurons according to the TSP size. After the on-line SOM training procedure, the ring transforms into a possible solution for TSP along which a determined tour of cities can be obtained.

To illustrate the on-line SOM behavior, an example of a TSP tour construction on the *pr124* instance from TSPLIB [Rei91] is given in Figure 2.3, showing different steps of a long simulation run. The ring network dispatches its vertices/neurons (small red circles) among cities (black dots). At the beginning starting from a scratch, which in the example shown in Figure 2.3 (a) is a ring with randomly generated vertex coordinates into the rectangular area containing cities, the local moves are performed with a considerable intensity in order to let the ring deploy toward cities. Then after 100 iterations as shown

in Figure 2.3 (b), the intensity of the moves slightly decreases in order to progressively freeze the vertices near cities (Figure 2.3 (b)—(c)). After 10000 iterations, the ring has almost completely been mapped onto cities, as shown in Figure 2.3 (c). As a final step, each city just has to be assigned or mapped to its nearest vertex in the ring in order to generate a final tour ordering.

2.2.1.2 On-Line SOM to Structured Meshing

A structured mesh in the plane is generally a grid of vertices deformed by some coordinate transformation. In this thesis, the application to structured mesh generation is presented as the solution of a clustering optimization problem in the plane. The goal is to homogeneously divide a density distribution, the disparity map in our application, between many triangles of the structured hexagonal mesh. The definition needs to consider both the hexagonal grid (the mesh) and the underlying density distribution (the disparity map). The target adaptive hexagonal mesh is illustrated in the left part of Figure 2.4. It is defined as a set of hexagonal cells, each one containing six subdivided triangles. These basic honeycomb cells are the units used to evaluate the amount of the underlying pixels they cover in the disparity map. The right part of Figure 2.4 shows such a hexagonal cell and its covering pixels from the disparity map. In this example, the total value covered, called the weight of the honeycomb cell, is the summation of the underlying pixel values ($W_k = 62$).

Let W_k be the weight of a single honeycomb cell. Note that this weight can be computed by using a standard pixel coloring algorithm. Let W be the average weight of the K honeycomb cells

$$W = \frac{\sum_k W_k}{K}. \tag{2.4}$$

We define the optimization problem as the minimization of the average percentage deviation of each individual honeycomb cell weight to the average honeycomb cell weight

$$\%\ cost = \frac{\sum_k |W_k - W|}{K \times W} \times 100. \tag{2.5}$$

Hence, the structured mesh generation problem consists in minimizing this criterion while preserving regularity of hexagonal topology. The problem is NP-hard [ZWC+13, WZC+15].

The application of on-line SOM to the structured mesh generation with stereo disparity map, consists of applying the training procedure to a regular mesh (the two-dimensional neural network), according to a density distribution represented by the disparity map. The size of the SOM neural network is lower than and in relation to the size of the

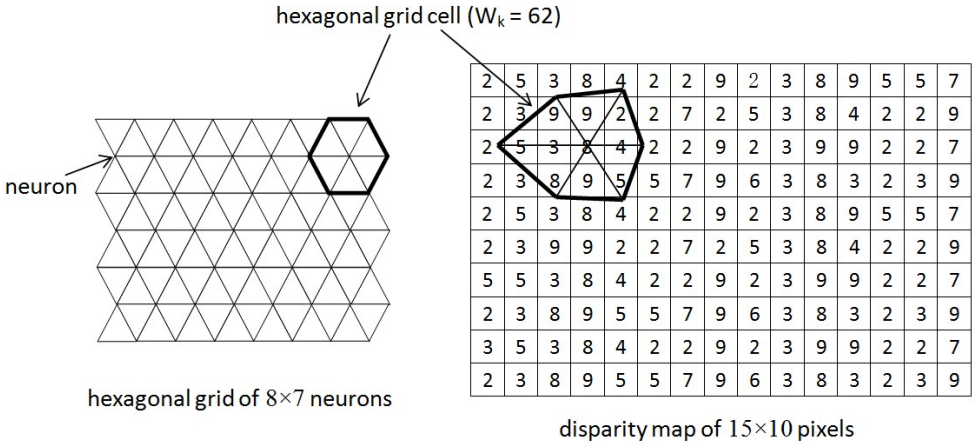
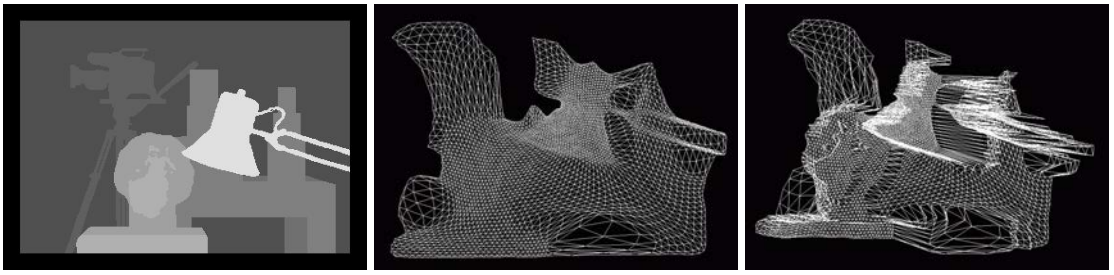


FIGURE 2.4: Hexagonal structured mesh with honeycomb cells and triangular subdivision (left part). Disparity map covered with a single honeycomb cell with weight $W_k = 62$ (right part).



(a) (b) (c)

FIGURE 2.5: A structured meshing example: (a) input disparity map, (b) meshing result viewed in 2D space, (c) meshing result viewed in 3D space.

disparity map in such a way that the mesh constitutes a compressed representation of the disparity map, with a parameterized compression rate. At the extraction step of each iteration, a pixel is randomly extracted from the disparity map according to a roulette wheel mechanism [WZC+15] depending on the density distribution (disparity values). Then, the mesh deformation must respect the density distribution and topology. This means that high disparity values, which correspond to objects close to the camera, are represented by higher densities of neural network vertices and that the structured network reflects the spatial topology or distances in 2D and 3D space. Proximity of grid vertices reflects proximity in Euclidean space. The surface reconstruction in 3D space obtained by using the adapted mesh can be seen as a compressed representation of the 3D surface, such that objects close to the camera have higher resolution and their details are more finely represented.

Figure 2.5 presents a structured meshing example. The *tsukuba* disparity map from the Middlebury stereo datasets [SS03, Mid15b] is the input as shown in Figure 2.5 (a). Figure 2.5 (b) and (c) are the meshing results viewed from 2D and 3D space

respectively. In the disparity map, brighter regions are nearer to the camera view point, such as the lamp, while in the meshing result, the adapted grid presents higher density in such regions, with respect of the topology of the scene.

2.2.2 Batch SOM Algorithm

Algorithm 2: Batch SOM training procedure.

1: Randomly generate weight vectors;
2: **for** $iter \leftarrow 0$ to t_{max} **do**
3: **for** $i \leftarrow 0$ to N **do**
4: For the i^{th} point p_i, perform competition to select the winner neuron n_j ($j = 1, 2, ..., K$) according to p_i;
5: Put p_i into the list U_j which denotes the union of all the points closest to neuron n_j;
6: **end for**
7: **for** $j \leftarrow 0$ to K **do**
8: For the j^{th} neuron n_j, compute the mean value of all the points in U_j, as the new weight vector for n_j;
9: Apply learning law to the neurons of a neighborhood of n_j with radius σ, toward the new target weight vector;
10: **end for**
11: Slightly decrease learning rate α and radius σ of neighborhood;
12: **end for**

The incremental-learning on-line SOM is a stochastic algorithm which updates the values of weight vectors sequentially iteration by iteration. Its deterministic batch equivalent, the batch SOM, uses all the input data points at each step. The batch SOM algorithm is illustrated in Algorithm 2. Instead of only one point being randomly extracted, at each iteration t of batch SOM, all points of the input data set are taken into account, each of them being associated to its closest cluster center. This procedure corresponds to lines 3—6, where N is the number of points in the input data set, and K is the number of cluster centers. The set U_j is the list of points associated to cluster center n_j. At the triggering step, as illustrated by lines 7—10, each cluster center firstly computes the mean weight of all its closest points, as the new target weight, and then the following learning law

$$w_n(t+1) = w_n(t) + \alpha(t) \times h_t(n^*, n) \times (\frac{\sum_{i=1}^{m} p_i(t)}{m} - w_n(t)) \tag{2.6}$$

is applied to each considered cluster center n^* and its neighboring cluster centers. Here, m is the number of points associated to n^*, while learning rate $\alpha(t)$ and function profile $h_t(n^*, n)$ are defined in the same way as in the on-line SOM algorithm.

2.2.3 Topological *K*-means

The *k*-means problem is a standard well-known clustering problem in density distribution estimation, data analysis, and data compression. Given a set of points in some space with a distance metric, and k cluster centers, the problem consists in finding the locations of cluster centers such that they minimize the sum of square distances of each data point to its closest cluster center location.

In this thesis, we view SOM as a *k*-means based clustering algorithm with topological relationships between cluster centers. In this case, the neurons in the SOM neural network act as k cluster centers, and the SOM learning process acts as the clustering process, during which the topological relationships between neurons are preserved through the topological gird of neural network. Especially, if we set $\alpha_{init} = \alpha_{final} = 1$ and $\sigma_{init} = \sigma_{final} = 0$, then the batch SOM algorithm turns into the standard *k*-means clustering algorithm.

2.3 Local Search Metaheuristics

The second class of Euclidean optimization algorithms we aim to parallelize in this thesis, is based on the local search metaheuristics. In this section, we firstly introduce the standard local search algorithm and then present the variable neighborhood search which is a metaheuristic algorithm using local search as basic operations.

2.3.1 Standard Local Search Algorithm

In some literature, local search is also referred as hill climbing, descent, iterative improvement, general single-solution based metaheuristics and so on. As illustrated in Algorithm 3, local search starts at a given initial solution. At each iteration, the heuristic replaces the current solution by a neighbor solution that improves the fitness function. The search stops when all candidate neighbors are worse than the current solution, meaning a local optimum is reached. For a large neighborhood, the candidate solutions may be a subset of the neighborhood. The main objective of this restricted neighborhood strategy is to speed up the search.

Variants of local search may be distinguished according to the order in which the neighboring solutions are searched. If the neighborhood is evaluated in a random manner, the algorithm is called *stochastic local search*. On the contrary, if the neighborhood is evaluated in a fully deterministic order, the algorithm is called

Algorithm 3: Template local search pseudo-code

Input: $S \in solution, operator, F_{fitness}$

```
1  begin
2      Initialize(S);
3      improvementFound ← true;
4      while improvementFound do
5          improvementFound ← false;
6          S' ← GenerateNeighbor(S, operator);
7          if F_fitness(S') is better than F_fitness(S) then
8              S ← S';
9              improvementFound ← true;
10     return S;
```

deterministic local search. With different selection strategies applied, local search algorithms can be divided into two categories:

- ***Best improvement local search*** (steepest descent). In each local search iteration, the best neighbor (i.e., neighbor that improves the most the fitness function) is selected. The exploration of the neighborhood is exhaustive, and all possible moves are tried for a solution to select the best neighboring solution. This type of exploration may be time-consuming for large neighborhoods.
- ***First improvement local search***. In each local search iteration, the first improving neighbor that is better than the current solution is selected to replace the current solution. This strategy involves a partial evaluation of the neighborhood. In the worst case (i.e., when no improvement is found), a complete evaluation of the neighborhood is performed.

The best improvement local search (BILS) is deterministic while the first improvement local search (FILS) could be either stochastic or deterministic. According to [Tal09], in practice on many applications, it has been observed that FILS leads to the same quality of solutions as BILS while using a smaller computational time.

2.3.2 Variable Neighborhood Search

Local search proceeds from an initial solution by a sequence of local changes, improving each time the value of the objective function until a local optimum is found. In order to avoid being trapped in local optima with poor values [OL96, Ree93], *variable neighborhood search* (VNS) [Mla95, MH97, BLS13] applies a strategy that consists in the exploration of dynamically changing neighborhoods for a given solution. VNS

works on a set of neighborhood operators: $operator_1$, $operator_2$, ..., $operator_{max}$. Each operator explores a corresponding neighborhood structure. Therefore, max operators define max neighborhood structures.

As illustrated in Algorithm 4, the main cycle of VNS consists of three steps: *shaking* (line 6), *local search* (line 7), and *move* (line 9). In the shaking step, $operator_n$ randomly generates a neighbor solution S' in the nth neighborhood of the current solution S. Then, S' is used as the initial solution of a local search procedure, to generate the solution S''. Note that the local search can use any neighborhood structure and is not restricted to the set of VNS. At the end of the local search process, if S'' is better than S, then S'' replaces S and the cycle starts again with $n = 1$. Otherwise, the algorithm moves to the next neighborhood $n + 1$ and a new shaking phase starts using $operator_{n+1}$.

Algorithm 4: Variable neighborhood search

Input: $S \in solution, operator_1, operator_2, \ldots, operator_{max}, F_{fitness}$

```
1  begin
2      Initialize(S);
3      while the stop condition is not satisfied do
4          n ← 1;
5          while n < max do
6              Shaking: S' ← GenerateRandomNeighbor(S, operator_n);
7              S'' ← LocalSearch(S');
8              if F_fitness(S'') is better than F_fitness(S) then
9                  S ← S'';
10                 n ← 1;
11             else
12                 n ← n + 1;
13     return S;
```

A local optimum within some neighborhood is not necessarily a local optimum within another. That is why change of neighborhoods/operators can also be performed during the local search itself. In some cases, the use of many neighborhoods in the local search is crucial [HM01]. This local search variant is called *variable neighborhood descent* (VND), as illustrated in Algorithm 5. In the step of "Exploration of neighborhood" in line 8, $operator_n$ select the best neighbor solution S' in the nth neighborhood of the current solution S. Globally, VND can be viewed as a generic framework [Tal09, BLS13] that allows us to deal with solution diversification and extends the power of simple local search.

Algorithm 5: Variable neighborhood descent

Input: $S \in solution, operator_1, operator_2, \ldots, operator_{max}, F_{fitness}$

```
1  begin
2      Initialize(S);
3      improvementFound ← true;
4      while improvementFound do
5          improvementFound ← false;
6          n ← 1;
7          while n < max do
8              Exploration of neighborhood: S' ← GenerateBestNeighbor(S, operator_n);
9              if F_fitness(S') is better than F_fitness(S) then
10                 S ← S';
11                 n ← 1;
12                 improvementFound ← true;
13             else
14                 n ← n + 1;
15     return S;
```

2.4 Elastic Image Matching

In this thesis, we propose a parallel local search algorithm, called *distributed local search* (DLS), based on the cellular matrix model. We apply the DLS algorithm to a class of Euclidean optimization problems: the visual correspondence problems. These problems including *stereo matching* and *optical flow* were formally studied by Keysers and Unger [KU03] under the appellation of *elastic image matching* problem. In this section, we firstly report the definition of elastic image matching. Then, we present the two specific examples of visual correspondence problems, after which we formulate them as pixel labeling problems and transfer them into energy minimization problems.

We now introduce the formal definition of elastic image matching. Without loss of generality, let the pair of graphs be given as regular two-dimensional grids with $W \times H$ vertices. In both girds, each vertex has a coordinate in the Euclidean plane and a value from a finite alphabet $\mathcal{V} = \{1, ..., v\}$. The grids represent images while the vertices represent pixels with gray or color values within $\mathcal{V}$. We define two distance functions. The first one is $d_v : \mathcal{V} \times \mathcal{V} \to \mathbb{N}$, and it acts on vertex values measuring the match differences in gray or color values; the second one is $d_d : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$, and it acts on displacement differences measuring the distortion introduced by the matching. Note that $\mathbb{Z}$ stands for the set of all integers. For these distance functions, we assume that they are monotonous functions that are computable in polynomial time. For example, one can use the Euclidean distance, i.e. $d_v(v_1, v_2) = f_1(\|v_1 - v_2\|)$ and $d_d(z) = f_2(\|z\|)$

with f_1, f_2 monotonously increasing. Let $\mathcal{W} = \{1, ..., W\}$, and $\mathcal{H} = \{1, ..., H\}$, the elastic image matching problem is defined as follows.

Input: The pair (A, B) of two grids of size $W \times H$.
Solution: A mapping function $f : \mathcal{W} \times \mathcal{H} \to \mathcal{W} \times \mathcal{H}$, from A to B.
Measure:

$$\begin{aligned} c(A, B, f) = & \sum_{(i,j) \in \mathcal{W} \times \mathcal{H}} d_v(A_{ij}, B_{f(i,j)}) \\ & + \lambda \cdot \sum_{(i,j) \in \{1,...,W-1\} \times \mathcal{H}} d_d(f((i,j) + (1,0)) - (f(i,j) + (1,0))) \\ & + \lambda \cdot \sum_{(i,j) \in \mathcal{W} \times \{1,...,H-1\}} d_d(f((i,j) + (0,1)) - (f(i,j) + (0,1))). \end{aligned} \tag{2.7}$$

Goal:

$$f = \underset{f \in \mathcal{W} \times \mathcal{H} \to \mathcal{W} \times \mathcal{H}}{\arg\min} \; c(A, B, f). \tag{2.8}$$

In other words, the problem is to find the mapping from A onto B that minimizes the distance between the mapped vertex values together with a weighted measure for the distortion introduced by the mapping. It is proven to be NP-complete [KU03], by means of a reduction from 3-SAT. The grid matching problem that we propose in Chapter 3 is an extension of this problem embeddings also *k*-means type problems.

2.4.1 Visual Correspondence

While the elastic image matching is as a specific case, the more general graph matching problems are numerous [HC03, KTP00, LL00, TKP01, BvdM87]. They are considered as complex problems in pattern recognition and computer vision. In the field of image processing, important versions are *stereo matching* and *optical flow* problems, which are called visual correspondence problems and studied in this thesis. The visual correspondence problem is to compute the pairs of pixels from two images that result from the same scene element. Normally in such problems, two pixels—e.g. one from the first image and the other one from the second image—that belong to the same scene element are called homologous pixels. For every pixel in the first image, if its homologous pixel is confined to the same epipolar line (with the same y coordinate) in the second image, then the problem is the stereo matching problem; if its homologous pixel can locate in any position in the second image, then the problem is the optical flow problem.

In the computer vision domain, the visual correspondence problem is often presented as an energy minimization problem [Vek99, BVZ01, BK04, Bar89]. Stereo matching and optical flow problems are generally formalized in a similar way as a pixel labeling problem [FH06, TF03, SZS+08, HRB+13, LYMD13] which assigns to every pixel in the first image a label indicating its displacement in the second image. Then, an energy function is defined based on the labeling that represents the matching solution, and the labeling which gives the lowest energy is viewed as the best matching solution. We give the definition of the pixel labeling problem and its general form of energy function in the following two subsections respectively.

2.4.2 Pixel Labeling

Many vision problems can be formulated as labeling problems. This formulation is convenient because it gives a common notation for diverse problems. In this thesis, we will use the same formal definition as in [Vek99] when necessary.

To specify a labeling problem we need a set of sites and a set of labels. Sites represent image locations on which we want to estimate some quantity, and labels represent the quantity to be estimated. Let

$$\mathcal{P} = \{1, 2, ..., n\} \tag{2.9}$$

be a set of n sites. $\mathcal{P}$ can represent pixels, edges, image segments, or other image features. Let

$$\mathcal{L} = \{l_1, l_2, ..., l_k\} \tag{2.10}$$

be a set of k labels. Labels represent intensities, disparities, or any other quantity to be estimated. For all the visual correspondence problems studied in this thesis, $\mathcal{P}$ represents image pixels and labels represent disparities for the stereo matching problem and spatial moves in the optical flow problem. For this reason we call $\mathcal{P}$ the set of pixels. Normally $\mathcal{P}$ has some natural structure in the plane. For example, pixels in an image are arranged in a two dimensional array, and each pixel which is not at the border has top, bottom, left, and right neighboring pixels. When solving a labeling problem, one frequently needs to define some relationships between sites, and measure similarity between labels, therefore a natural ordering both on the set of sites and the set of labels is helpful. In this thesis, the neighborhood structure of pixels is defined as a graph, with different regular topologies.

The labeling problem is to assign a label from the label set $\mathcal{L}$ to each site in the set of sites $\mathcal{P}$ that minimizes an energy function. Thus, a labeling is a mapping from $\mathcal{P}$ to $\mathcal{L}$.

We denote a labeling by

$$f = \{f_1, f_2, ..., f_n\}. \tag{2.11}$$

The set of all labelings $\mathcal{L}^n$ is denoted by $\mathcal{F}$.

When specialized to the visual correspondence problems, the labeling problem and the elastic image matching problem are interchangeable. Let $\mathcal{P}$ in the labeling problem represent the grid A in the elastic image matching problem, where sites (pixels) in $\mathcal{P}$ one-to-one correspond to vertices (pixels) in A. Since the labels in the labeling problem represent disparities, we can view the label of a pixel as the displacement from this pixel in A to its homologous pixel in the target image B. Then, a labeling f^l in the labeling problem is equivalent to a mapping f^m in the elastic image matching problem. For a give pixel (i, j) in A, the matched position $f^m(i, j)$ in B equals $(i, j) + f^l_{(i,j)}$, where $f^l_{(i,j)}$ represents the label of pixel (i, j) in $\mathcal{P}$. Note that the label $f^l_{(i,j)}$ is a one-dimensional disparity value in the stereo matching application, whereas a two-dimensional motion vector in the optical flow application.

2.4.3 General Form of Energy Function

The labeling problem is to assign a labeling from $\mathcal{P}$ to $\mathcal{L}$ that minimizes an energy function which expresses both the constraints of data and prior knowledge of the problem. Normally the general form of such energy function is defined as

$$E(f) = E_{data}(f) + \lambda \cdot E_{smooth}(f). \tag{2.12}$$

The first term $E_{data}(f)$, called *data energy*, encodes the constraints of the data and penalizes solutions (labelings) that are inconsistent with the observed data. The second term $E_{smooth}(f)$, called *smoothness energy*, encodes the constraints provided by prior knowledge. The design of the prior constraint is tricky, and it is frequently hard to formalize the prior constraint concisely and consistently for all problems, because its particular form depends on the particular problem in hand. A popular choice of prior which can be easily formalized expresses smoothness constraints on the labelings enforcing spatial coherence. The smoothness assumption is one of the oldest in vision [MP76, HS81, Vek99]. In this thesis, we refer to the prior energy as the smoothness energy and denote it by $E_{smooth}(f)$. The constant λ controls the relative importance of data and smoothness energy. One of the reasons why this framework is so popular is that it can be justified in terms of maximum a posteriori estimation of a *Markov random field* (MRF) [Bes86, GG84].

2.4.3.1 Data Energy

The data energy has the form

$$E_{data}(f) = \sum_{p \in \mathcal{P}} D_p(f_p). \tag{2.13}$$

Here, $D_p(f_p)$ measures how much assigning label f_p to pixel p disagrees with the data. When applied to the visual correspondence problems, the label f_p corresponds to vector (u, v) which defines the displacement in x (horizontal) and y (vertical) directions[1]. Then, $D_p(f_p)$ stands for the matching cost that expresses how well a pixel p in the first image I matches the corresponding pixel in the second image I' shifted by vector f_p.

Some very simple matching cost examples include:
1. absolute difference (AD)

$$D_p(f_p) = \left\| I_p - I'_{(p+f_p)} \right\|_1 , \tag{2.14}$$

2. Euclidean difference

$$D_p(f_p) = \left\| I_p - I'_{(p+f_p)} \right\|_2 , \tag{2.15}$$

3. squared difference (SD)

$$D_p(f_p) = \left\| I_p - I'_{(p+f_p)} \right\|_2^2 , \tag{2.16}$$

where I_p is the intensity value of pixel p in the first image I, whereas $I'_{(p+f_p)}$ is the intensity value of the corresponding pixel $(p + f_p)$ in the second image I'. Here, the intensity value of a pixel could be a single-channel gray value or a three-channel color value. In order to increase robustness, some more complex matching cost usually adds the gradient value into consideration, such as

$$D_p(f_p) = (1-\alpha) \cdot \|I_p - I'_{(p+f_p)}\| + \alpha \cdot (\|\nabla_x I_p - \nabla_x I'_{(p+f_p)}\| + \|\nabla_y I_p - \nabla_y I'_{(p+f_p)}\|), \tag{2.17}$$

where ∇_x and ∇_y are the intensity gradients in x and y direction, respectively, while α balances the intensity and gradient terms.

2.4.3.2 Smoothness Energy

To formalize the smoothness energy, we need to model how pixels interact with each other. Usually, we use the standard 4-connected neighborhood system, where

[1] For the stereo matching application, the displacement in the x direction corresponds to the disparity d $(u = d)$ and there is no shift in the y direction $(v = 0)$.

each pixel has the top, bottom, left, and right pixel as its neighbors. Then, the smoothness energy is the sum of spatially varying horizontal and vertical nearest neighbor smoothness costs, $V_{pq}(f_p, f_q)$, where if $p = (i, j)$ and $q = (s, t)$, then $|i - s| + |j - t| = 1$. If we let $\mathcal{N}$ denote the set of all such neighboring pixel pairs, the smoothness energy is

$$E_{smooth}(f) = \sum_{\{p,q\} \in \mathcal{N}} V_{p,q}(f_p, f_q) \tag{2.18}$$

where the notation $\{p, q\}$ stands for an unordered set, that is, the sum is over *unordered* pairs of neighboring pixels. $V_{p,q}(f_p, f_q)$ is a neighbor interaction function [Vek99] which gives penalties to neighboring pixels p and q if they have different labels. The form of $V_{p,q}$ determines the type of smoothness prior. Thus, the smoothness energy E_{smooth} is just the sum of neighbor interaction functions for all neighbor pairs. It assigns the labelings which are not smooth a high cost by counting all penalties between neighbor pairs having different labels. Three possible forms of $V_{p,q}$ could be:

1. everywhere smooth prior

$$V_{p,q}(f_p, f_q) = u_{p,q} \cdot \|f_p - f_q\| \tag{2.19}$$

2. piecewise constant prior

$$V_{p,q}(f_p, f_q) = u_{p,q} \cdot \delta \tag{2.20}$$

$$\delta = \begin{cases} 1 & \text{if} f_p \neq f_q, \\ 0 & \text{otherwise.} \end{cases}$$

3. piecewise smooth prior

$$V_{p,q}(f_p, f_q) = \begin{cases} u_{p,q} \cdot \|f_p - f_q\| & \text{if} \|f_p - f_q\| < C, \\ u_{p,q} \cdot C & \text{otherwise.} \end{cases} \tag{2.21}$$

where $u_{p,q}$ is a weight that varies depending on contextual information, whereas C is some constant which sets the bound on the magnitude of $V_{p,q}$. In all the three cases, $V_{p,q}$ is the product of a spatially varying per-pairing weight and a nondecreasing function of the label difference. More details about how to choose the smoothness term are discussed in [Vek99].

2.5 Classification of Parallel Metaheuristic Implementations

One of the main works of this thesis lies in designing a conceptual model for parallel and distributed implementations of different optimization algorithms. Here, in this section, we firstly give a review on parallel computing, especially focusing on the parallelization strategies applied to metaheuristics. Then, we propose a taxonomy for parallel implementation models of metaheuristics, with which we analyze existing models of the two classes of algorithms that we deal with in the thesis: SOM and local search. Models of other metaheuristic algorithms such as *genetic algorithm* and *ant colony optimization* are also discussed based on the proposed taxonomy, after which a summary classification for models of different metaheuristics is given.

Generally, parallel computing speeds up computation by dividing the work load among a certain amount of processors. In the parallel computing community, two main sources of parallelism which are well accepted are *data parallelism* and *task parallelism*.

- ***Data Parallelism*** refers to the execution of the same operation or instruction on multiple large data subsets at the same time [FL00]. In a multiprocessor system executing a single set of instructions, which is called *single-instruction stream-multiple-data stream* (SIMD) according the Flynn's taxonomy [Fly72], data parallelism is achieved when each processor performs the same operation on different pieces of distributed data. In these scenarios, the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. The source collection is usually partitioned so that multiple processors can operate on different segments concurrently.
- ***Task Parallelism***, also called *control parallelism*, or *function parallelism*, or *operation parallelism*, refers to the concurrent execution of different tasks allocated to different processors, possibly working on the "same" data and exchanging information [CT10]. Task parallelism focuses on distributing tasks, which represent asynchronous operations, across different parallel processors so that independent tasks run concurrently. In the general case of task parallelism, each processor executes a different process on the same or different data, and different execution processors communicate with one another as they work.

Parallel computations based on these two parallelisms are particularly efficient when algorithms manipulate data structures that are strongly regular, such as matrices in matrix multiplications. Algorithms operating on irregular data structures or on data with strong dependencies among the different operations remain difficult to parallelize

efficiently and to characterize comprehensively, using only data parallelism or task parallelism. Metaheuristics generally belong to this category, and parallelizing them offers opportunities to find new ways to use parallel and distributed computational systems and to design parallel algorithms [CT03].

Crainic and Toulouse [CT03, CT10] have proposed a classification specific to the parallelization strategies applied to metaheuristics. In their classification, three dimensions are considered which respectively indicate (1) how the global problem-solving problem is controlled, (2) how information is exchanged among processes, and (3) the variety of methods involved in the search for solutions. Specific classifications applied to some particular metaheuristic algorithms are also found in the literature. For example, Luong et al. [VLMT13] have studied various algorithmic issues to design efficient parallel local search metaheuristics, and they have summarized three major parallel models for local search metaheuristics: *solution-level parallel model*, *iteration-level parallel model*, and *algorithmic-level parallel model*. Classifications for parallel strategies of other specific metaheuristic algorithms also exist, such as the works by Tomassini [Tom99, AT02] and Alba [AT02] for parallel and distributed *evolutionary algorithms* (EAs), and the work by Pedemonte et al. [PNC11] for parallel *ant colony optimization* (ACO) algorithms. However, we did not find a classification that clearly distinguishes the possibility for "massive parallelism" for optimization. We think that this property relies to adequate association of processors to the data, in such a way to define precisely the relation to the problem size. This can be achieved by distinguishing between data decomposition parallelism and data duplication parallelism, as in our proposed classification in the following subsection.

2.5.1 A Taxonomy Based on Three Factors

In our opinion, the traditional dual data/task classification for general parallel computing looks not sufficiently precise when dealing with the various parallel metaheuristics specifically dedicated to the implementation level. One important point that should be emphasized concerns the allocation of processors and memory according to the problem size. We think that this point specific to optimization should be alighted in the taxonomies of parallel and distributed metaheuristic implementations, since it determines the maximum size of the input that should be solved with the system and how the performance should grow according to the amount of physical cores and memory.

Taking the above considerations into account, we propose a new classification criterion as shown in Figure 2.6. We now have three criteria. They are based on the three

factors that every parallel implementation model of metaheuristic optimization needs to consider: *control*, *data*, and *memory*. Note that though two terms of our taxonomy are literally similar to the traditional parallelism classification, they stand for different considerations.

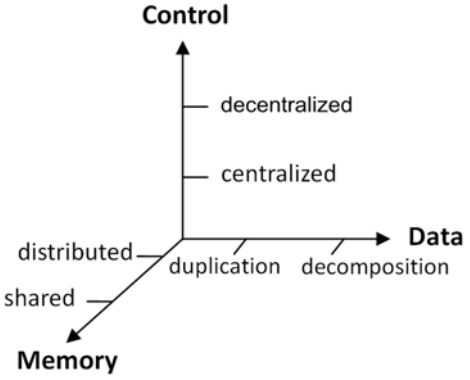


FIGURE 2.6: A taxonomy based on *control*, *data*, and *memory*, for parallel and distributed implementation models.

— ***Control.*** This term is about algorithmic organization and its corresponding execution pattern on parallel processors. Some parallel implementation models are based on centralized control on different levels, such as the classic *master-slave* model as shown in Figure 1.1 (a), where the master process plays a central role. In our classification, we call this kind of implementation model "control centralized". The opposite implementation model should be in a completely distributed control pattern, without depending on any central control, as the cellular model shown in Figure 1.1 (b). Thus, the robustness can be guaranteed because the computation can continue even when some computing units fall down. We call this kind of implementation model "control decentralized".

— ***Data.*** This term denotes the input data with size N of the problem to be solved, and the representation of the solution. The size of the solution could generally be $O(N)$ since it is in relation to the input problem size. However, the size might depend on optimizing operations and the implementation choices of designers. Some algorithms perform metaheuristic exploration and exploitation within a set of solutions (population), handling each solution in parallel, and then selecting the best-so-far solution iteratively. Implementation models of this kind are built upon "data duplication". Alternatively, other algorithms generate every part of the whole solution separately in parallel. The final solution can then be obtained by combining together partial results from all the processors. Hence, implementation models of this kind are founded on "data decomposition". A linear relationship of processors to the problem size makes these models able to handle larger scale problems with limited physical memory, than the models which follow "data duplication" pattern.

— ***Memory.*** This term concerns concrete implementations of memory access in parallel computing platforms. Two commonly used categories are “global memory” (also called “shared memory” in parallel computing community) and “distributed memory”, and we adopt them in our classification. Normally, if the considered algorithm is implemented in global memory systems, such as GPU platforms, then it usually has to deal with the problem of memory access contention, especially if global memory access is through a single path such as a bus. Cache memory alleviates the problem but it does not solve it. On the other hand, distributed memory means that each processors only manages its local memory and the communication is by message passing as in the peer-to-peer communication scheme. The information exchange among different processors is via message passing mechanism. As a result, the communication bottleneck of distributed memory computing systems usually becomes the main obstacle to high performance of the “distributed memory” implementation models. Note that GPU implementations are in the “global memory” category, even if the process behaviors only concern local memory accesses.

It should be clarified that our taxonomy is not only for the parallelization strategies of metaheuristic algorithms, but also for their concrete implementations which are related to the employed platforms, owing to the *memory* term. By classifying the concrete implementation models of metaheuristic algorithms according to our taxonomy, the employment of processors and memory according to the problem size can be predicted, and the possible performance bottlenecks could also be forecast. For example, most of the implementations under master-slave model are based on “control centralized, data duplication”, as shown in Figure 1.1 (a). Given a parallel computing system with a fixed amount of memory, the reciprocal relation between the input problem size and the number of employed processors could be predicted as shown in Figure 1.2, if the implementation is in “data duplication” model. By contrast, the linearly increasing relation between the input problem size and the number of employed processors is implied if the implementation is in “data decomposition” model. If an implementation is done in global memory computing systems, for example on GPU computing platforms, then a lot of attention should be paid on the global memory access efficiency and contention. Implementations under coarse-grained models, where the ratio of computation to communication is high, are more adapted to distributed memory computing systems, such as clusters. This is the case of *cellular genetic algorithm* [Tom99] implementation model that is based on “distributed memory, data duplication, control decentralized”.

2.5.2 Parallel Self-Organizing Map Implementations

The first class of algorithms we aim to parallelize in this thesis, is based on the SOM algorithm which we view as a *k*-means based heuristic for different Euclidean optimization problems. We continue a collective work started during the PhD work of Naiyu Zhang [Zha13]. Between the neural network and the input data, we add a uniform two-dimensional cellular matrix with linear relationship to input size, as a level of decomposition of the plane and the input data. Its role is to memorize the neurons in a distributed fashion and authorize many parallel closest point searches in the plane by a spiral search algorithm [BWY80, CK09], and then many training procedures in parallel. Our contribution is to propose a unified view of such methods in the context of grid matching, and produce specific GPU implementations that are effective for large size problem instances, into a generic GPU kernel framework in different topologies. More details of our proposed cellular matrix model are provided in Chapter 3.

In the literature, other methods for computing SOM on GPU have been proposed [MSH+12, YKN+12]. These methods accelerate SOM process by parallelizing the inner steps at each basic iteration, firstly, to find out the winner neuron in parallel, secondly, to move the winner neuron and its neighbors in parallel. Consequently, these kinds of implementation models fall into the "control centralized" category. By contrast, our cellular matrix approach should be attributed to the "control decentralized, data decomposition, global memory" category, in that, firstly, decentralized controlling guarantees the model's robustness, secondly, data decomposition eases the burden of massive memory usage when dealing with large-scale problems, and thirdly, global memory reduces the communication costs among different processing units and allows easy implementation on GPU like systems. Because of the very low level of granularity, applications on distributed memory systems look actually unrealistic.

2.5.3 Parallel Local Search Implementations

The second class of Euclidean optimization algorithms we aim to parallelize by the cellular matrix model, is based on the local search metaheuristics. In the literature, several parallel strategies for local search implementations can be found. Generally, local search is a metaheuristic which could be viewed as "walks through neighborhoods". The walks are performed by iterative procedures that allow moving from one solution to another, through the solution domains of the problems in hand. Parallelism naturally arises when dealing with a neighborhood, since each of the solutions belonging to it is an independent unit. This kind of parallelization, called *iteration-level parallel model*, is a low level master-slave model in which evaluation of the

neighborhood is made in parallel [Tal09, VLMT13]. At the beginning of each iteration, the master duplicates the current solution among parallel computing nodes. Each of them manages a number of candidates, and the results are returned to the master. This implementation model obviously follows the "control centralized, data duplication" pattern. In [VLMT13], Luong et al. have redesigned the above model on GPU platform. Considering a neighbor as a slight variation of the candidate solution which generates the neighborhood, they only copy the representation of this candidate solution from CPU to GPU. Then, N^2 threads are employed to carry out the parallel *2-opt* moves and evaluations, where N is the TSP instance size. Each parallel evaluation only deals with the slight variation based on the candidate solution, with the help of a neighborhood mapping which locates each thread's corresponding variation position in the solution representation. The fitness results generated by parallel threads need to be gathered and selected for a best one, which will become the new starting solution, called *pivot*, at the next local search iteration. The solution representation and the fitness structure are stored in the global memory of GPU. From the above, it can be concluded that this strategy follows the "control centralized, data decomposition, global memory" pattern.

Other two major parallel models for local search can be distinguished as *solution-level* and *algorithmic-level* [Tal09, VLMT13]. In the solution-level parallel model, the focus is on the parallel evaluation of a single solution, and the function can be viewed as an aggregation of partial functions. Implementations based on this model follow the "control centralized, data decomposition" pattern. In the algorithmic-level parallel model, several local search metaheuristics are simultaneously launched for computing robust solutions. The well-known multistart local search (MLS), in which different local search algorithms are launched using diverse initial solutions, is an instantiation of this model [Tal09]. Implementations based on this model follow the "control centralized, data duplication" pattern. In our opinion, centralized selection procedures among parallel processors are inevitable, as long as each processor deals with a whole solution.

We think that an interesting implementation model should be fully distributed, where each processor carries out its own local search based on part of the input data, considering only a local part of the whole solution. Operations on different processors should be similar, with no centralized selection procedure, except for final evaluation. A final solution should be obtained with the partial operations from different processors. Therefore, this implementation model of local search should follow the "control decentralized, data decomposition" pattern, as shown in Figure 1.1 (b). Then, in its "distributed memory" form, it should be able to solve very large challenging problems, such as the *World TSP Challenge*, in distributed computing systems with message passing connection. The *World TSP* has already been solved by Nguyen et al. [NYYY07] using master-slave *genetic algorithm* (GA) and data decomposition. They have applied an

effective implementation of hybrid GA incorporating *Lin-Kernighan* (LK) heuristic, to the 1,904,711-city World TSP Challenge. They divided the instance into a number of smaller sub-instances and then applied GA-LK to these sub-instances. Finally, they reconnected all the best segments of each sub-instance to form a new best tour for the World TSP Challenge instance. This example, however, has a high level of granularity since each processor deals with a significant part of the input data using GA-LK. As a result, it was implemented in distributed memory systems. Another interesting implementation model for local search metaheuristics is reported by Sánchez-Oro et al. in [SOSR+15], where parallel local search processes are conducted under the VNS framework, following a divide and conquer strategy. In this implementation, the solution for the dynamic memory allocation problem is equally partitioned among the available processors, and the partial local search behaviors performed by different processors improve different parts of the solution independently of each other, without relying on any central selection. Hence, this implementation model can be classified into "control decentralized, data decomposition" category. In this thesis, our proposed cellular matrix model for local search implementations also follows the same spirit of "control decentralized, data decomposition" as the implementation model of Sánchez-Oro et al. [SOSR+15]. The difference between our work and their approach is that we systematically consider problem size, whereas such analysis was not developed in their paper. We implement our model in GPU parallel computing systems, which are "global memory" systems with GPU global memory.

2.5.4 Parallel Implementations of Other Metaheuristics

With the proposed taxonomy based on three factors, we can also classify and analyze parallel implementations of other metaheuristics, such as GA and ACO. There are several possible levels at which GAs can be parallelized: the fitness evaluation level, the individual level or the population level [Tom99]. Parallelization at the fitness evaluation level is usually implemented under master-slave model, in which each individual fitness is evaluated simultaneously on a different processor. This architecture belongs to the "control centralized, data duplication" category, and it can be implemented on both shared memory multiprocessors as well as distributed memory machines, ie. network of workstations. Individual or population-based parallel approaches for GAs introduce additional terms that should be considered, such as *deme*, *migration* and *topology* [Kon04]. These approaches are inspired by the observation that natural population tends to possess a spatial structure. The two important spatial structure based categories are *island model* and *cellular model*. The island model [CHMR87] features geographically separated subpopulations of relatively

large size. Subpopulations may exchange information from time to time by allowing some individuals to migrate from one subpopulation to another according to various patterns. In the cellular model [MS89], individuals are placed on a large toroidal one or two-dimensional grid, one individual per grid location. Fitness evaluation is done simultaneously for all individuals, and selection, reproduction and mating take place locally within a small neighborhood. From an implementation point of view, these two kinds of models are often adapted to distributed memory systems [AK96, FPS03] and accordingly they are classified into the "control decentralized, data duplication, distributed memory" category.

As early as when Dorigo [Dor92] initially proposed the ant colony optimization (ACO), he suggested the application of parallel computing techniques to enhance both the ACO search and its computational efficiency. A comprehensive survey on parallel ACO can be found in [PNC11]. Among various parallel ACO implementations, the master-slave model has been quite popular in the research community, mainly due to the fact that this model is conceptually simple and easy to implement. According to Pedemonte et al. [PNC11], the master-slave model is further divided into three distinguished subcategories regarding the "granularity". The standard implementation of *coarse-grain master-slave* ACO assigns one ant to a slave that is executed on an available processor. The master globally manages the global information (i.e. the pheromone matrix, the best-so-far solution, etc.), and each slave builds and evaluates a single solution. The communication between the master and slaves usually follows a synchronous model. This kind of implementation model follows the "control centralized, data duplication" pattern. In the *medium-grain master-slave* model, a domain decomposition of the problem is applied. The slaves solve each subproblem independently, whereas the master manages the overall problem information and constructs a complete solution from the partial solutions reported by the slaves. Furthermore, in the *fine-grain master-slave*, the slaves perform minimum granularity tasks, such as processing single components used to construct solutions, or parallel evaluation of solution elements. These two kinds of implementation models follow the "control centralized, data decomposition" pattern and they can be implemented both in shared memory systems and in network of workstations or clusters, with each computing node having independent memory. Frequent communications between the master and slaves are usually required in these models, and this issue is more severe when they are implemented in distributed memory systems than shared memory systems.

There exist other parallel and distributed ACO implementation models that follow the "control decentralized" pattern. In the *cellular* model [PNC11, PC10], a single colony is structured in small neighborhoods, each one with its own pheromone

matrix. Each ant is placed in a cell of a toroidal grid, and the trail pheromone update in each matrix considers only the solutions constructed by the ants in its neighborhood. In the *multicolony* model [PNC11, RL02], several colonies explore the search space using their own pheromone matrices. The cooperation is achieved by periodically exchanging information among the colonies. In the *parallel independent runs* model [PNC11, Stü98, BOL+09], several sequential ACOs, using identical or different parameters, are concurrently executed on a set of processors. The executions are completely independent, without communication among the ACOs, therefore the model does not consider cooperation between colonies. The latter two models have distributed controlling at colony level. These three models above all follow the "data duplication" pattern and they can be implemented in both shared memory [BOL+09] and distributed memory [PC10] systems.

2.5.5 Discussion on Parallel Implementation Models

In Table 2.1 is reported a summary that lists the above mentioned metaheuristic implementations according to our classification criteria. In the literature, many metaheuristic models are labeled as "distributed" models because of the conceptual underlying model over the implementation. These implementations most often belong to the "control centralized, distributed memory" category while others belong to the "control decentralized, global memory" category, or many are simply sequential simulations of a conceptually parallel model. In our opinion, these implementation models could be called *partly distributed*, or distributed in a weak sense. For example, even if the master-slave model is implemented in distributed memory systems with computing nodes communicating by message transfers, the master process necessarily deals with specific data structures different from the slave data structures. We think the implementation model based on "control decentralized, distributed memory" could be called *fully distributed*, or distributed in a strong sense. In this case, no component has special role and it could be carried out on networks of workstations or processors, communicating by message transfers with strong robustness. This is the case for cellular GA.

From our point of view, a very significant conceptual implementation model should follow the "control decentralized, data decomposition" pattern, because of the ability to solve very large size problems in computing networks. Decentralized control guarantees the robustness of the model, while data decomposition decides the linear association from input data to processors and memory needed, as the problem size increases, when dealing with Euclidean optimization problems. Because of the low level of granularity, designing algorithms that belong to the "control decentralized,

TABLE 2.1: Classification for implementation models of different metaheuristics according to our taxonomy with three factors.

	control	*data*	*memory*
master-slave GAs	centralized	duplication	distributed
island GAs	decentralized	duplication	distributed
cellular GAs	decentralized	duplication	distributed
coarse-grain master-slave ACO	centralized	duplication	distributed
medium-grain master-slave ACO	centralized	decomposition	distributed
fine-grain master-slave ACO	centralized	decomposition	distributed
cellular ACO	decentralized	duplication	distributed
multicolony ACO	decentralized[1]	duplication	distributed
parallel independent runs ACO	decentralized[1]	duplication	distributed
iteration-level parallel local search	centralized	duplication	distributed
GPU local search [VLMT13]	centralized	decomposition	global[2]
solution-level parallel local search	centralized	decomposition	distributed
algorithmic-level parallel local search	centralized	duplication	distributed
hybrid GA with LK heuristic [NYYY07]	centralized	decomposition	distributed
parallel VNS [SOSR+15]	decentralized	decomposition	global[3]
our cellular matrix model	decentralized	decomposition	global[2]

[1]colony level. [2]GPU platform. [3]CPU platform.

data decomposition, global memory" category should be a reasonable starting point exploiting the GPU platform with global memory. This is what we do through the proposed cellular matrix model in this thesis. On the other hand, executing low-level granularity algorithms based on data decomposition in distributed memory systems remains an important challenge.

2.6 Conclusion

We have reviewed different parallel metaheuristic optimization implementations, according to our proposed classification method with three criteria: *control*, *data*, and *memory*. Parallel implementation models based on data duplication are numerous, and they are more adapted to standard peer-to-peer multi-processors or networks of workstations, since the level of granularity allows communications by message passing. Even though some parallel implementation models based on data decomposition exist, most of them are in the master-slave model with a central controller that plays an important role in the algorithm behavior. Some parallel implementation models based on "control decentralized" also exist, but they are often built upon data duplication of each solution. In this thesis, we aim to design a conceptual model in the "control decentralized, data decomposition" pattern. The model, called cellular matrix

model, partitions data distributed in the plane; defines a given level of computation granularity; and allows generic and systematic association of processors to the massive data deployed in the plane, under different topologies of local interactions between processors. Each parallel processor carries out simple and local operations on the topological data structure.

We have introduced two classes of Euclidean optimization algorithms, SOM and local search metaheuristics, for which we develop parallel implementations. They are the target algorithms that we will implement in a massively parallel way based on the cellular matrix model. In the literature, approaches with data decomposition already exist, for example the parallel VNS implementation reported by Sánchez-Oro et al. [SOSR+15]. The difference between our work and the existing approaches is that we systematically consider problem size, whereas such analysis was not developed in other papers. According to data decomposition paradigm, GPU looks like a good platform suited to the low level of granularity and the many local interactions that take place, because all processors (threads) have direct accesses to global memory with high throughput. Thus, we try to implement models of "control decentralized, data decomposition, global memory" type in GPU parallel computing systems, even though mainly with local memory accesses.

We have also presented the background work on visual correspondence problems under the appellation of elastic image matching problems. We have reported the definition of elastic image matching; we have given the two specific examples of stereo matching and optical flow; we have formulated the problems as pixel labeling problems and transfered them into energy minimization problems. It will be the focus of the following chapters to propose a unified framework to describe both *k*-means problems and elastic image matching problems, and derive common parallel processing solutions.

Chapter 3

Cellular Matrix Model

3.1 Introduction

This chapter is dedicated to the first contribution of this thesis, the *cellular matrix model* that allows systematic association of the grid of processors to the massive data distributed in the plane, under different topologies of local interactions between processors. Each parallel processor carries out simple and local operations on the topological data structure. One important property is the application of parallel *spiral search* findings and neighborhood examinations by the many processors. We assume a linear association from input data to processors and memory as the problem size increases, when dealing with Euclidean optimization problems. This property is a precondition of our work that we explicitly state in order to deal with large size problems in a massively parallel way.

In order to provide a general formulation for the class of problems addressed in this thesis, we propose a grid matching problem definition that embeds both *k*-means and elastic image matching in the Euclidean plane into a single and unified formulation of a generic problem. This general formulation can be instantiated in different ways depending on the problems under consideration. For Euclidean optimization problems, the input data are usually numerical entities distributed onto the plane. Such distribution of data can either be a grid of pixels for image processing applications or a set of points representing requests or customers located on the plane for other applications. The goal of the problem is to find a best matching grid, generally a moving grid, which fits the input data, according to specific requirements of the problem under consideration. For example, the matching grid could represent an image in elastic image matching, or routes of vehicles in routing problems.

The cellular matrix is defined as a hierarchical data structure which is used to partition data distributed in the plane; to access, search and memorize brute data moving onto the plane; and to allow different levels of parallel computation granularity. It is also defined as a set of functions to allow easy manipulation of grid coordinate systems, where the main tools are coordinate transfer functions provided in relation to some subdivisions of the plane at different levels of recursive decomposition. We present the principle of plane and data recursive decomposition with the cellular matrix, which defines a given level of computation granularity, and allows generic and systematic association of processors to the data deployed in the plane. We explain the principle of spiral search and define structures that allow us to access data and perform spiral search findings. Moreover, we provide a generic loop template algorithm that exemplifies the massively parallel projections performed in our approaches, concerning a matching grid projected onto another matching grid, meanwhile allowing us to implement different Euclidean optimization algorithms in the cellular matrix model.

3.2 Euclidean Grid Matching Problem

We define a Euclidean grid matching problem version that embeds both *k*-means and elastic image matching in the plane into a single and unified formulation of a generic problem. This general formulation will be instantiated in different ways depending on the problem under consideration. A grid matching problem involves two Euclidean graphs, matching one onto the other. More precisely, only one grid plays the role of the moving grid, which embeds both variables and some input values of the problem, and which will match the other grid, being only considered as input. The former moving grid is called the *match**er*** grid, where as the latter the *match**ed*** grid.

Let us first define basic notations. A matching grid is a non directed graph $G = (V, E)$, called the network, where each vertex $p \in V$ is a composite vector of $\Re^d$ that embeds a location in the plane $p_e \in \Re^2$, color component $p_c \in \Re^3$, intensity or density value $p_d \in \Re$, and other attributes depending on the problem under consideration. Attributes can be variables of the problem or input values depending on the role played by the grid: a matched grid with only inputs, or a matcher grid with inputs and variables. The variable part of the matcher grid at least includes the Euclidean locations of its vertices. The network (graph) $G = (V, E)$ is always a regular grid in some topology (either one-dimensional or two-dimensional depending on the problem under consideration) that represents the elastic structure of the problem, where the edges of the graph represent topological relationships in data, or links between pixels, or connections of successive points in a TSP tour. As usual, d_G is the induced canonical metric $d_G(v, v') = 1$ if and

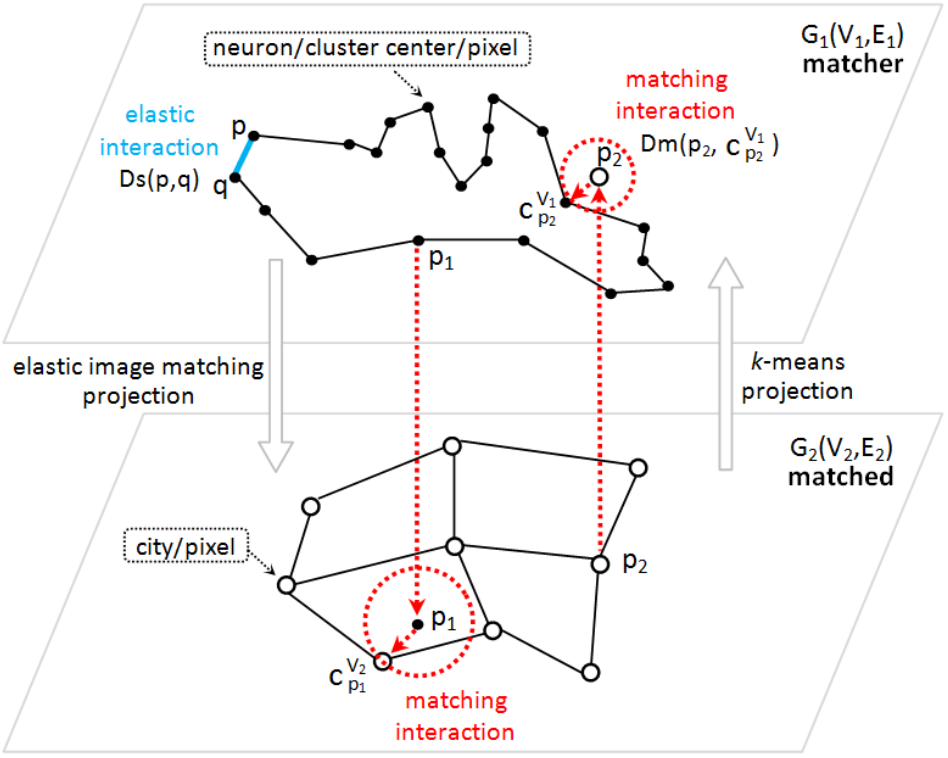


FIGURE 3.1: Grid matching illustration.

only if $(v, v') \in E$, and $d(v, v')$ stands for the usual Euclidean distance between location attributes.

To each point x in the plane, c_x^V denotes its closest vertex in set V, for some composite distance function, including composition with the standard Euclidean norm between location attributes. Thus, c_x^V is called Voronoi projection of point x into V.

The input of the problem is given by two matching grids $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, such that G_1 is the matcher grid and G_2 the matched grid. The goal of the problem is to find the matcher vertex locations in the plane, and its other variable attributes, such that the following energy function

$$E(G_1) = \sum_{p \in V} D_m(p, c_p^{V'}) + \lambda \cdot \sum_{\{p,q\} \in E_1} D_s(p, q) \tag{3.1}$$

is minimized, where (V, V') stands for (V_1, V_2) or (V_2, V_1), depending on the direction of the projection under consideration, while the projection direction itself depends on the problem under consideration. The first term of the energy function expresses the data matching interaction with composite distance matching function D_m, and the second term expresses the smoothness or elastic constraint on the matcher with D_s smoothing function.

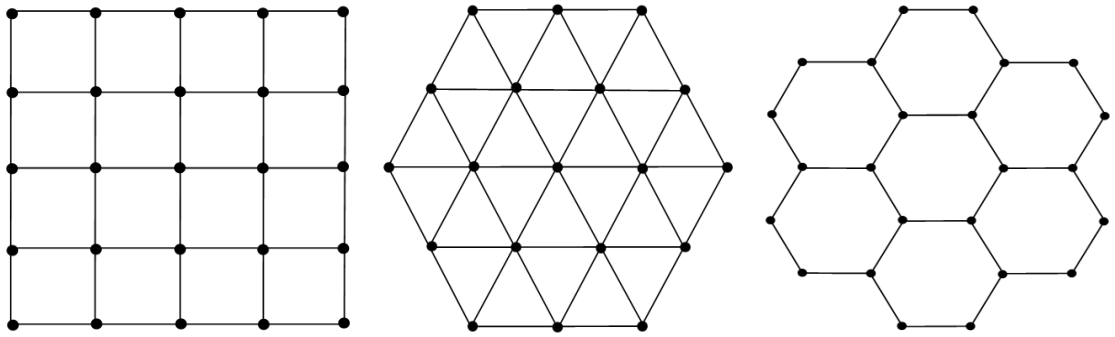
An illustration of the network components and their interaction matchings in the plane is given in Figure 3.1. The upper part represents the marcher grid $G_1 = (V_1, E_1)$, with vertices that could be neurons of the SOM network, cluster centers of *k*-means, or pixels of elastic image matching. The lower part represents the marched grid $G_2 = (V_2, E_2)$, with vertices that could be cities of the TSP, or pixels of topological *k*-means and elastic image matching. The red part in Figure 3.1 indicates the closest point projection on both directions: the projection of p_1 is from the matcher grid G_1 to the matched grid G_2, corresponding to elastic image matching problems; and the projection of p_2 is from the matched grid G_2 to the matcher grid G_1, corresponding to topological *k*-means problems. Based on the projection, the data matching interaction (red color) D_m is defined accordingly. The smoothness term of elastic interaction D_s for neighboring vertices (p and q in the matcher grid G_1), is denoted by blue color. Note that for projections on both directions, the data and smoothness terms share similar structures. These natural common structures yield basic parallel processing functions and components that constitute the cellular matrix framework.

3.3 Cellular Matrix Concept

We define the *cellular matrix* as a hierarchical data structure which is used (1) to partition data distributed in the plane, based on some regular topology; (2) to access, search and memorize brute data moving onto the plane, according to the topology; and (3) to allow different levels of parallel computation granularity. The cellular matrix is also defined as a set of functions to allow easy manipulation of grid coordinate systems, where the main tools are coordinate transfer functions provided in relation to some subdivisions of the plane at different levels of recursive decomposition. In this section, we firstly describe the basic hierarchical structure which constitutes the skeleton of cellular matrix model. Then, we report the possibility for recursive decomposition based on the basic hierarchical structure. Afterwards, we provide implementations in different topologies.

3.3.1 Basic Hierarchical Structure in the Plane

The starting points of the cellular decomposition of the plane are three possible tessellations [Sto97] of the plane with regular polygons of the same type. The three tessellations are presented in Figure 3.2 (a)—(c). They correspond to dividing a plane into regular squares, triangles, and hexagons, respectively. The three tessellations will be the basis for our regular discretization of the plane into the cellular matrix.



(a) square tessellation (b) triangular tessellation (c) honeycomb tessellation

FIGURE 3.2: Three possible tessellations of a plane with regular polygons.

To be more specific, the square tessellation will be the basis for the cellular matrix in quad topology or rhombus topology; the triangular tessellation and the honeycomb tessellation will together be the basis for the cellular matrix in hexagonal topology. More details are given in Subsection 3.3.3.

The cellular matrix consists of three essential levels of discretization of the plane. Each level is in close relationship to the others, as shown in Figure 3.3 in 3D view and in Figure 3.4 in 2D view for juxtaposition of the grids. First is the low level grid on the bottom of the figure where nodes represent pixels or basic input data. Second is the zoom-out grid (also called base grid) of low level grid. Third is the cell centers grid called dual grid.

The low level can be viewed as a discretization of the plane according to a given topologically regular grid. Here, we use hexagonal topology for illustration where each node has 6 neighbors and where honeycomb cells are easily abstracted. We can state that a cellular matrix is given by a regular grid dimension $W \times H$, a specific point C that locates in the center of the grid, and a topology type. From the low level, are computed two higher-level grids for covering the plane at a higher level of granularity. They are the zoom-out grid and, what we call, its dualization grid, respectively presented at middle and top of Figure 3.3 and Figure 3.4.

The zoom-out regular grid is obtained from the low level grid considering a neighborhood around the center point C and replicating the new hexagonal cell obtained with the center and the corner points. The level of the zoom-out map is adjusted with the cell radius R, in the sense of the topological distance d_G. Width and height ($WZ \times HZ$) are computed such that the covering is guaranteed. Basing upon zoom-out level and center C, the dualization consists in extracting only the honeycomb cell centers from the zoom-out grid. The dual grid, in blue color in Figure 3.3 and Figure 3.4, is the usual geometric dual of the honeycomb tessellation derived from the zoom-out map.

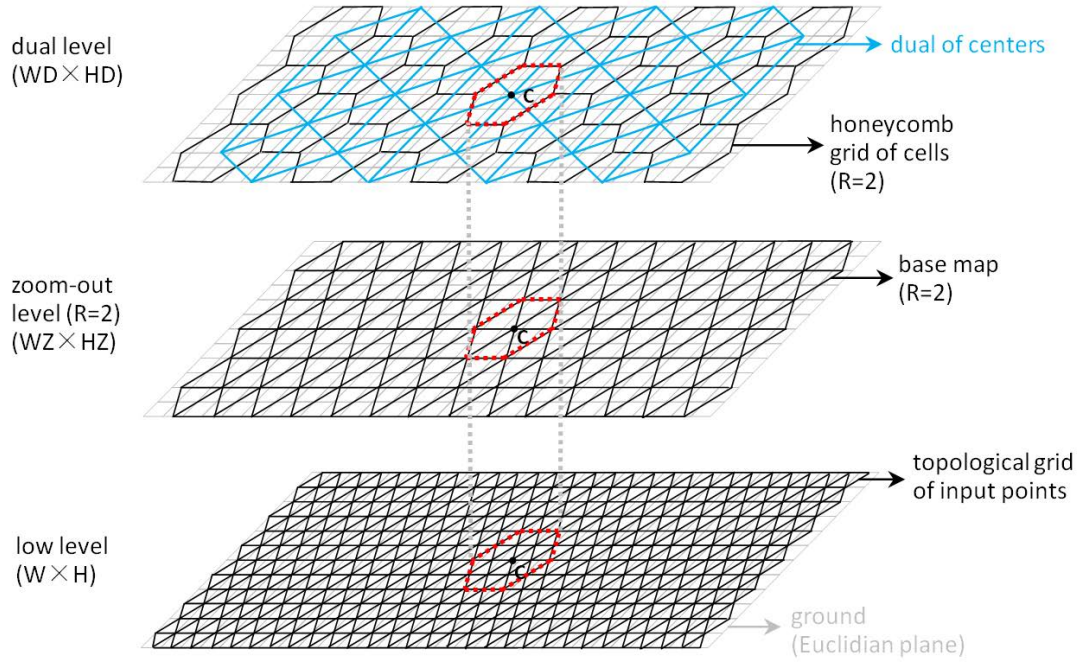


FIGURE 3.3: Cellular matrix model with hexagonal topology (3D view).

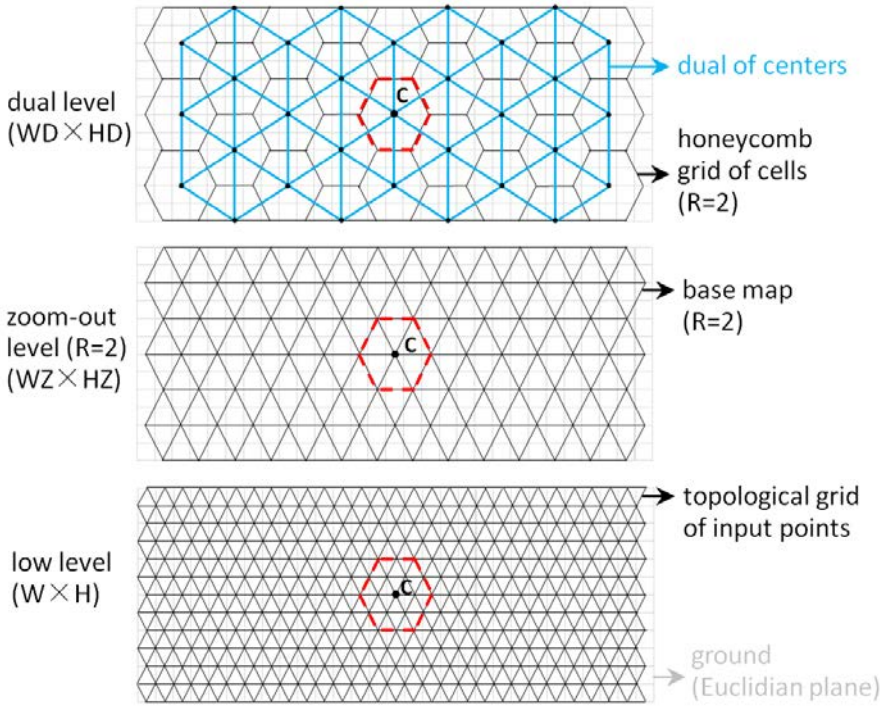


FIGURE 3.4: Cellular matrix model with hexagonal topology (2D view).

This duality corresponds to the usual Voronoi partition or Delaunay triangulation of data points.

Based on the basic hierarchical structure in the plane, the cellular matrix framework is aimed to provide the coordinate transfer functions that allow direct accesses to input data coordinates at the different grid sub-levels, meanwhile making systematic

association of processors to the massive data. Since parallel processors are usually organized into regular 2D grids, we start from the idea of systematic association of data grids to processor grids. A main principle, is the identification of data grids with processor grids at a given level of abstraction. Most often in our work, it is the dual grid which constitutes the processor grid and where parallel computation takes place. Honeycomb cells are the covering regions associated to the dual grid of processors. The association is one cell corresponds to one processor.

3.3.2 Possibility for Recursive Decomposition

On the basis of the three basic levels of the cellular matrix, more levels could be continually introduced, for further recursive decomposition. As shown in Figure 3.5 as 3D representation and in Figure 3.6 as 2D view for grid juxtaposition, we repeat the process of dualization from the dual grid. First we take "the dual of the dual" to obtain a zoom-out grid at level $R' = 3R$, as shown in green color in Figure 3.6, then again we take the dual of the green map to generate a new dual level (orange grid) for process computation according to a higher honeycomb subdivision (the green one). This new level, obtained within a two step dualization, is called the dual^2 level since it repeat the original honeycomb tessellation at a higher zoom-out level, i.e. at level $R' = 3R$.

We can repeat the dualization process for recursive decomposition. Eventually, all data will be included by one single honeycomb cell at some higher level, as illustrated on top of Figure 3.5. Parallel reduction operations could take place from the lowest level up to the highest level using adequate coordinate transfer functions. However, we do not implement recursive decomposition in this thesis. Reductions on sum computation are done in a classical way with unidimensional reduction. Further works should investigate recursive decomposition since it relies on quad-trees and spatial reductions that could allow both parallel data computing, quick data access and memorization in a recursive way.

3.3.3 Cellular Matrix in Different Topologies

The previous illustrations of cellular matrix model are with hexagonal topology. As we have seen, hexagonal topology is built upon the triangular tessellation (see Figure 3.2 (b)) of the low level. This topology allows us to consider honeycomb (hexagon) cells of the hexagonal neighborhood regular grid. Two other topologies can be derived from the square tessellation (see Figure 3.2 (a)). They can be quad topology and rhombus topology, as presented in Figure 3.7. Note that we do not use

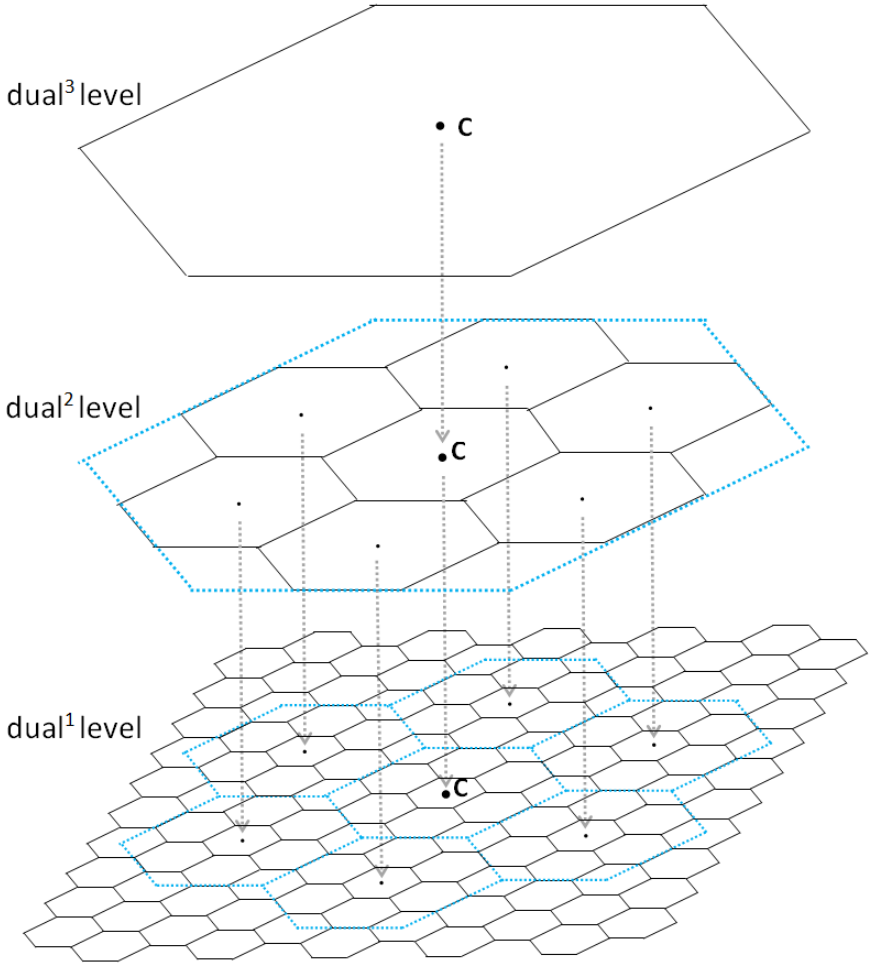


FIGURE 3.5: Recursive decomposition of the plane (3D view).

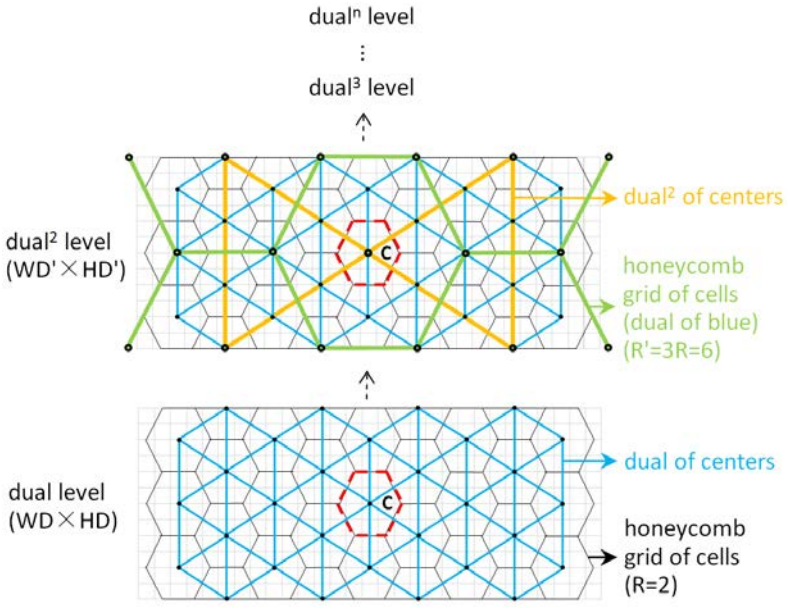


FIGURE 3.6: Recursive decomposition of the cellular matrix model (2D view).

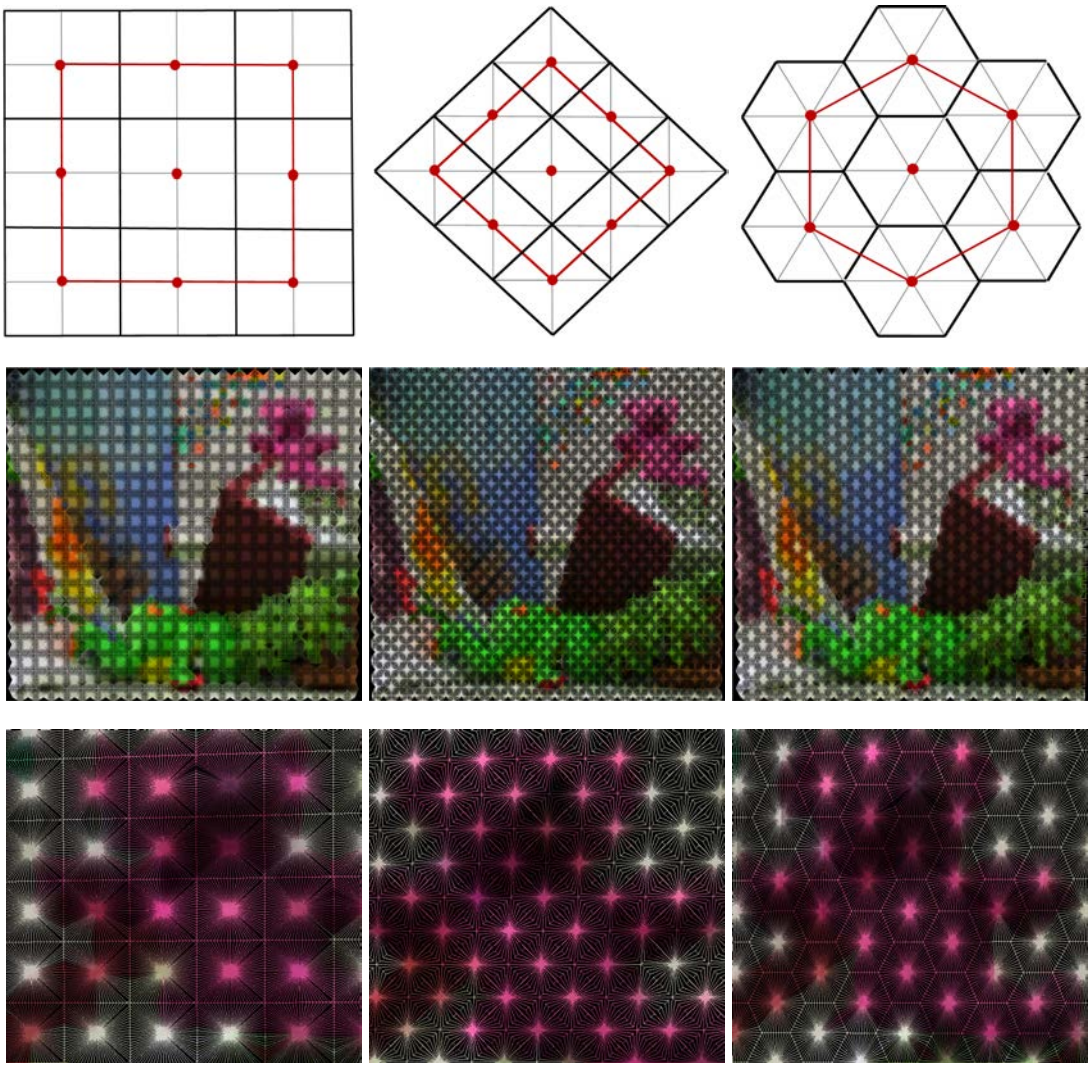


FIGURE 3.7: Cellular matrix illustration with different topologies. The left column is with quad topology; the middle column is with rhombus topology; the right column is with honeycomb topology. Top row: the abstract cellular level derived from the regular base map topology. Second row: covering cellular matrix illustration. Third row: zoom in of second row. The cell radius (R) is set to 10 for each topology.

the honeycomb tessellation (see Figure 3.2 (c)) for the low level, since the honeycomb tessellation could be derived from the triangular tessellation, by simply making a dualization, as the dual level in Figure 3.3 and Figure 3.4.

In Figure 3.7, the top row presents the abstract cellular level derived from the regular base map topology. The two other rows are illustrations of the cellular covering of a given low level image. In the figure, for each cell, its internal pixels (except for the cell frontier) are projected to the location of the center of the cell.

We implement the three types of topologies, and most of our cellular matrix applications use honeycomb cells under the hexagonal topology. Note that brute data can also be managed by buffered cells. Thus, it is not a strict requirement for a matching grid to be

isomorphic to a low level cellular matrix map. It is a convenient possibility only. Cities or moving grids (such as cluster center grid) in the plane can be unstructured and managed by buffering, whereas images are generally isomorphic to low level grids. There is no strict selection criterion for different topologies, and the choice should depend on the application under consideration. For example, the hexagonal topology is more appropriate for cluster center grid, which plays the role of matcher grid in the structured meshing application. Actually in most of applications, we did not find great performance differences using this or that topology for the cellular matrix.

3.3.4 Massive Parallelism Property

The role of the cellular matrix is to memorize data in a distributed fashion and authorize massively parallel operations. Each cell is responsible for a constant and small part of the data according to the problem size. While data decomposition is not a new way of dealing with parallelism in general, here, we specifically assume a linear association from input data to processors as the problem size increases, when dealing with Euclidean optimization problems. This is the main property that we refer to as “massive parallelism”, and it allows us to address large size problems. Here by massive parallelism, we mean the theoretical and ideal possibility to execute $O(N)$ simultaneous parallel operations, where N is the problem size. It is worth noting that uniform distribution corresponds to the most balanced cellular decomposition and hence corresponds to the equilibrated multi-processor load.

Let us suppose that the input data is with size $W \times H$, and the cellular matrix is with quad topology, for the sake of easy analysis. Then, the cellular matrix partition (dual level grid of cells) is with size $WD \times HD = W/2R \times H/2R$, where the parameter R is the radius of cell measured by input data unit. So, R controls the degree of parallelism, and the cellular matrix is in linear relationship to input size. Each uniformly sized cell in the cellular matrix is a basic training unit and will be handled by one parallel processor. This is the level on which massive parallelism takes place. Since the cellular matrix division is proportional to input size, and the processors correspond one-to-one to the cells respectively, then, both the memory and processors needed are in linear relationship to input size. Hence, according to the increase of physical parallel processors in the future, the approach should be more and more competitive, while at the same time being able to deal with larger size problems.

Based on the cellular matrix model, each parallel processor can carry out spiral search for closest point/node finding. More details about the spiral search are given in the next subsection. A single spiral search process takes $O(1)$ computation time, on average,

for a bounded distribution according to input size [BWY80]. In our implementation of the cellular matrix model, the bounded distribution is addressed by using cell buffers (see the buffered cells in the next subsection) of fixed size independent of N, where $N(= W \times H)$ is input size. Then, one of the main interests of the cellular matrix model is to allow the execution of approximately N spiral searches in parallel, and thus transforming an $O(N)$ sequential search algorithm into a parallel algorithm with theoretical constant time $O(1)$ in the average case for bounded distributions. This is another property that we refer to as "massive parallelism", the theoretical possibility to reduce computation time by factor N, when solving a Euclidean NP-hard optimization problem.

3.4 Spiral Search in Data Grids

For the implementation of closest-point search, we perform the spiral search algorithm as stated by Bentley [BWY80]. The expected running time of spiral search is $O(1)$ for multivariate uniform distributions over the data. Depending on the type of cell, and the level of operation into the cellular matrix, different iterators allow us to operate step by step on the data, starting from a center point and in a spiral way, independently of the topology used. In this section, we present notations and symbols for data structures and detail the spiral search paths into the different topologies.

3.4.1 Cell Data Structures

How the cells are in relation to the grid matching data is a primordial question. Data structures in relation to the cells are visualized in Figure 3.8. Two types of cells are necessary. One type is required for nodes memorization when the graph to manage is distorted or unstructured. Nodes covered by a cell are memorized into a buffer associated to the cell in sequential order. Such cells are called *buffered cells*. The second type is *spiral cell*. In that case no buffering memorization is required, and accesses are directly into the matching low level grid associated to the cell. This is only possible if the cell is at some zoom-out level of the low level map. A transfer of coordinates into the cellular matrix allows directly accessing the data and performing spiral search, or memorization in the matching (matcher or matched) low level grid itself. Such cells are called spiral cells.

With the technique of "cells" and the topological organization of the cellular matrix model, we can perform an efficient spiral search [BWY80] in optimal expected-time, for

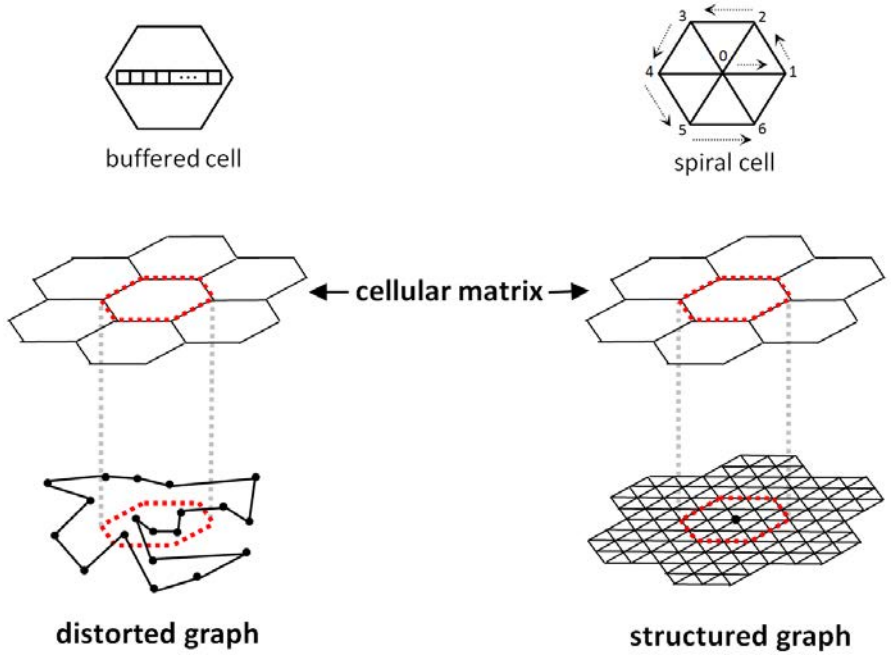


FIGURE 3.8: Cellular matrix data structures.

many closet-point problems, including nearest neighbor searching, finding all nearest neighbors, computing planar minimum spanning trees, and constructing the Voronoi diagram of a point set. We assume that the point sets are either uniformly distributed or randomly drawn from some "smooth" underlying distribution, and then use the cell technique to give fast expected-time algorithms for these problems.

3.4.2 Nearest Neighbor Searching

Among many closest-point problems, the *nearest neighbor searching*, sometimes called the *post office problem*, is easiest to state meanwhile can most clearly illustrate the spiral search algorithm based on our cellular matrix model. It calls for organizing a set S of n points in Euclidean *k*-space, into a data structure such that subsequent queries asking for the nearest point in S to a new point can be answered quickly. Here, we consider the problem of nearest neighbor searching in the Euclidean plane, where the points (both the original n points and the query point) are chosen independently from a uniform distribution over the unit cell. Through the data partition of the cellular matrix, each point is exclusively contained by one cell (except for the points locate on cell frontiers). When a query point comes in, we search the cell in which it would locate. If that cell is empty, then we start searching the cells surrounding it in a spiral-like pattern until a point is found. Once we have one point found, we are guaranteed that there is no need to search any cell that does not intersect the circle centered at the

query point and with the radius equal to the distance between the query point and the found point.

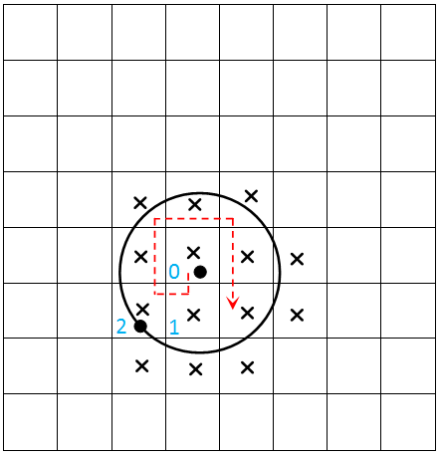
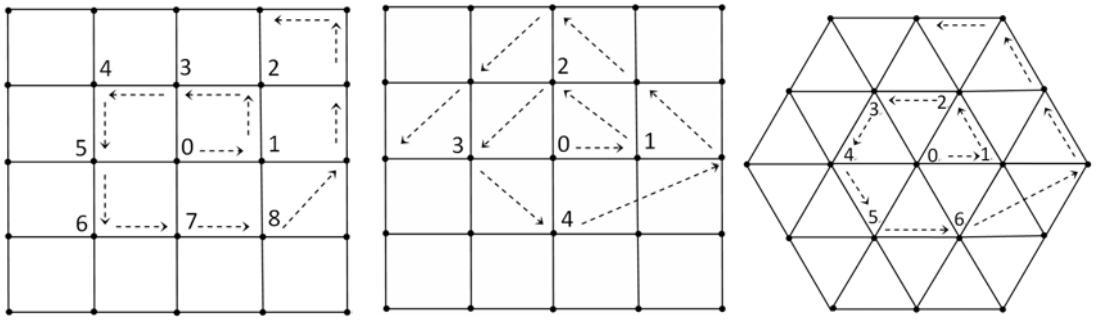


FIGURE 3.9: Spiral nearest neighbor search in the cellular matrix model.

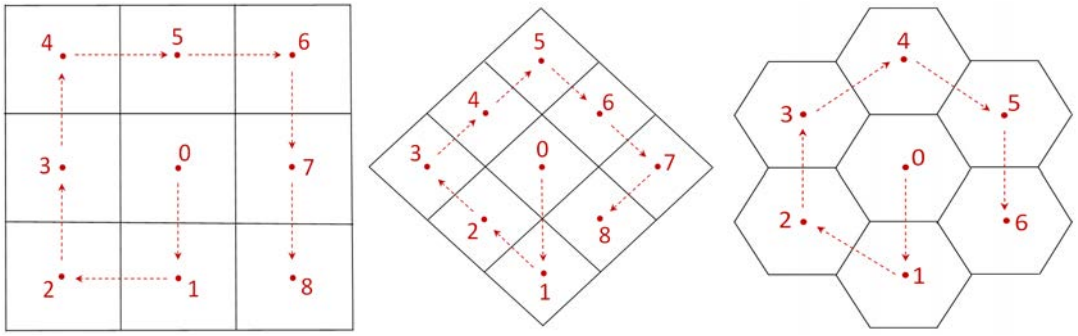
Figure 3.9 shows how the spiral search might proceed in the cellular matrix with quad topology. For the query point in the middle of the circle, the cell (Cell 0) in which this query point lies will firstly be searched. If this cell is empty, then the cells surrounding it are searched one by one in the spiral sequence as indicated by the red dashed line, starting from the downside neighbor cell (Cell 1). Once the point in Cell 2 is found, only cells intersecting the circle must be searched. Each of these cells is marked with an "x".

The spiral nearest neighbor search is efficient because closest-point problems investigate local phenomena, and cells capture locality. It has been proven in [BWY80] that the expected running time of spiral search is in $O(1)$, independent of the value of n, if n points are chosen independently from a uniform distribution over the cells. Then given the fast spiral nearest neighbor search algorithm, one can easily solve the *all-nearest-neighbors* problems (which calls for finding the nearest neighbor of each point) in linear expected time, for point sets drawn from uniform distributions. This is accomplished by doing n spiral searches, each of expected constant cost. Here, it is important to note that our cellular matrix model allows the execution of n spiral searches in parallel, and thus transforming an $O(n)$ sequential search algorithm into a parallel algorithm with theoretical constant time $O(1)$ in the average case. In our implementation of the cellular matrix model, the bounded distribution for point sets is addressed by using the buffered cell whose size is independent of n.



(a) quad topology (b) rhombus topology (c) hexagnal topology

FIGURE 3.10: Spiral search on low level data.



(a) quad topology (b) rhombus topology (c) hexagnal topology

FIGURE 3.11: Spiral search on cellular matrix.

3.4.3 Spiral Search in Different Topologies

In the example of Figure 3.9, the spiral search is performed on the cellular matrix level through buffered cells. Moreover, the spiral search can also be done on low level data through spiral cells. Here, the convention should be counter-clockwise order for low level data search and clockwise order for cellular matrix data search because of the specific coordinates arrangement when dealing with a dual grid. The searching trajectories on low level data in different topologies are shown in Figure 3.10, while the searching trajectories on the cellular matrix level in different topologies are shown in Figure 3.11.

During the spiral search performed on the low level data through spiral cells, for a given distance of d in the grid to the center vertex, the number of vertices at this distance is $(d \times size)$, while the number of vertices within this distance (distance d included) is $((d \times (d+1) \times size)/2 + 1)$. As shown in Figure 3.10, $size$ is equal to 8, 4, and 6, respectively for the cellular matrix with quad topology, rhombus topology, and hexagonal topology.

3.5 Basic Operations

In the cellular matrix model, data are partitioned by a grid of cells, and each cell contains one part of the data. The grid of cells allows systematic association to grid of processors, in such a way that each processor locally acts on the partial data the cell covers, and if necessary, the data around the cell by extending the search to neighboring cells. Under this mechanism, several fundamental operations are defined in this section, as building blocks of different optimization algorithms.

3.5.1 Generic Parallel Projection Loop

The most basic type of operations to be carried out in the optimization algorithms studied in this thesis is parallel projection of a grid into another grid, with specific search and application of some grid moving operation.

Kernel 6: Generic loop template.

```
// For each cell_i of cellular matrix WD × HD, do:
Input: G_1, G_2, cell_i, FGet, FSearch, FOperate
1 begin
2     FGet.initialize(cell_i);
3     while FGet.next(cell_i) do
4         point ← FGet.get(cell_i, G_1(or G_2));
5         if point then
6             closest ← FSearch.search(cell_i, G_2(or G_1), point);
7             if closest then
8                 FOperate.operate(cell_i, G_1, point, closest);
9     return G_1;
```

A generic projection loop is illustrated in Kernel[1] 6. Three basic functions are used as parameters of the projection. Such functions are called *functors*, and they are $FGet$, $FSearch$, and $FOperate$ as template arguments. The functor $FGet$ allows traversing the data units (points, pixels, cluster centers, et al.) in the cell and enumerating such data points one by one. Normally, it is implemented by a specific iterator according to the specific topology in hand. For the data points, let us say pixels of image, that locate on cell frontiers, they are considered as inside the cell, and will be enumerated by the $FGet$ functor. Therefore, these pixels are not only included by one cell. As a result, during the parallel executions of different threads for different cells, these cell frontier

[1] We use the term "Kernel" instead of "Algorithm" in order to indicate that the algorithm is performed by many processors in parallel. In our GPU CUDA implementations, this corresponds to a kernel function which is executed by many GPU threads in parallel.

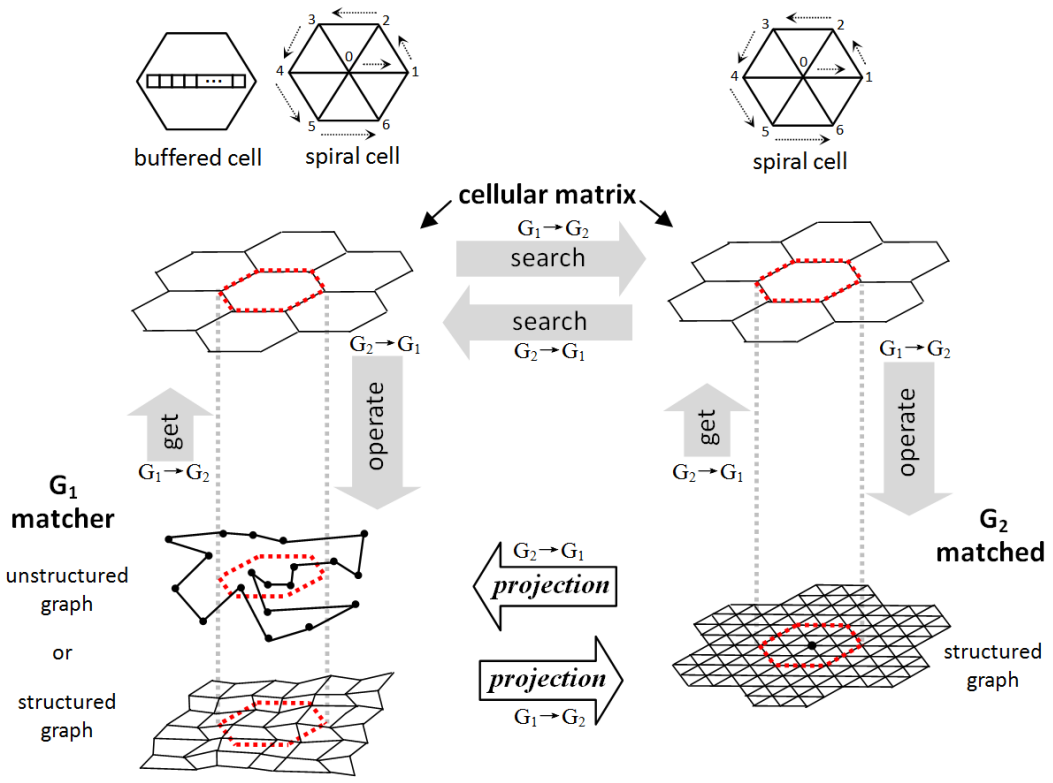


FIGURE 3.12: Basic projections through cellular matrix model.

pixels may be accessed by more than one thread at the same time. Then, conflict operations will happen if the following operations by the other functors are related to moving the pixels, such as the situation in the elastic image matching problem. To handle this issue, we propose two specific strategies for the management of cell frontier access in Subsection 6.3.4 of Chapter 6, where more details are given. The functor $FSearch$ allows performing searching into the data, such as spiral search for closest finding, under some distance metric, and possibly local condition. The functor $FOperate$ allows carrying out some specific moves or modifications of variables.

Each cell is supposed to correspond to one processor, and the size of the cellular matrix (dual level size) is supposed to be the same as the size of the processor grid. However, under some parallel computing architecture, such as GPU implementation, the size of the actually launched grid of threads (processors) for parallel execution is sometime larger than the size of the cellular matrix. Therefore, at the beginning of the generic loop template, we need to ensure that the thread really encapsulates a valid cell.

For a valid thread, it firstly initializes the $FGet$ functor by $FGet.initialize$ as illustrated in Line 2 of Kernel 6. Basically, this function resets the iterator of $FGet$. Then, points (data units) will be extracted inside the cell by the $FGet.get$ function, until $FGet.next$ returns false state and the data extraction procedure stops, as illustrated in Line 3. Note that in Kernel 6, the terms *point* and *closest* are all used to denote particular data

units. For different algorithms, the points supposed to be extracted in one loop may be one randomly chosen point, or a portion of the total points inside the cell, or all the points inside the cell. For an extracted point, a searching behavior is carried out by the $FSearch.search$ function, in order to find the closest point to the extracted point. If the closest point is successfully found, then some specific operations defined by specific algorithms are performed by the $FOperate.operate$ function, on the extracted point and the closest point, or even with some other relevant points. For example, these operations could be affiliating the extract point with its closest point, as in the Voronoi partition computation (see the next subsection), or moving the closest point and its neighboring points toward the extracted point, as in the SOM algorithm.

A vivid graphical illustration of the generic projection is given in Figure 3.12. The gray arrows represent functors, while the white arrows indicate projection directions. Arrows denoted with "$G_1 \rightarrow G_2$" indicate the projection from the matcher grid to the matched gird. On the contrary, arrows denoted with "$G_2 \rightarrow G_1$" indicate the projection from the matched grid to the matcher gird. Normally, the matched grid is a structured graph, and the spiral cell data structure is needed for the "get" functor to access points. For the matcher grid, it could be either unstructured graph or structured graph, depending on the algorithm under consideration. In the case of unstructured graph, the buffered cell data structure is needed. In the case of structured graph, **either** the buffered cell data structure is needed, if the spiral search on the cellular matrix level is performed on the structured graph which moves and distorts, like in the case of the parallel SOM algorithm (see Chapter 4); **or** the spiral cell data structure is needed, if the "get" functor needs to access points according to the graph structure, like in the case of distributed local search algorithm (see Chapter 5).

3.5.2 Voronoi Partition Computation

One of the most basic tools one can use in planar geometry is Voronoi partitioning. We will use it thoroughly in this thesis. A *Voronoi diagram* is a partitioning of a plane into regions based on distance to points in a specific subset of the plane. The Voronoi diagram of a point set in the Euclidean plane is a device that captures many of the closeness properties necessary for solving closest-point problems. For any point x in a set S, the *Voronoi polygon* of x is defined to be the locus of all points that are nearer x than any other point in S. Notice that the Voronoi polygon of point x is a convex polygon with the property that any point lying in that polygon has x as its nearest neighbor. The union of the edges of all the Voronoi polygons in a set forms the Voronoi diagram of the set. Normally, if the distance is measured by the Euclidean distance metric, then

an edge for a pair of points that are close together is formed by drawing a line that is equidistant between the two points and perpendicular to the line connecting them.

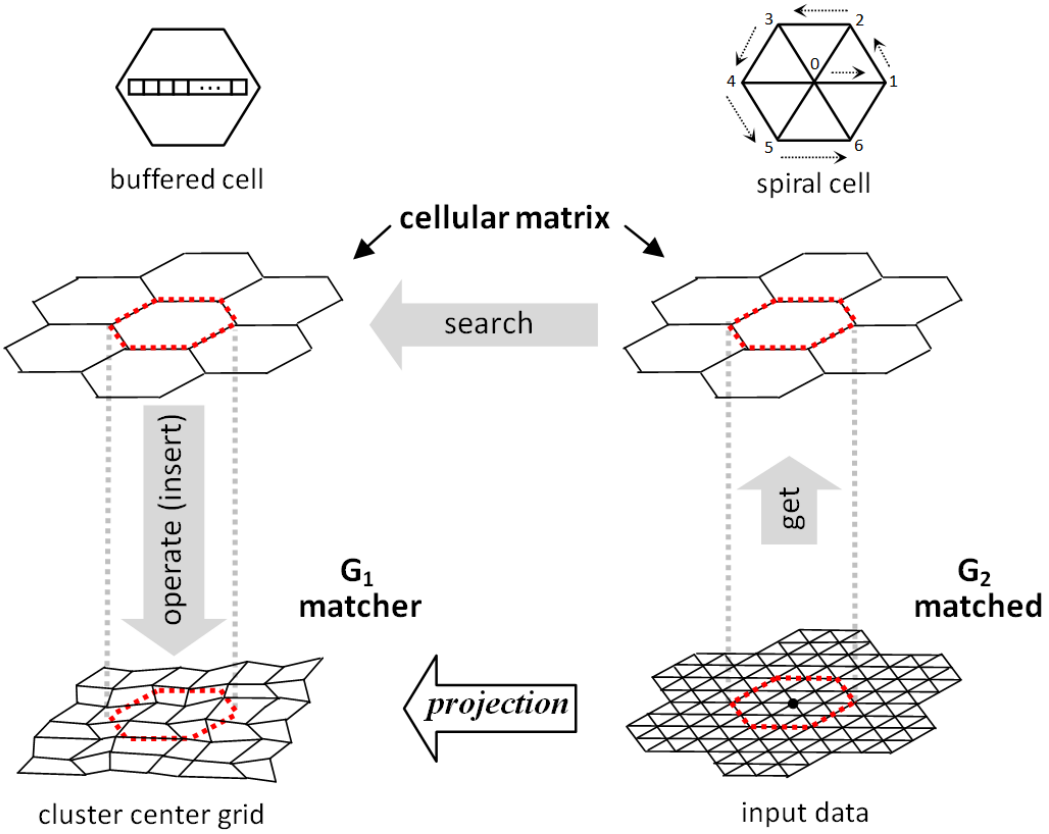
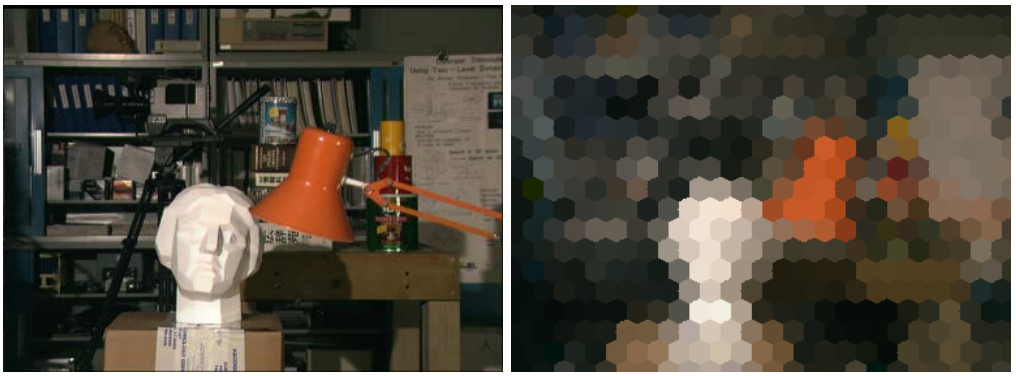


FIGURE 3.13: Voronoi projection model.

It has been proven in [BWY80] that constructing the Voronoi diagram with spiral search algorithm takes linear expected-time, under the assumption of point sets drawn from a bivariate uniform distribution. Here, we only perform Voronoi affectation, meaning that we will only have to project points from some brute data onto their closest cluster center of some cluster center grid. The structure of the projection loop instantiation can be illustrated within Figure 3.13, where the projection direction is from the matched grid to the matcher grid. It consists in projecting each input data point to its closest vertex into a cluster center grid. To be more specific, the $FGet$ functor extracts all the input data points inside the cell. For each extracted point, the $FSearch$ functor finds the closest cluster center in the matcher grid by spiral search. Then, the point is affiliated to this cluster center by the $FOperate$ functor, which inserts the point into the corresponding recording list for this cluster center. Since the $FSearch$ functor performs the spiral search on the cluster center grid that can be a distorted grid, the buffered cell data structure is needed.

An example of Voronoi partition is shown in Figure 3.14. In this case, the input data is a color image as the matched grid, where the points are pixels. After the Voronoi



(a) input data (b) Voronoi partition

FIGURE 3.14: An example of Voronoi partition with an image as input data. The input image size is 384×288; the size of cluster center grid is 25×21; the cell radius (R) in the cellular matrix is set to 16.

partition, as shown in Figure 3.14 (b), each pixel is affiliated to its closest cluster center, with its color value set according to its closest cluster center. Here, the color of a cluster center is the average color of all its affiliated pixels.

3.5.3 Cell Refresh Kernel

For a given cell in the cellular matrix model, it contains points of the input data or moving points in the plane. When dealing with moving points in the plane and closest point findings in cellular matrix buffers, it should be necessary to regularly refresh the content of the buffered cells. This behavior is implemented by a cell refresh loop kernel presented by Kernel 7. The functor $FGetDivide$ uniformly divides the K points to be inserted among cells of the cellular matrix. For a given point, the function $FSearchFindCell.search(point)$ finds which cell this point lies in, while the function $FInsert.operate(point, cell_j)$ inserts the point to the designated cell. In practice, it is sufficient to make the refreshing from time to time depending on the application.

3.5.4 Random Number Generation Kernel

During stochastic processes, a cell needs random numbers. The random numbers can be used for random cell activation, operator choices, random neighborhood examinations, roulette wheel extraction. With respect to the large-scale input instances with huge cellular matrix and numerous iterations, the random numbers are generated in advance by a specific kernel. Every time the random numbers are used out, a set of new random numbers are generated at the beginning of the procedure, or according to a constant rate factor, called *random number generation rate*. The random number

Kernel 7: Cell refresh loop.

```
// For each cell_i of cellular matrix WD × HD, do:
Input: G, FGetDivide, FSearchFindCell, FInsert
1 begin
2     FGetDivide.initialize(cell_i);
3     while FGetDivide.next(cell_i) do
4         point ← FGetDivide.get(cell_i, G);
5         if point then
6             cell_j ← FSearchFindCell.search(point);
7             if cell_j is in the cellular matrix then
8                 FInsert.operate(point, cell_j);
9     return
```

generators we use are from Nvidia CURAND library [NVI12]. It is worth mentioning that alternative ways to generate random numbers are either to generate them on CPU, then copy them into GPU, or to generate them on GPU in real-time at each time to be used. Neither is faster than our method according to our trial tests.

3.6 Conclusion

The role of the cellular matrix is to memorize data in a distributed fashion and authorize massively parallel operations. Each cell is responsible for a constant and small part of the data according to the problem size. We specifically assume a linear association from input data to processors as the problem size increases, when dealing with Euclidean optimization problems. This is the main property that we refer to as “massive parallelism”, and it allows to address large size problems. Here, by massive parallelism, we mean the theoretical and ideal possibility to execute $O(N)$ simultaneous parallel operations, where N is the problem size. Based on the cellular matrix, we can perform efficient spiral search with constant time $O(1)$ in average for uniform distribution. Then, one of the main interests of the cellular matrix model is to allow the execution of approximately N spiral searches in parallel, and thus transforming an $O(N)$ sequential search algorithm into a parallel algorithm with theoretical constant time $O(1)$ in the average case for bounded distributions. This is another property that we refer to as “massive parallelism”, the theoretical possibility to reduce computation time by factor N, when solving a Euclidean NP-hard optimization problem.

We have presented the detailed design of the cellular matrix model which partitions data, defines the level of computation granularity, and allows generic and systematic association of processors to the data deployed in the Euclidean plane. We have

provided a set of basic concepts and tools needed for the further developments of parallel computation in the cellular matrix model. An important tool is our proposed generic loop template that encompasses the basic working procedures of the parallel algorithms developed in this thesis. Moreover, we have presented the Voronoi partition computation as a simple application of the generic loop. In the following chapters, algorithms to be developed are parallel *k*-means and parallel local search.

Chapter 4

Parallel Topological *K*-means for Superpixel Image Segmentation

4.1 Introduction

Superpixels have become an essential tool to the vision community. As building blocks of many vision algorithms, superpixels divide raw image into perceptually meaningful atomic regions which can be employed to substitute the rigid structure of the pixel grid [ASS+12, RM03, ME07]. Therefore, these atomic regions should represent or reflect some local properties with respect to the attributes distributions of the raw image. However, most of the existing algorithms produce uniformly distributed superpixels, as it is the case for the state-of-the-art SLIC algorithm [ASS+12] to which we refer in our comparative study. In this chapter, we propose the *superpixel adaptive segmentation map* (SPASM) algorithm, which is a combination of the on-line and batch SOM algorithms for *k*-means clustering in composite color space domain. The goal should be to generate adaptive segmentation map where the distribution/density of superpixels coincides with the distribution of some specified attribute of the input image, such as edges, textures, and depths.

Based on the cellular matrix model, this chapter focuses on the designs of parallel topological *k*-means algorithms and the corresponding applications to superpixel image segmentation problems. Firstly, we provide a general energy function for topological *k*-means problems in order to highlight the generic components of the proposed parallel computation framework. Then, we present the parallel self-organizing map (SOM) algorithm as a *k*-means based clustering algorithm with topological relationships between cluster centers. The SOM algorithm has two versions: on-line SOM and batch SOM. We show how to use the cellular matrix model to implement both of the two

versions in a parallel fashion. Afterwards, we detail our proposed SPASM algorithm, which is a combination of the parallel on-line SOM and parallel batch SOM algorithms. The SPASM algorithm extends the state-of-the-art SLIC algorithm, by using on-line SOM to deploy the initial cluster center grid, with respect to the density distribution of image attributes and topological relationship between cluster centers; and by using batch SOM with a composite space-color-density distance measure for clustering. Differently from SLIC which performs a restricted nearest point search within a square region, through the cellular matrix model, we can conduct the true closest point finding in a massively parallel way, using the efficient spiral search algorithm under different topologies.

4.2 Topological *K*-means Problem

The *k*-means problem and its related *k*-means standard gradient descend algorithm are well-known and popular clustering algorithms. Here, we are dealing with its extension according to topological interactions as exemplified by the SOM algorithm. As we have shown in Section 2.2, the SOM algorithm can be viewed as a *k*-means based clustering algorithm with topological relationships between cluster centers. In order to easily formulate its cellular matrix implementation, we first formulate the problem itself into our proposed Euclidean grid matching generic problem. The basic parallel computation structure will naturally follow.

To state *k*-means in our framework, we propose a general formulation related to standard *k*-means, elastic net, and *k*-medians, that all consider a combination of matching and smoothing. We define it by instantiating the generic problem of Euclidean grid matching as follows. Given a matcher grid $G_1 = (V_1, E_1)$, where vertices stand for cluster centers, and a matched grid $G_2 = (V_2, E_2)$, where vertices stand for input data distribution in some color space domain, the goal of the topological *k*-means problem is to find the matcher vertex locations in the plane, and possibly its color attributes, such that the following energy function

$$E(G_1) = \sum_{p \in V_2} D_m(p, c_p^{V_1}) + \lambda \cdot \sum_{\{p,q\} \in E_1} D_s(p, q) \tag{4.1}$$

is minimized, with a direction of projection from V_2 to V_1. The matching cost D_m can be either the standard Euclidean distance for clustering in the plane or composite color space distance for segmentation purposes. The smoothness cost D_s, or elastic constraint, is the Euclidean distance in the plane $d(p, q)$, or the squared Euclidean distance. According to the direction of projection, follows the natural gradient descent

type procedure of SOM with the sequence of operations “project into” then “trigger”, driven by the data distribution.

4.3 Parallel SOM Algorithm in Cellular Matrix

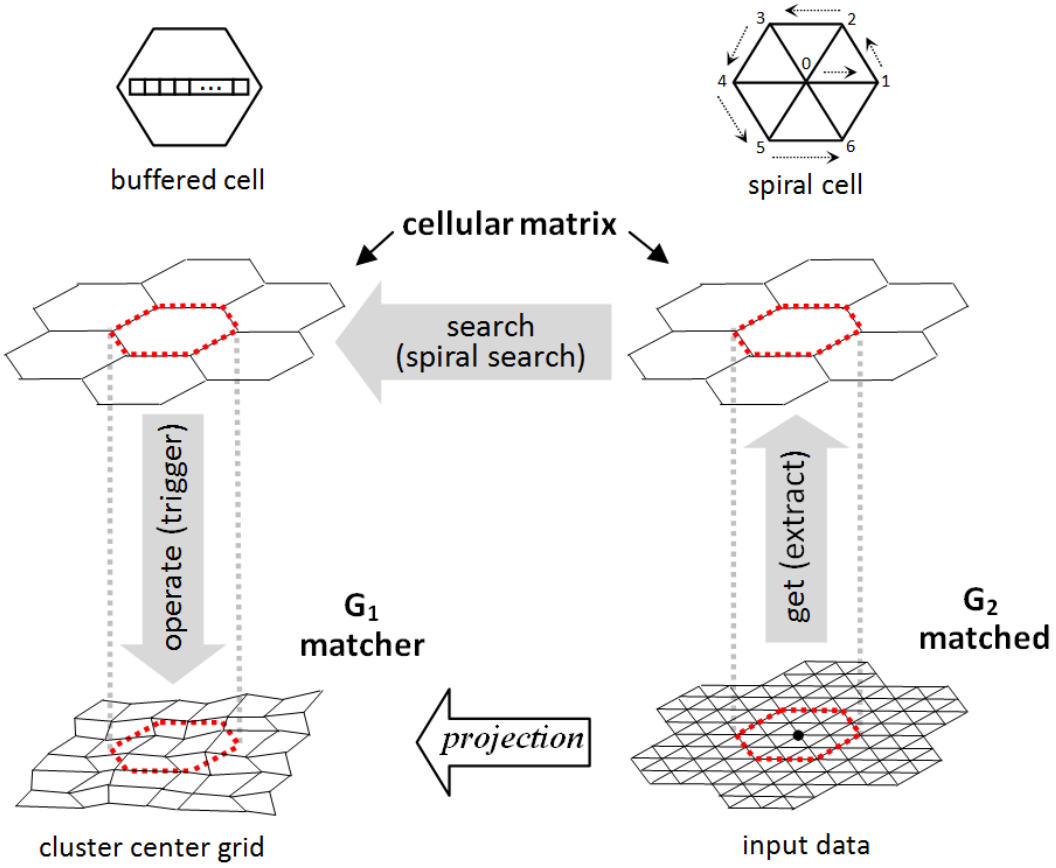


FIGURE 4.1: Basic projection for *k*-means based algorithms.

According to the formulation of the topological *k*-means problem, the data structures and the direction of operations for parallel *k*-means algorithms are illustrated by Figure 4.1. The input data set is deployed on the low level of matched grid, represented by a regular image in the figure. The cluster center grid, represented by the SOM neural network, is deployed in the same plane as the input data, acting as the matcher grid. The honeycomb cells represent the cellular matrix level of operations. Each cell is a basic processing unit that handles a basic SOM processing iteration (see Subsection 2.2.1) with the three steps: the extraction step (**get**) where input data points are randomly extracted according to the density distribution; the competition step (**search**) where the spiral search is performed for the closest cluster center; and the triggering step (**operate**) where the cluster centers are moved toward the extracted

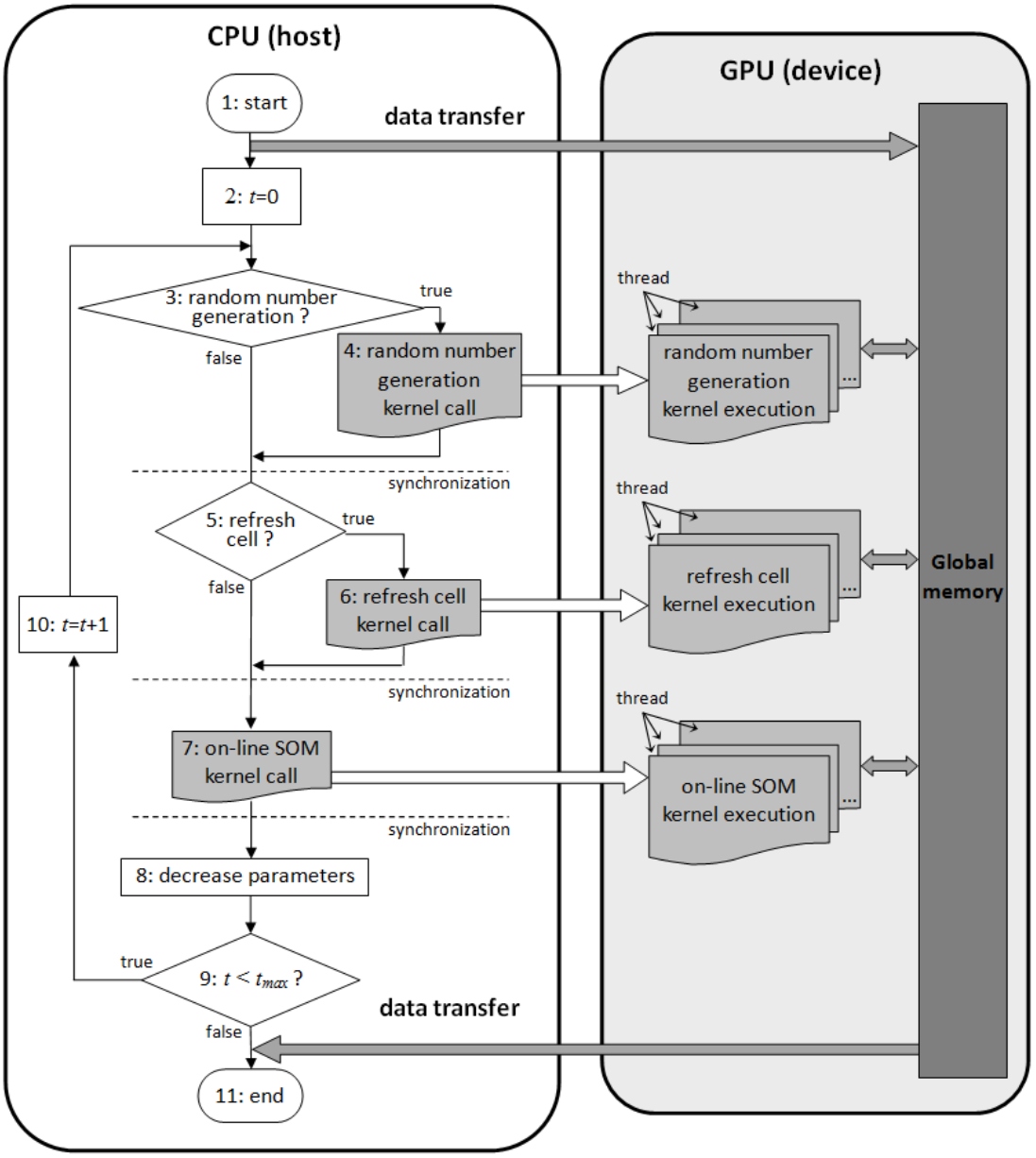


FIGURE 4.2: Flowchart of parallel on-line SOM.

data points. Since the spiral search on the cellular matrix level is performed on the cluster center grid, the buffered cell data structure is needed.

4.3.1 Parallel On-Line SOM

We choose the GPU parallel computing platform to implement parallel optimization algorithms, based on the cellular matrix model which should benefit from GPU's enormous computational power. Under the *compute unified device architecture* (CUDA) programming interface, a GPU works as a SIMT (*single instruction, multiple thread*) co-processor of a conventional CPU. It is based on the concept of kernels which are

functions written in C, called from CPU, and executed by a given number of CUDA threads. These threads will be launched onto GPU's streaming multi-processors (SMs) and executed in parallel [SK10].

In the parallel on-line SOM implementation based on the cellular matrix model, we employ CUDA threads, as the parallel processors, to handle cells in parallel; we use CPU (host code) for flow control and the entire thread synchronization. The data transfer between the CPU side and the GPU side only occurs at the beginning and the end of the algorithm. In Figure 4.2 is reported the flowchart of the kernel calls sequence from the CPU side. The three kernels that are called from the CPU side and executed on GPU are: the random number generation kernel, the refresh cell kernel, and the on-line SOM kernel. In CUDA program, a kernel will not be executed until all the threads for the previous kernel launch finish. Therefore, between separate kernel calls are synchronization barriers, as indicated by the dashed lines in Figure 4.2.

On the GPU side, random numbers are needed for random cell activation and roulette wheel extraction. With respect to the large-scale input instances with huge cellular matrix and numerous iterations, the random numbers are generated in advance by the random number generation kernel stated in Subsection 3.5.4. This kernel is regularly called during the algorithm according to the random number generation rate.

For a given cell in the cellular matrix model, it contains the cluster centers that move in the plane. When dealing with moving cluster centers in the plane and closest point findings in cellular matrix buffers, it should be necessary to regularly refresh the content of the buffered cells. This behavior is implemented by the refresh cell kernel, as the Kernel 7 stated in Subsection 3.5.3.

Now, we only have to detail the on-line SOM kernel. Each GPU thread, that corresponds to a cell, will have to perform the three steps of the sequential SOM iteration in parallel. These steps are the extraction step where input data points are randomly extracted according to the underlying distribution; the competition step where the spiral search is performed for the closest cluster center; and the triggering step where the cluster centers are moved toward the data points. We first focus on the random point extraction step since it is an important step, where the probability for a locally extracted point must reflect the overall density distribution of the input data. Then, we will turn to a detailed description of the on-line SOM kernel.

4.3.1.1 Density Point Extraction

In the original sequential version of the on-line SOM (see Section 2.2), input data points are randomly extracted according to the density distribution in the whole data set. Here in the parallel version, each GPU thread corresponds to a cell, and it performs the basic on-line SOM process based on the data the cell covers. A problem that arises is to allow many data points extracted in parallel by many threads, at a given parallel iteration, to reflect the input data density distribution in the whole data set. As a solution to this problem, we propose a particular cell activation formula

$$pr_i = \frac{q_i}{\max\{q_1, q_2, \ldots, q_{num}\}} \times \delta \tag{4.2}$$

to choose those cells that will execute or not at the considered iteration. Here, pr_i is the probability that $cell_i$ will be activated; q_i is the quantity of the input data that $cell_i$ contains; and num is the number of cells in the cellular matrix model. Formally, q_i is the sum of the density values of the covered vertices. It can be the number of cities that locate in $cell_i$, for a given TSP; or the sum disparity value of all the pixels that locate in $cell_i$, for a given disparity map. Anyway, q_i should reflect the density distribution of the input data, at the cellular matrix level. The empirically preset parameter δ is used to adjust the degree of activity of cells/threads. As a result, the more data density a cell has, the higher is the probability of this cell to be activated to carry out the three steps of SOM execution at each parallel iteration.

At the extraction step, each activated cell performs a local roulette wheel mechanism in the cell itself, in order to get the extracted random point. The probability of a pixel choice local to a cell is defined by

$$pr'_i = \frac{s_i}{\sum_{j=0}^{Npic} s_j} , \tag{4.3}$$

where $Npic$ is the number of pixels in the cell, and s_i is the density value of pixel i in the cell. The roulette wheel mechanism is depicted in Figure 4.3. A sampling example by roulette wheel extraction in the structured meshing application is shown in Figure 4.4, where (b) is a sampling of the disparity map (a), obtained by extracting 10000 points with the roulette wheel mechanism. This sample has been obtained after removing the background small density (disparity) values from the disparity map, and after augmenting the contrast in it. Figure 4.4 (c) is the corresponding mesh result according to the extracted points in the sample.

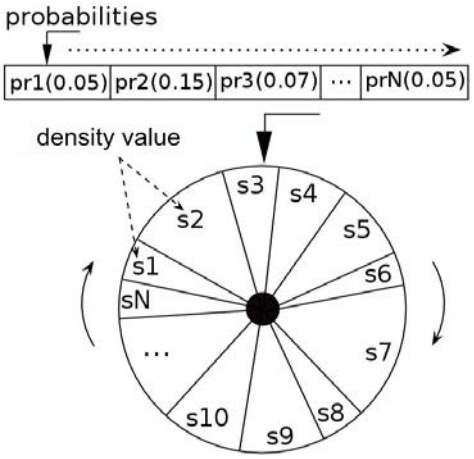


FIGURE 4.3: Roulette wheel random selection. Each pixel gets a portion of the wheel according to its density value.

(a) (b) (c)

FIGURE 4.4: A sampling example by roulette wheel extraction in the structured meshing application: (a) input disparity map, (b) density sampling by roulette wheel extraction, (c) corresponding mesh result.

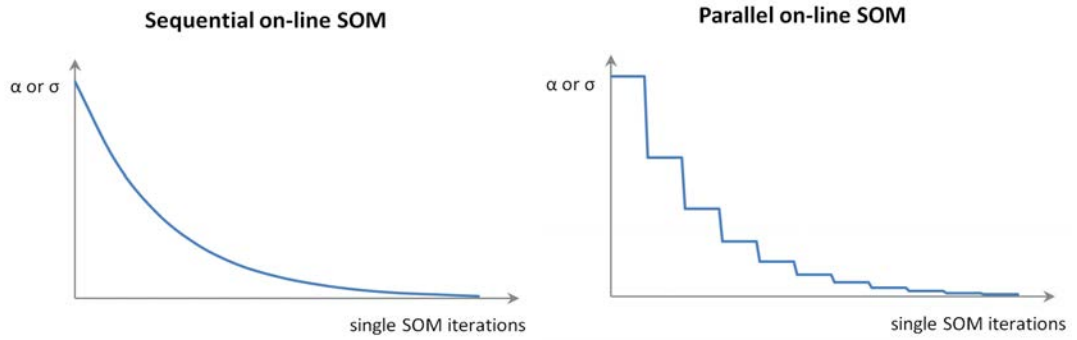
4.3.1.2 On-Line SOM Kernel

Each GPU thread performs the basic on-line SOM process independently in parallel, through the on-line SOM kernel detailed in Kernel 8. It corresponds to an instantiation of the generic projection loop.

The $FGetRandom.initialize$ function is used to set the number of times (m_i) that the function $FGetRandom.get$ will be executed, i.e. the number of iterations of the loop defined in lines 3—8. Each time the function $FGetRandom.get$ firstly checks if $cell_i$ is activated for the current extraction step, then, it randomly selects an input data point (from the matched grid G_2) that locates in $cell_i$ if it is activated. For the extracted point, the spiral search is carried out by the $FSpiralSearch.search$ function, in order to find the closest cluster center (in the matcher grid G_1). The learning operation in the triggering step is done by the $FTrigger.operate$ function on the cluster center grid (the matcher grid G_1). The function $FGetRandom.next$ returns false state after m_i times executions of the function $FGetRandom.get$.

Kernel 8: On-line SOM iteration loop.

```
// For each cell_i of cellular matrix WD × HD, do:
Input: G_1, G_2, cell_i, FGetRandom, FSpiralSearch, FTrigger
1 begin
2     FGetRandom.initialize(cell_i);
3     while FGetRandom.next(cell_i) do
4         point ← FGetRandom.get(cell_i, G_2);
5         if point then
6             closest ← FSpiralSearch.search(cell_i, G_1, point);
7             if closest then
8                 FTrigger.operate(cell_i, G_1, point, closest);
9     return
```



(a) sequential on-line SOM (b) parallel on-line SOM

FIGURE 4.5: Parameter decreasing behaviors of sequential on-line SOM and parallel on-line SOM.

Note that the learning rate α and radius σ are decreased during the external parallel iterations, as shown at step 8 in Figure 4.2. For a given cell, it extracts at most m_i points in one execution of Kernel 8, performing m_i single SOM iterations with same learning rate α and radius σ. Then for the t_{max} external parallel iterations, the maximum number of single SOM iterations is $t_{max} \times WD \times HD \times m_i$, with the learning rate α and radius σ decreasing in a stair-stepping way, as illustrated in Figure 4.5 (b), in contrast to the parameter decreasing behavior of sequential on-line SOM as illustrated in Figure 4.5 (a).

All the cluster center locations are stored in GPU global memory that is accessible to all the threads. Like all the multi-threaded applications, different threads may try to modify a same cluster center location at the same time, which causes *race conditions*. In order to guarantee a coherent memory update in this situation, we use the CUDA atomic function which performs a read-modify-write atomic operation without interference from any other threads [NVI12]. The atomic operation only concerns the triggering step.

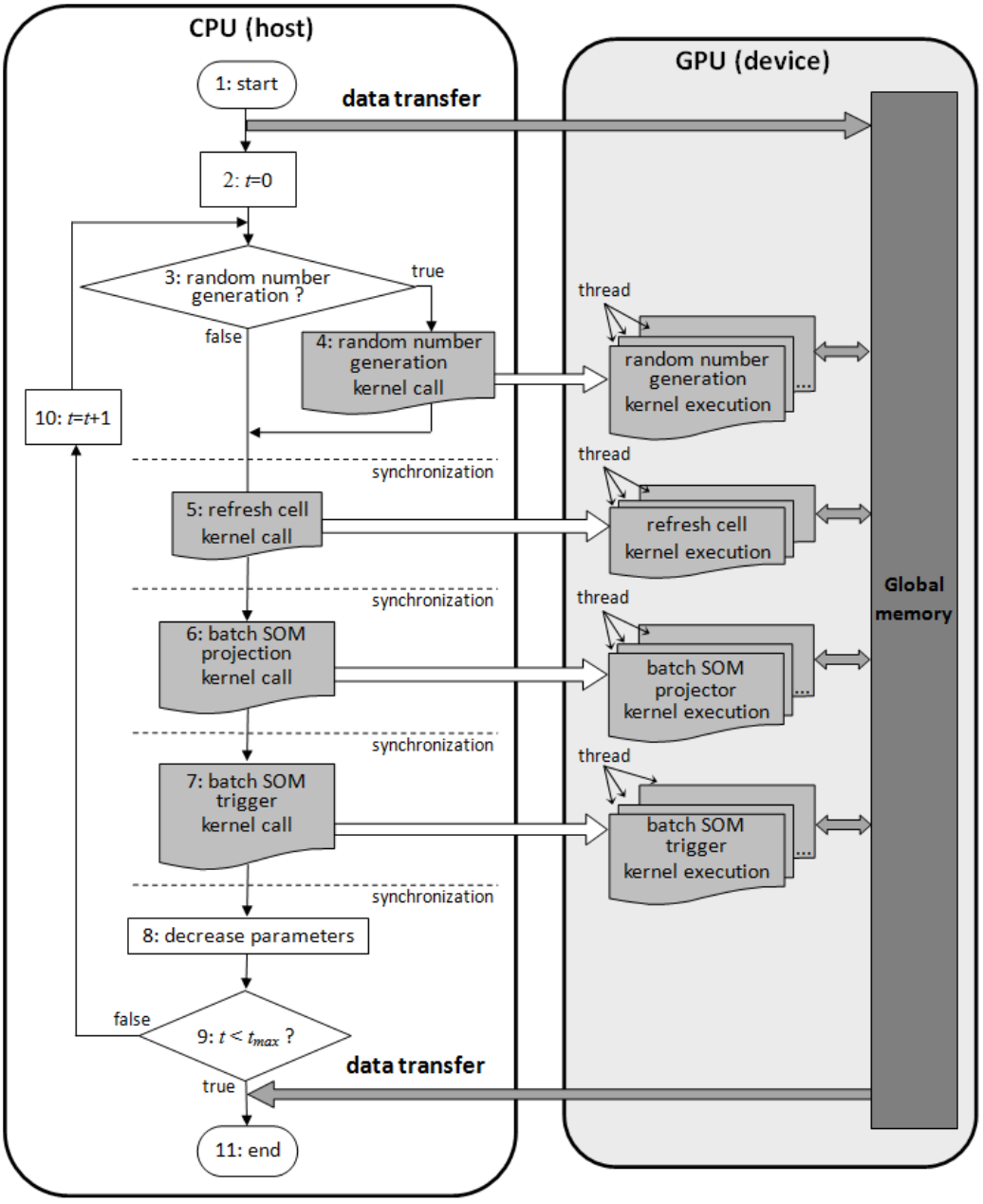


FIGURE 4.6: Flowchart of parallel batch SOM.

4.3.2 Parallel Batch SOM

In Figure 4.6 is reported the flowchart of the kernel calls sequence from the CPU side of the parallel batch SOM implementation. The difference between on-line SOM and batch SOM is that for the former the data point arrives on-line, whereas all the data points are known in advance for the latter. Then, the batch SOM procedure can be decomposed into two main steps. The first step performs projection to the cluster centers of all the data points. The second step performs the triggering based on the average cluster

center locations. To instantiate the parallel program, we duplicate the generic loop template into two parts as detailed in Kernel 9 and Kernel 10.

Kernel 9: Batch SOM projector loop.

```
// For each cell_i of cellular matrix WD × HD, do:
Input: G_1, G_2, cell_i, FGet, FSpiralSearchComputeAverage
1 begin
2     FGet.initialize(cell_i);
3     while FGet.next(cell_i) do
4         point ← FGet.get(cell_i, G_2);
5         if point then
6             FSpiralSearchComputeAverage.search(cell_i, G_1, point);
7     return
```

Kernel 10: Batch SOM trigger loop.

```
// For each cell_i of cellular matrix WD × HD, do:
Input: G_1, G_2, cell_i, FGet, FTrigger
1 begin
2     FGet.initialize(cell_i);
3     while FGet.next(cell_i) do
4         neuron ← FGet.get(cell_i, G_1);
5         if neuron then
6             FTrigger.operate(neuron);
7     return
```

In Kernel 9, the functor $FGet$ is used to get one by one all the input data points (from the matched grid G_2) that locate in $cell_i$. For every point, the function $FSpiralSearchComputeAverage.search$ firstly performs the spiral search on the cluster center gird (the matcher grid G_1) to find the closest cluster center, let us say c_j. Then, the function adds the point to the closest point list of c_j and recomputes the new weight of c_j according to the updated list. Thus, the expected weight of each neuron is computed and recorded in an incremental way during the batch SOM projector loop. Note that these expected weights are not taken until in the batch SOM trigger loop of Kernel 10, which is assigned to each cell right after the batch SOM projector loop, at each of the t_{max} external parallel iterations, as illustrated in Figure 4.6.

In Kernel 10, the functor $FGet$ is used to get one by one all the cluster centers (from the matcher grid G_1) that locate in $cell_i$. For every cluster center, the function $FTrigger.operate$ updates its weight along with the related neighbors, according to the new weight computed by $FSpiralSearchComputeAverage.search$ in Kernel 9. At each of the t_{max} external parallel iterations of the parallel batch SOM, the batch SOM

projector loop and the batch SOM trigger loop are executed sequentially, as illustrated in Figure 4.6.

4.4 Superpixel Adaptive Segmentation Map (SPASM)

As building blocks of many vision algorithms, superpixels divide raw image into perceptually meaningful atomic regions which can be employed to substitute the rigid structure of the pixel grid [ASS+12, RM03, ME07]. Therefore, these atomic regions should represent or reflect some local properties with respect to the attributes distributions of the raw image. However, most of the existing algorithms produce uniformly distributed superpixels. We propose the concept of *superpixel adaptive segmentation map* (SPASM), aiming to generate an adaptive segmentation result where the distribution/density of superpixels coincides with the distribution of some specified attribute of input image, such as edges, textures, and depths. The corresponding algorithm, called the SPASM algorithm, is a superpixel segmentation algorithm. By the word "adaptive" we mean in the final segmentation map of input image, (1) the distribution (density) of superpixels is adaptive and (2) the size of superpixel is adaptive. In order to achieve such goal, we add a learning phase for fast density "projection" by the parallel on-line SOM, before the *k*-means based segmentation by the parallel batch SOM. The density projection is very similar to the structured meshing application, but with various input choices that are not only restricted to the disparity map.

The SPASM could be interesting to vision applications based on superpixel representation. For example, in the visual correspondence computation tasks that employ superpixels as basic matching units [ZK07, TWZ08, GG15, KT15, MG15], it should be beneficial to have more superpixels in the areas with dense edge distribution because these areas tend to contain various objects and they need to be matched with finer superpixels for a higher precision. On the contrary, the areas with sparse edge distribution are more likely to contain single object or background, and hence they should need fewer superpixels for fast computation.

4.4.1 Two-Phase *K*-means

The process of the SPASM algorithm is illustrated in Figure 4.7. It consists in applying the parallel SOM *k*-means algorithm in two phases. The first phase consists of the parallel on-line SOM procedure, similar to the structured meshing, in order

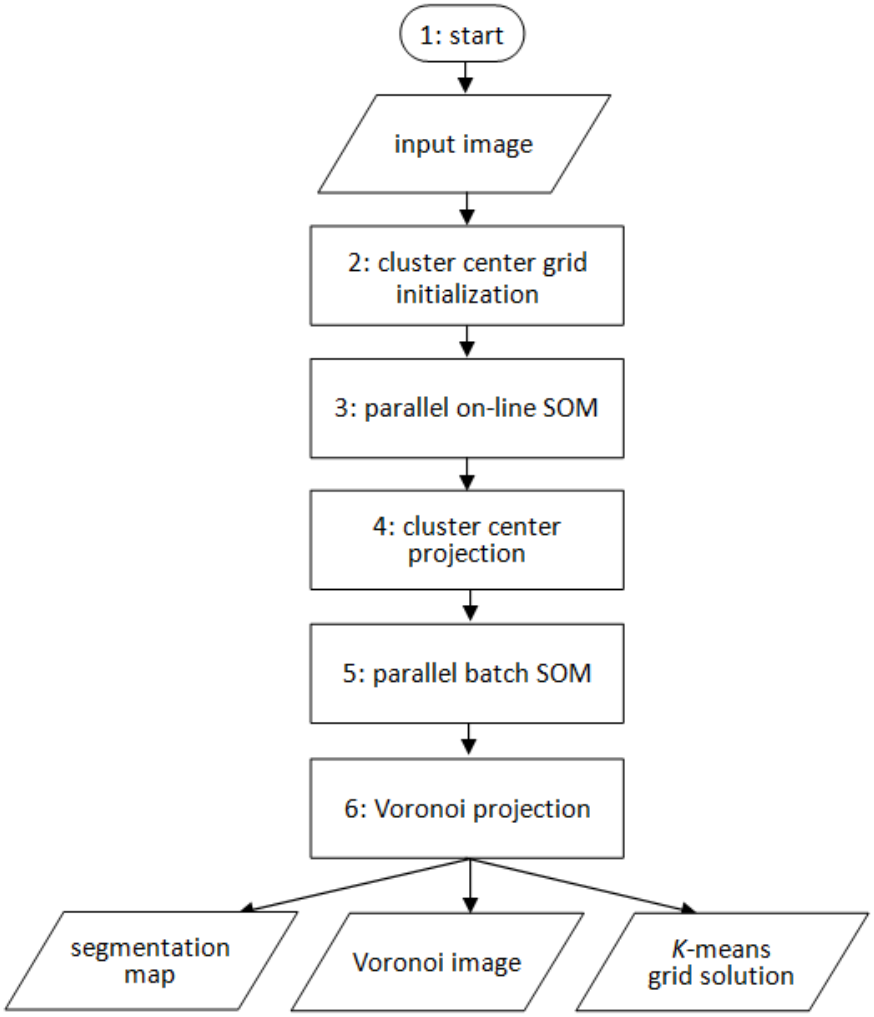


FIGURE 4.7: Flowchart of the SPASM algorithm.

for the cluster centers to reflect the underlying distribution of some image attribute. This first phase is intended to give a cluster center initialization in order for the parallel batch SOM to be applied with color space composite distance as a second phase of superpixel segmentation. The parallel on-line SOM could be viewed as a construction phase where the cluster centers are further "initialized" from the uniform distribution to our specified distribution; while the parallel batch SOM could be viewed as an improvement phase where the *k*-means clustering is performed based on the specifically deployed cluster centers.

During the parallel on-line SOM, as indicated by Step 3 in the flowchart of Figure 4.7, the Euclidean distance is used, and the cluster centers only change their positions (2D coordinates) in the plane defined by the input image. Once the parallel on-line SOM learning is finished, a cluster center projection procedure is carried out, as indicated by Step 4 in the flowchart, where each cluster center searches its closest non-edge pixel with spiral search, and copies all attributes (coordinate, color, density value) from

the closest pixel to the cluster center. Note that the direction of this projection is from the cluster center grid (the matcher grid) to the input image (the matched grid), which is opposite to the direction of the basic *k*-means projection as illustrated in Figure 4.1. After the projection, cluster centers are ready for the next step of parallel batch SOM clustering.

During the parallel batch SOM, as indicated by Step 5 in the flowchart, the distance measure between two vertices i, j consists of spatial proximity, color, and density value, as defined in

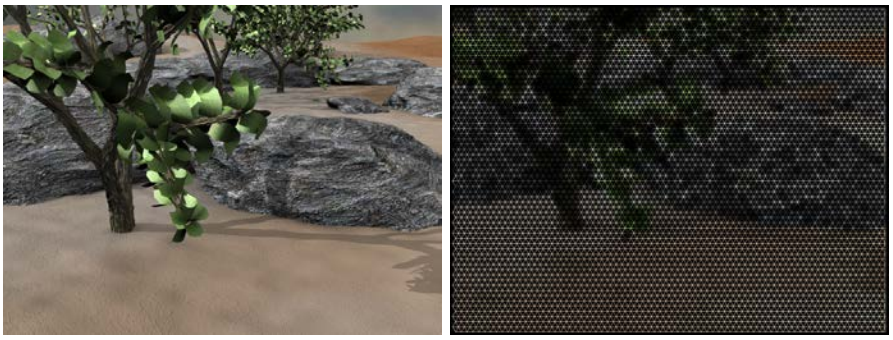
$$D(i,j) = \tau_s \|\boldsymbol{X}_{spa}(i) - \boldsymbol{X}_{spa}(j)\| + \tau_c \|\boldsymbol{X}_{rgb}(i) - \boldsymbol{X}_{rgb}(j)\| + \tau_d \|\boldsymbol{X}_{den}(i) - \boldsymbol{X}_{den}(j)\|, \quad (4.4)$$

where $\boldsymbol{X}_{spa}$, $\boldsymbol{X}_{rgb}$, and $\boldsymbol{X}_{den}$ respectively correspond to 2D coordinate, 3D *RGB* color, and 1D density value, while τ_s, τ_c, and τ_d are their corresponding normalized coefficients. Note that this distance measure is an extension to the SLIC distance measure [ASS+12] with a third component of density. Therefore, the SPASM algorithm should have the same ability of edge/boundary adherence as the SLIC algorithm, meanwhile considering the density distribution of some attribute for distance computation between points and cluster centers. At the triggering step of batch SOM, cluster centers update the three component attributes—2D coordinate, 3D *RGB* color, and 1D density value—according to the learning law of Equation 2.6.

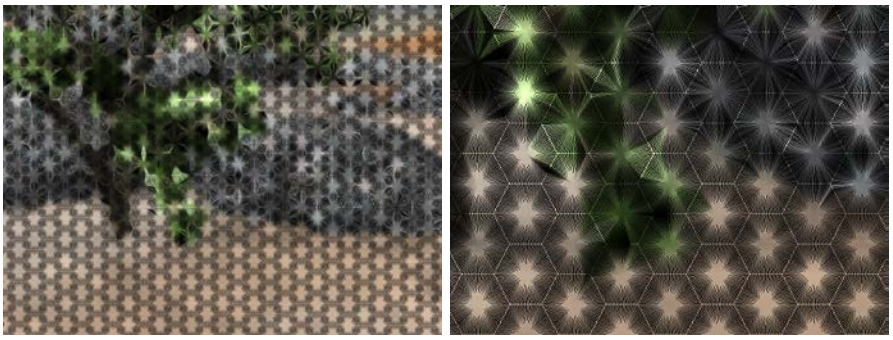
After the parallel batch SOM clustering process is finished, the parallel Voronoi projection procedure is carried out, as indicated by Step 6 in the flowchart, where each pixel is assigned to its closest cluster center, with spiral search using the distance measure of Equation 4.4. Then, a SPASM image is obtained where each cluster center forms a superpixel with the pixels assigned to it. Moreover, we also generate a Voronoi color image where the color of each pixel is filled by the color of its cluster center according to the superpixel composite distance.

4.4.2 Implementation in Cellular Matrix Model

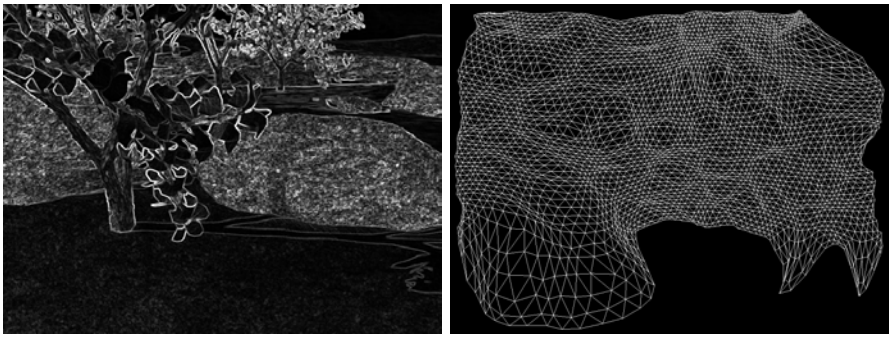
An example of the SPASM algorithm in the cellular matrix model is illustrated in Figure 4.8. The input image *Grove2* (640×480) [BSL+11, Mid15a], as shown in (a), along with the two-dimensional grid of superpixel cluster centers which is deployed in the Euclidean plane defined by the input image, as shown in (b), are partitioned into uniformly sized cells with rigid topologies, as shown in (c) and (d). Note that in this example, both the cellular matrix and the grid of cluster centers are with hexagonal topology. However, they can be other topologies, such as quad and rhombus, in



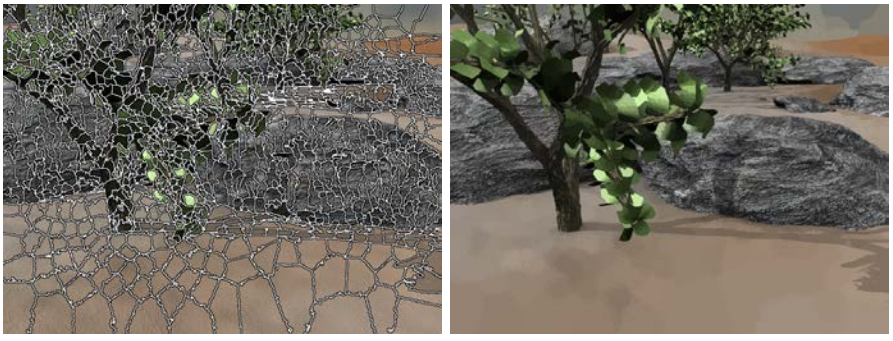
(a) input image (b) initial cluster center grid



(c) cellular matrix covering illustration (d) zoom in of (c)



(e) image gradient (f) on-line SOM training result



(g) superpixel segmentation (h) Voronoi image

FIGURE 4.8: An example of the SPASM algorithm in the cellular matrix model. Note that in (g) the superpixel contours are drawn after a post-processing step applied to the raw segmentation result where isolated pixels, which belong to very small superpixels, are forcedly assigned to nearest bigger superpixels, for the sake of a better visualization; while in (h) the Voronoi image is generated from the raw segmentation result. The same situation applies to all the other examples demonstrated in this thesis.

the cellular matrix model. According to some underlying attribute distribution of input image, such as the image gradient used in this example and shown in (e), massively parallel on-line SOM training processes are carried out, to deploy superpixel cluster centers. In the training result as shown in (f), the distribution of cluster centers reflects the image gradient distribution, with respect to scene topology. Based on the training result, the batch SOM is employed for the final superpixel segmentation result as shown in (g), where the density of generated superpixels reflects the image gradient distribution. In the corresponding Voronoi image as shown in (h), regions with higher gradient values are more finely represented with more superpixels for detail preservation.

Implemented based on the cellular matrix model, the SPASM application, and the structured meshing application which can be seen as a sub-procedure of the SPASM application, can benefit from the massive parallelism property of the cellular matrix model. Given an input image or a disparity map with $W \times H$ pixels, a Euclidean plane is defined. This corresponds to the low level of the cellular matrix model. Normally, an original image or disparity map can be viewed as a quad[1] topological grid of input data points. However, it could be easily transfered into a hexagonal grid by shifting the horizontal coordinates of pixels in odd lines, as shown in Figure 3.3 and Figure 3.4. Then, a two-dimensional SOM neural network will be deployed at the zoom-out level, with the size of $WZ \times HZ = W/RC \times H/RC$. The size is in relation to the size of the input image in such a way that the mesh constitutes a compressed representation of the input image. The compression rate is $1/RC^2$. In the SPASM application, the network is employed by both the on-line SOM learning and the batch SOM clustering, while each neuron represents a cluster center which will be used to generate a superpixel at the end of the algorithm. The dual level of the cellular decomposition is with size $WD \times HD = W/2R \times H/2R$, where R is a constant factor to control the degree of parallelism. Note that in a concrete application, R is not necessarily equals to RC, which is the default parameter setting as illustrated in Figure 3.3 and Figure 3.4. The cellular matrix decomposition is in linear relationship to the input disparity map/image size, in $O(W \times H)$.

4.5 Conclusion

In this chapter, we have presented topological *k*-means problems using our generic formulation of grid matching, and we have provided the related parallel *k*-means

[1]The tessellation of input data points at low level is the same in quad cellular matrix model and in rhombus cellular matrix model.

procedures, in the cellular matrix model. We have illustrated how the parallel SOM algorithm, both the on-line version and the batch version, can be implemented by combination of basic projection functions and functors.

We have proposed the SPASM algorithm, as a superpixel image segmentation algorithm, which combines the parallel on-line SOM structured mesh generation and the parallel batch SOM clustering. The aims is to generate the segmentation where the distribution (density) of superpixels coincides with the distribution of some specified attribute of input image, such as edges, textures, and depths. Normally, the segmentation result of classical *k*-means clustering is sensitive to the cluster center initialization and distribution, and that is the reason we add the parallel on-line SOM to deploy cluster centers along with the density distribution, before the *k*-means based batch SOM clustering. The on-line SOM could be viewed as a construction phase where the cluster centers are further "initialized" from the uniform distribution to our desired distribution; while the batch SOM could be viewed as an improvement phase where the *k*-means clustering is performed based on the specifically deployed cluster centers. Batch SOM deals with positioning of cluster centers in homogeneous superpixel regions by using a specific space-color-density distance, whereas on-line SOM only deals with Euclidean distance. These two phases are all necessary: if we remove the on-line SOM, then the superpixel distribution will be only decided by the strict cluster center initialization, such as the uniform distribution used in the *k*-means based SLIC algorithm; if we remove the batch SOM, then we can only get a rough segmentation result based on the on-line SOM deployed cluster centers, eliminating the chance of well-segmented results with good edge/boundary adherence, which are supposed to be obtained by our batch SOM clustering using the true closest point finding of spiral search in the cellular matrix model, with the required space-color-density composite distance measure.

Experiments reported in the next chapter will allow us to evaluate the computation gain of the parallel SOM approach on the three applications: TSP, structured meshing, and SPASM.

Chapter 5

Experimental Study of Parallel Topological *K*-means

5.1 Introduction

This chapter focuses on GPU implementations of the three parallel SOM applications: TSP, structured meshing, and SPASM. For each application, we perform experimental analyses and carry out comparative studies between our GPU parallel SOM approach and other approaches. We evaluate our GPU implementation of the parallel SOM TSP application, on different large-size TSP instances from different benchmarks, including the largest TSPLIB [Rei91] instance with 85900 cities and the largest National TSP's [nat09] with 71009 cities. We experiment on our GPU implementation of the structured meshing application and its counterpart CPU implementation, using input images with different sizes and analyzing the relationship between execution time and input image size. We test our SPASM algorithm utilizing two kinds of image attributes as the density distributions for cluster center initialization with the on-line SOM: image gradient and disparity value. We compare the SPASM algorithm with the state-of-the-art SLIC algorithm [ASS+12], on input images of growing sizes, setting different initial superpixel sizes. The comparison is carried out between both GPU version and CPU version for both two algorithms.

The experimental studies are conducted on the following platforms:

- *On the CPU side:* An Intel Core i5-750 processor running at 2.67 GHz and endowed with four cores and 4 GBytes memory. It is worth noting that only one single core executes the SOM process in our implementation.

- *On the GPU side:* A Nvidia GeForce GTX 570 Fermi graphics card endowed with 480 CUDA cores (15 streaming multi-processors with 32 CUDA cores each) and 1280 Mbytes memory. The compute capability is 2.0.

5.2 Basic On-Line SOM Applications

For all the experiments of basic on-line SOM applications, the default values of four SOM parameters are set as follows: $(\alpha_{init}, \alpha_{final}, \sigma_{init}, \sigma_{final}) = (1, 0.01, 24, 1)$. Note that parameter values for SOM are standard values: they were adjusted after a preliminary round of experiments which are not reported in this thesis. Also, note that these parameter values may change from the default values for various experimental purposes in our experimentation, and the change will be mentioned when happens.

To speed up the program, during the parallel spiral search of each thread, we introduce a *spiral search range* (SSR) parameter to control the search scope. Then, the maximum number of cells a thread would search equals $(SSR \times 2 + 1)^2$. Note that the original spiral search corresponds to setting SSR infinite.

5.2.1 Large-Size TSP Instances

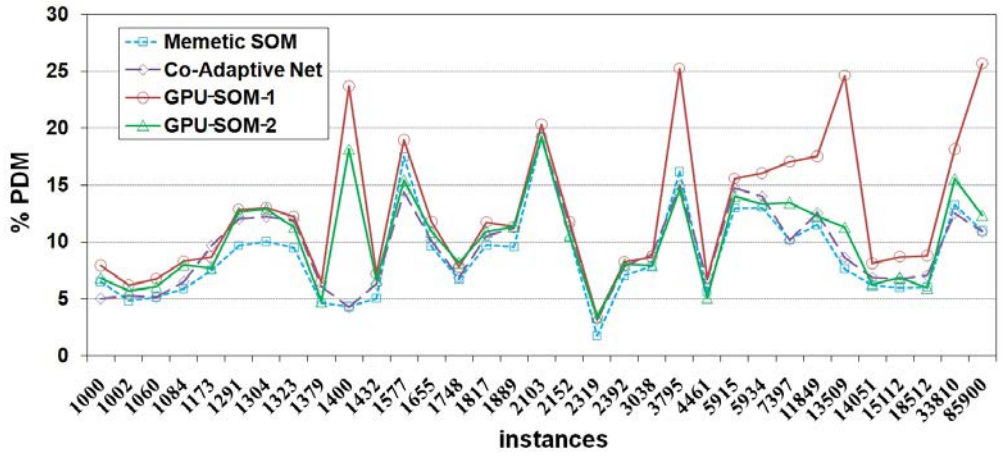
As the cellular matrix model is designed for large-scale optimization problems, we evaluate our GPU implementation of the parallel SOM TSP application, on different large-size TSP instances from different benchmarks. These instances include 33 TSPLIB [Rei91] instances with sizes from 1000 cities to 85900 cities and 19 National TSP's [nat09] with sizes from 1621 cities to 71009 cities.

When dealing with the number of iterations to achieve good trade-offs between result quality and execution time, a first remark is about the ability of the method to, **either** act as a construction method, since the ring deploys from random initialization; **or** act as an improvement method when the algorithm has reached the asymptotic phase in which small intensity moves massively adapt the ring to the data. It appears that dissociating a construction phase with large initial neighborhood from an improvement phase with small initial neighborhood, yields best performance. We adopt this scheme here. The only parameter that distinguishes construction phase with improvement phase is the neighborhood radius parameter σ_{init}, which defines the window size around the winner neuron. Then, the number of parallel iterations must be set accordingly: it should be small in construction phase, just enough to deploy the network. For the construction phase, the neighborhood radius σ_{init} is proportional to the instance size, being set to

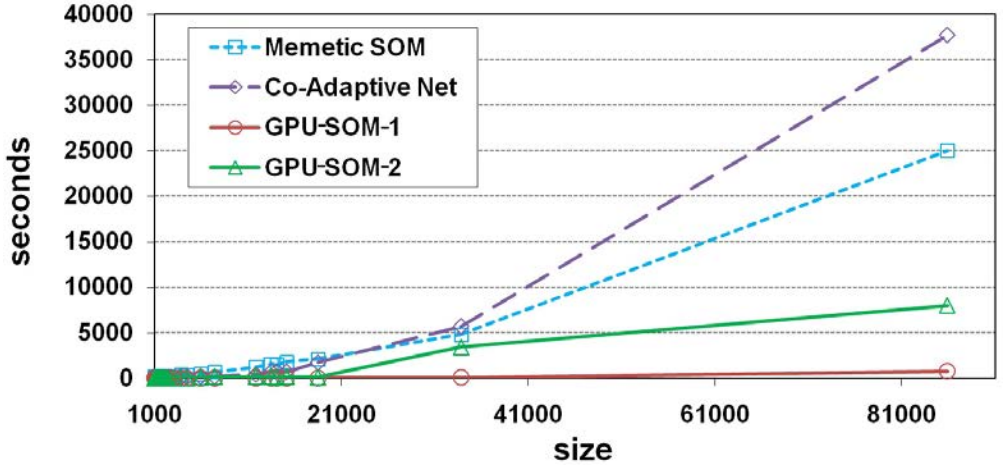
$min\{instance\_size/10, 500\}$. Thus, local moves are performed with high intensity to let the solution ring deploy from scratch. The subsequent improvement phase can be seen as a continuation of the construction phase using a smaller intensity for moves. In that case, the neighborhood radius parameter σ_{init} is set constant to a value of 10 neurons on each side of the winner neuron.

For the 52 tested TSP instances, the number of iterations is set as follows: 100 iterations in the construction phase with the neighborhood radius σ_{init} set to $min\{instance\_size/10, 500\}$, plus 10000 iterations in the improvement phase with σ_{init} set to 10. The SSR parameter is set to 10 and *infinite* respectively in two implementation versions, namely "GPU-SOM-1" and "GPU-SOM-2". Executions with two SSRs will allow us to explore different trade-offs between execution time and result quality. The results obtained from 33 TSPLIB instances and 19 National instances are given in Table A.1, Table A.2, and Table A.3 in Appendix A. In each table, the first column reports the names of the instances to which their sizes are concatenated. Instances are ordered by size. The second column reports the optimum tour lengths. Other columns present results for the different approaches considered. All the results are obtained over 10 runs. The percentage deviation from the optimum of the mean solution value over the 10 runs is reported in column "%PDM". The percentage deviation from the optimum of the best solution value that was found is reported in column "%PDB". The average execution time per run for each instance is reported in column "Sec" in seconds. The last two rows of the tables report the average results of all instances and the average results of the instances with the number of cities $N \geq$10000. Besides our two GPU implementations GPU-SOM-1 and GPU-SOM-2, in the last three columns, we also report the results obtained by the *memetic SOM* algorithm (CPU implementation) which is, as claimed in [CK09], the winner of the neural network approaches to the TSP, with respect to solution quality and/or execution time.

In Figure 5.1 are plotted the results obtained from the 33 large-size TSPLIB instances, where Figure 5.1 (a) shows "%PDM" values according to the instances ordered by size, and Figure 5.1 (b) shows execution time as a function of the instance size. Besides our two GPU implementation versions of parallel SOM TSP, in Figure 5.1 we also report the results of another two dominant neural network methods—*memetic SOM* [CK09] and *Co-Adaptive Net* [CB03]—for comparison. It is worth noting that these two approaches are the only neural network approaches that have performed extensive evaluations on such large size well-known TSP instances. Our GPU SOM implementations run a lot faster than the CPU memetic SOM and Co-Adaptive Net. In spite of the absolute execution time results obtained from different implementations on different platforms (CPU and GPU), we think what is more important is the relationship between execution time and instance size, as depicted in Figure 5.1 (b). We can note that the execution



(a) percentage deviation of the mean solution value over 10 runs



(b) execution time according to instance size

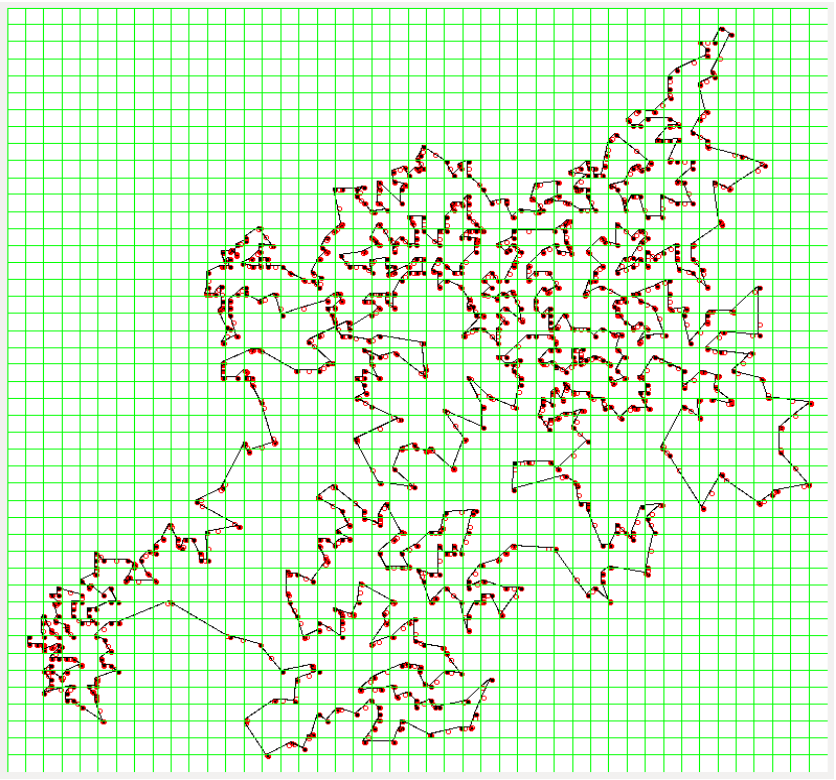
FIGURE 5.1: Experimental results of 33 TSPLIB instances.

time of our GPU implementation increases very slowly compared to either memetic SOM or Co-Adaptive Net, when the input instance size augments.

In Figure 5.2 is given an example of TSP tour obtained by our GPU SOM implementation, using the *rw1621* TSP instance from National TSP's [nat09]. The 1621 cities of the instance are deployed in a rectangle plane, as shown in Figure 5.2 (a). The plane is partitioned into 45×45 squares by our cellular matrix model with quad topology, as shown in Figure 5.2 (b) where each green square represents a cell and is handled by a GPU thread. A TSP tour is obtained along the SOM ring network of vertices/neurons (small red circles) and edges (black lines) where each city (black dot) is mapped onto its nearest vertex in the plane.

In Figure 5.3 are depicted the relationships between execution time and result quality, of different state-of-the-art methods on the 33 large-size TSPLIB instances in average. Co-Adaptive Net, Memetic SOM, GPU-SOM-1 and GPU-SOM-2 are neural

(a) *rw1621* TSP instance



(b) tour result

FIGURE 5.2: An example of TSP tour obtained by our GPU SOM implementation, using the *rw1621* TSP instance from National TSP's [nat09]. Black dots represent cities; small red circles represent vertices/neurons of the SOM ring network; green lines define the cellular matrix decomposition on the dual level, where each green square represents a cell and is handled by a GPU thread.

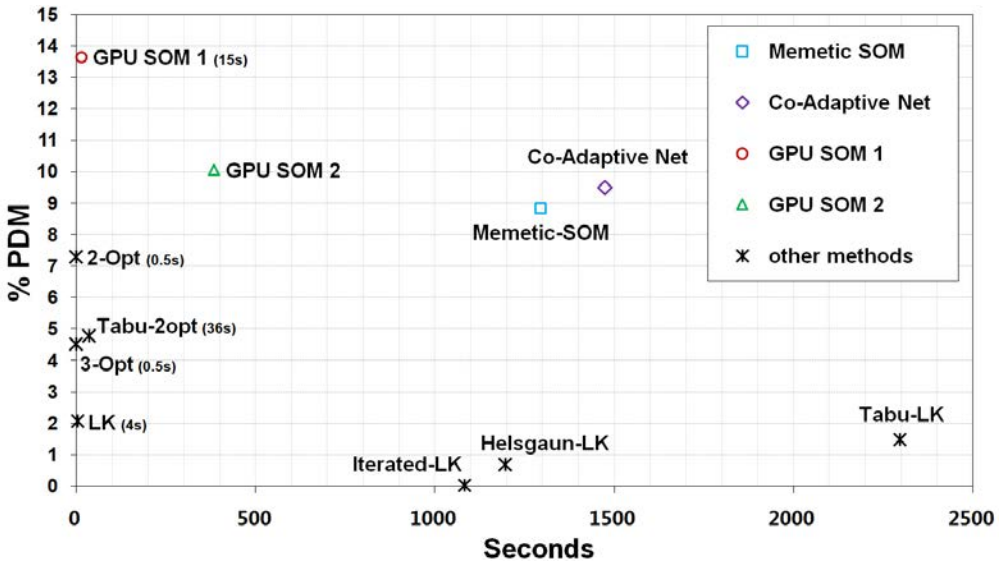


FIGURE 5.3: The relationship between execution time and result quality, with 33 TSPLIB instances in average, compared with state-of-the-art operations research (OR) heuristics.

network methods while others are very sophisticated implementations of standard heuristics such as 2-*opt*, 3-*opt*, or *Lin-Kernighan* (LK) as described in DIMACS TSP Challenge [JM07, DIM09]. Certainly, neural networks do not compete with state-of-the-art powerful heuristics of operations research (OR) for the TSP. To be competitive, the solution quality produced by our GPU SOM implementation would have to be improved by a factor ten. In order to further improve result quality, it is worth trying to add other operators rather than only employ the standard on-line SOM. It can be envisaged to extend the approach by using evolutionary operators and population based search as in memetic SOM, or improve SOM learning procedures as in co-adaptive net, or simply mix the method by using standard neighborhood search operators such as k-opt operator. In the last case, the algorithm could be configured as a distributed local search (similar method to our proposed DLS introduced in Chapter 6).

5.2.2 Structured Meshing Results with Different Disparity Maps

We experiment on our GPU implementation of the structured meshing application as defined in Chapter 2 and its counterpart CPU implementation. Here, the CPU sequential version of implementation is a simple simulation of the GPU parallel version. In the CPU implementation, all operations are carried out on the CPU side sequentially.

We firstly perform experiments using six small size disparity maps (*Tsukuba, Rocks1, Aloe, Venus, Teddy, Cones*) from the Middlebury datasets [Mid15b], with a smaller spiral search range (SSR=3). Detailed results are reported in the first set (the first six rows) of

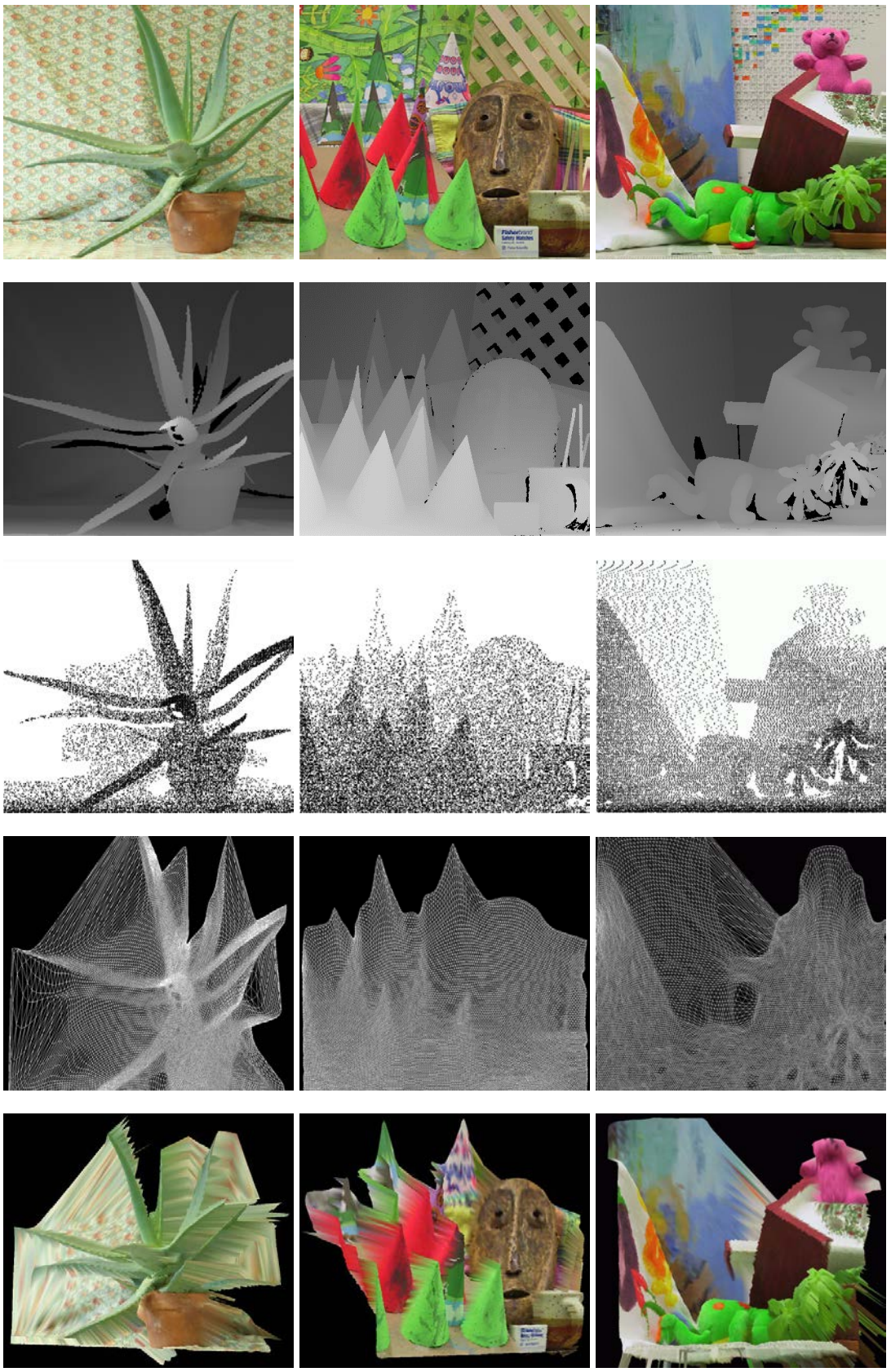


FIGURE 5.4: Structured meshing application. From left to right: *Aloe*, *Cones*, *Teddy*. From top to bottom: color image, disparity map, density sampling, meshing, 3D visualization. The used images are from Middlebury datasets [Mid15b].

Table A.4 in Appendix A. Note that the number of iterations for CPU version is set to its corresponding GPU version value multiplied by *cellular matrix size*[1], in order to make the total SOM operations approximately similar between the two versions, and to reach similar result qualities. Here, the result quality is measured by the $\%cost$ value defined in Equation 2.5 in Subsection 2.2.1.2, which is the average percentage deviation of each individual honeycomb cell weight to the average honeycomb cell weight. All the tests are done on a basis of 10 runs per input image, and the test results are reported in the right four columns in Table A.4. The execution time of GPU version for all six tested images is steady and no more than 0.36 seconds. The acceleration factor, which is the ratio of CPU version's execution time by GPU, varies from factor 5.84 (*Tsukuba*) to factor 8.33 (*Aloe*). Experimental results show that GPU version of the structured mesh generation with small size disparity maps can provide near real-time performance, with very similar result quality in average, compared with its counterpart sequential CPU version. We also study the relationship between execution time and input size of GPU implementation and its counterpart CPU version, by carrying out experiments with four disparity maps, each at small, medium and large scales, respectively. Inputs, parameter settings and results of mean values from 10 runs are all reported in the last three sets of Table A.4 in Appendix A. The average acceleration factors of the three sets are 5.49, 12.68, and 39.74 respectively, as input size grows, which indicate augmentation of acceleration factor with input size.

Figure 5.4 displays snapshots of our structured meshing application dealing with *Aloe*, *Cones*, and *Teddy* benchmarks, respectively. In the first row are the color stereo images from left view, and in the second row are their corresponding disparity maps. In the third row are samplings of disparity maps obtained by extracting 10000 points with the roulette wheel mechanism. This sample has been obtained after removing the background small density values from the disparity map, and after augmenting the contrast in it. Then, objects that are close to the camera have higher density values. In the fourth row are the adaptive 2D mesh results obtained by SOM algorithm with the disparity maps. In the second row, brighter regions are nearer to the camera view point, and in the fourth row, the adapted grid presents higher density of network vertices in such regions, with respect of the topology of the scene. The fifth row shows the surface reconstruction in 3D space obtained by using the adapted mesh which can be seen as a compressed representation of the 3D surface, such that objects close to the camera have higher resolution and their details are more finely represented.

[1]The cellular matrix size is the number of cells in the cellular matrix model. Note that this setting for CPU version corresponds to the extreme situation of GPU version where all cells in the cellular matrix model are activated during each external iteration.

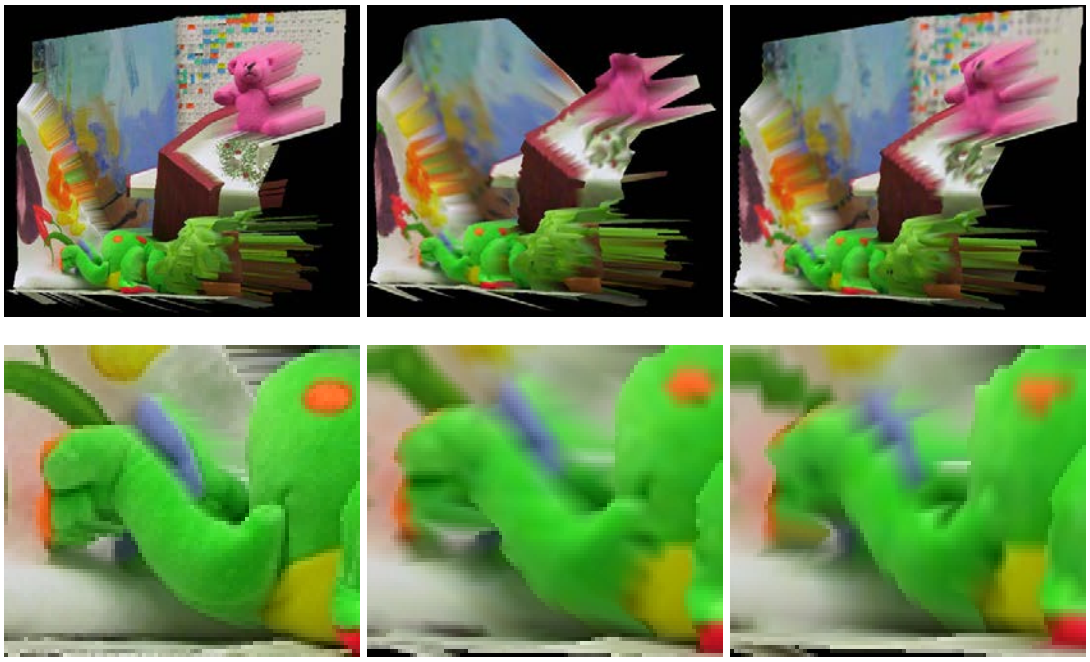


FIGURE 5.5: An example of higher resolution for closer objects. From left to right: uniform mesh with high resolution (450×375), adapted mesh (75×63), uniform mesh (75×63). First row: global scene. Second row: zoom-in on a part of a closest object of the scene.

Figure 5.5 shows an example of higher resolution for closer objects, where different visualizations of 3D reconstructions are respectively from high resolution uniform mesh of original image; adapted mesh of low resolution with a compression rate of 36; and uniform mesh of low resolution with a compression rate of 36. The zoom-in comparison illustrates the resolution around a closest object. The adapted mesh presents a higher resolution for the closest object in the scene than the uniform reduced mesh.

5.3 Experimental Results of SPASM Application

In our experiments of the SPASM application, the parallel on-line SOM parameters are fixed as $(\alpha_{init}, \alpha_{final}, \sigma_{init}, \sigma_{final}, t_{max}) = (1, 0.01, 20, 0.5, 100)$, while the parallel batch SOM parameters are fixed as $(\alpha_{init}, \alpha_{final}, \sigma_{init}, \sigma_{final}, t_{max}) = (1, 0.1, 1.5, 0.5, 5)$. Note that parameter values for SOM are standard values: they were adjusted after a preliminary round of experiments. For the number of iterations, we set a relatively small value for both the parallel on-line SOM (t_{max}=100) and the parallel batch SOM (t_{max}=5), in order for fast computation. Note that both the two processes are indispensable for SPASM generation: if we remove the on-line SOM (set parallel on-line SOM t_{max}=0), then the superpixel distribution will only be decided by the strict cluster center initialization, which in our application is a regular grid with a uniform distribution of

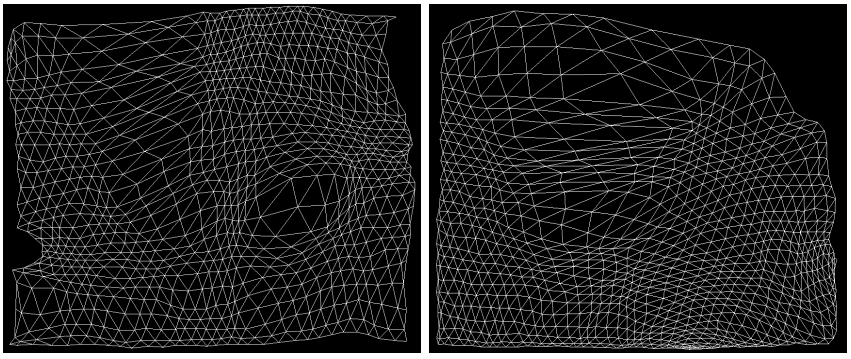
cluster centers; if we remove the batch SOM (set parallel batch SOM t_{max}=0), then we can only get a rough segmentation result based on the on-line SOM deployed cluster centers, eliminating the chance of well-segmented results with good edge/boundary adherence, which are supposed to be obtained by our topological *k*-means clustering using the true closest point finding of spiral search in the cellular matrix model, with the required space-color-density composite distance measure.

5.3.1 Different Image Attributes for Cluster Center Initialization

We utilize two kinds of image attributes as the density distributions for cluster center initialization with the on-line SOM. The first attribute is image gradient. In this case, before running the algorithm, we initialize an input density map with gradient values. In the point extraction step, we transfer the gradient g of each pixel into $1/(1+g^2)$ for the local roulette wheel extraction. The reason is to make edge points (with high gradient values) less likely to be extracted. Therefore, the image gradient distribution is preserved on the cellular matrix level, meanwhile inside activated cells, situations of cluster centers being moved onto edge points (pixels) are reduced. We compute image gradient through *Sobel* operator which gives us a fast approximation of the edge distribution of input image, as shown in Figure 5.6 (a). The activation probability of each cell is computed according to the sum of gradient values of all the pixels inside the cell. The second attribute is disparity value that is supposed to be known in advance for the input image. The disparity map reflects the proximity of objects to the camera view point, as presented in Figure 5.6 (b).

The other images in the two columns of Figure 5.6 respectively present the cluster center grids after on-line SOM (c) and (d); superpixel maps after the batch SOM phase (e) and (f); and the corresponding Voronoi maps (g) and (h) with Voronoi uniform cell colors identical to their cluster center color components. Note that the Voronoi maps are generated according to the space-color-density composite distance, as defined in Equation 4.4. These results can be appreciated visually. As shown in (c) and (d), in the adapted grid after on-line SOM learning phase, areas with high image gradient or disparity value present high density of network vertices (cluster centers). Then, these areas generate more superpixels after the batch SOM phase, as shown in (e) and (f). Hence, in the Voronoi superpixel image they have higher resolution and their details are more finely represented, as the example in the red box of (g) and the example in the yellow box of (h). The SPASM algorithm's ability of generating adaptive segmentation with respect to user-specified density distribution is demonstrated through these two tests and the comparison of their results.

(a) image gradient (b) disparity map



(c) 2D online SOM learning result (d) 2D online SOM learning result

(e) superpixel segmentation result (f) superpixel segmentation result

(g) Voronoi image (h) Voronoi image

FIGURE 5.6: Results obtained with image gradient (left column) and disparity map (right column) as density distribution. Both the cellular matrix and the SOM grid are in hexagonal topologies. The *Teddy* benchmark from [SS03] is used. (e) and (g) are the results from (c); (f) and (h) are the results from (d).

5.3.2 SPASM vs. SLIC

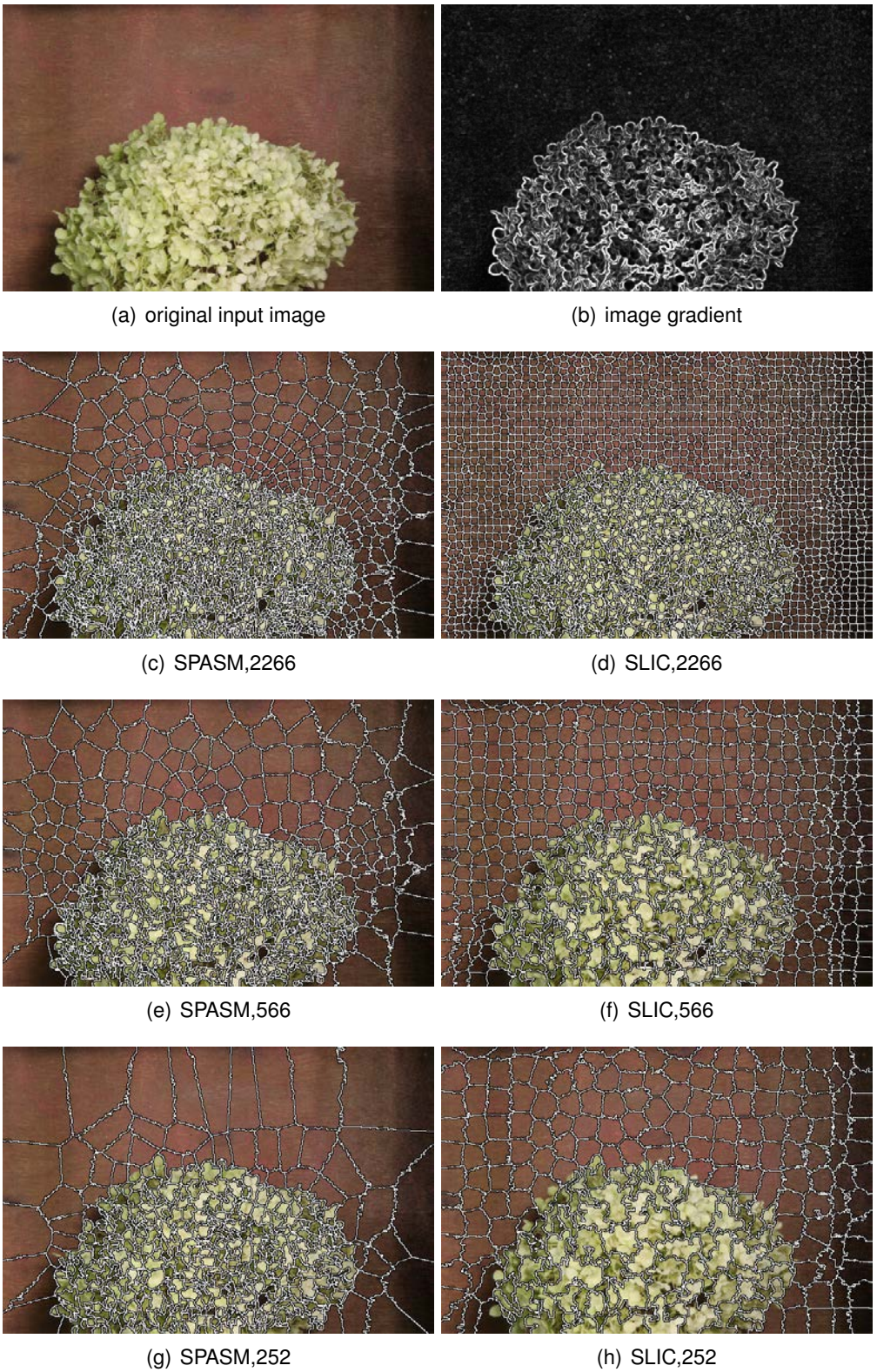
A comprehensive survey and comparison study of superpixel algorithms can be found in [ASS+12], where a fast algorithm called *simple linear iterative clustering* (SLIC) is proposed to adapt *k*-means clustering to generate superpixels with good adherence to image edges/boundaries. The SPASM algorithm extends the state-of-the-art SLIC algorithm, using our adaptive meshing tool to add compression abilities, with respect to the density distribution of image attributes and the topological relationship of cluster centers. Differently from SLIC which performs a restricted nearest point search within a square region, through our cellular matrix model, we can conduct the true closest point finding in a massively parallel way, using the efficient spiral search algorithm under different topologies.

We compare the SPASM algorithm with the SLIC algorithm using its publicly available source code[2]. The image gradient, as shown in Figure 5.7 (b), is used as density map for SPASM. We respectively set the cluster center grid as a zoom-out map in the cellular matrix model, at levels $R = 10, 20, 30$, which respectively correspond to $N/100 = 2266$, $N/400 = 566$, $N/900 = 252$ cluster centers ($N = 584 \times 388$ being the input image size). For SLIC, we set the initial superpixel size accordingly, in order to make the two algorithms work with same number of initial cluster centers. As shown in Figures 5.7 (c), (e), and (g), in all cases the SPASM algorithm produces a high density of superpixels in the edge-dense area (the flower clump) while the background is covered with relatively coarse superpixels sparsely. On the other hand, as illustrated in Figures 5.7 (d), (f), and (h), no matter how the initial superpixel size is set, the SLIC algorithm always generates uniformly distributed cluster centers. This is because the SLIC algorithm is a tailored *k*-means approach with no function of distribution learning like the SPASM algorithm. Therefore, the superpixel resolution of SLIC correlates with the uniform distribution of initial cluster centers, while SPASM allows finer-resolution regions selected by the density map attribute, meanwhile obtaining finer segmentation results in such regions, regardless of the number of initial cluster centers.

To provide a quantification of the segmentation quality in comparison to the SLIC algorithm, we propose to compute the *mean color deviation* (MCD) which is the mean deviation of pixel's true color to its cluster center's color, as defined in

$$MCD = \frac{1}{N}\sum_{i=1}^{N}(\|\boldsymbol{x}_{rgb}(i) - \bar{\boldsymbol{x}}_{rgb}(center(i))\|), \tag{5.1}$$

[2] *http://ivrl.epfl.ch/research/superpixels*



(a) original input image (b) image gradient

(c) SPASM,2266 (d) SLIC,2266

(e) SPASM,566 (f) SLIC,566

(g) SPASM,252 (h) SLIC,252

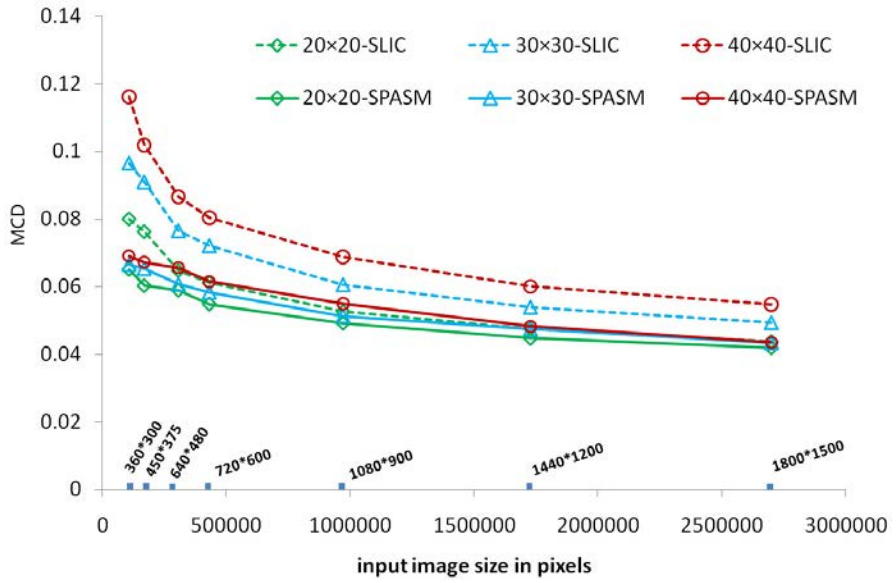
FIGURE 5.7: Segmentation result comparison between SPASM and SLIC. From the second row to the last row, for both algorithms, the number of initial cluster centers is set to 2266, 566, and 252, respectively. The *Hydrangea* benchmark from [BSL+11] is used. The image size is 584×388.

where *N* is the input image size, $\boldsymbol{x}_{rgb}(i)$ is the ground truth 3D *RGB* color of the i^{th} pixel, and $\bar{\boldsymbol{x}}_{rgb}(center(i))$ is the color of the i^{th} pixel's cluster center.

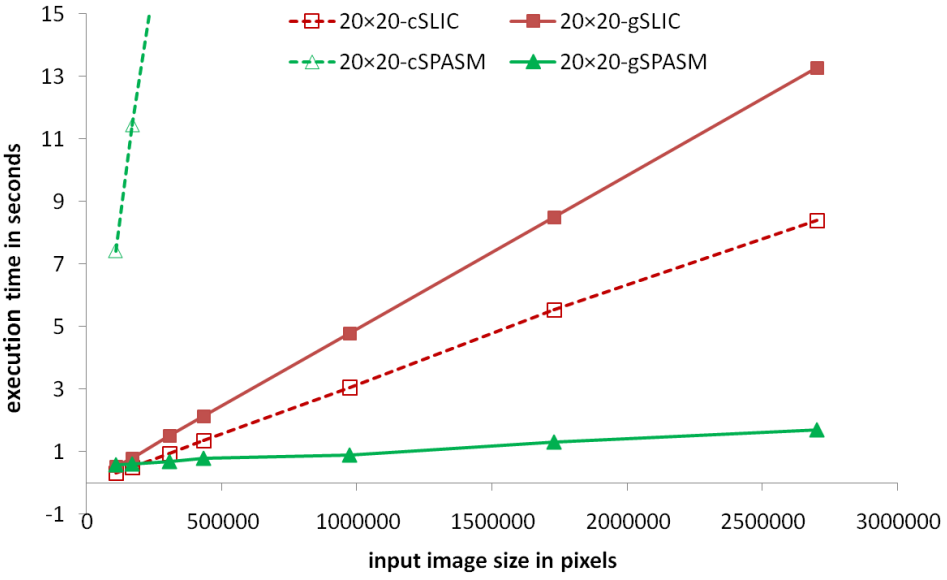
To test the performance with different input image sizes, we use seven input images of growing sizes and set different initial superpixel sizes (20×20, 30×30, 40×40). For a good adherence to image edges/boundaries, the distance parameters of SPASM are fixed as $(\tau_{spa}, \tau_{rgb}, \tau_{den}) = (1/2{R_c}^2, 1/3, 1/100)$ and the distance parameter (the m parameter in [ASS+12]) of SLIC is set to 20. The iteration number of SLIC is set to the default value of 10. All results are the mean values of 10 runs. We measure the result quality with MCD values which are reported in Figure 5.8 (a). Results from SPASM have smaller MCD values in all the tested cases when compared with results from SLIC. Also, the MCD values of SPASM results are steadier either with respect to different input image sizes, or with respect to different initial superpixel sizes. This shows the "adaptive" ability of our proposed SPASM which produces similar results regardless of the number of initial cluster centers and the input image size.

In order to compare execution time, we test four versions: the SLIC CPU implementation (cSLIC) from the raw public source code, the SLIC GPU implementation[3] (gSLIC) reported in [RR11], the SPASM CPU implementation (cSPASM) which is a sequential simulation of parallel SPASM algorithm, and the SPASM GPU implementation (gSPASM). The initial superpixel size is set to 20×20 for the four versions. The execution time results are reported in Figure 5.8 (b). For small size images (360×300 and 450×375), the execution time of cSLIC, gSLIC, and gSPASM is similar. For other larger size images, gSPASM clearly runs faster than all the other versions, especially for the largest image. cSPASM runs much slower than others for all images. This is because the sequential simulation of the iterative on-line SOM is time consuming. gSLIC runs slower than cSLIC for all images. This is because the gSLIC library from the raw public source code is specifically optimized for high-end GPU cards, with a lot of shared memory employment, while the platform we have used is a relatively out-dated GPU card. In average for the seven tested images, the acceleration factor of gSPASM against cSLIC is 2.44; the acceleration factor of gSPASM against gSLIC is 3.84; the acceleration factor of gSPASM against cSPASM is 39.59. In spite of the absolute execution time results of different implementations on different platforms (CPU and GPU), we think the more important conclusion is that the execution time of gSPASM increases in a linear way with a very weak increasing coefficient, when the input image size augments.

[3] *https://github.com/carlren/gSLICr*



(a)



(b)

FIGURE 5.8: Performance comparison between SPASM and SLIC: (a) comparison between SPASM and SLIC with different input image sizes and different initial superpixel sizes; (b) execution time results of CPU SLIC (cSLIC), GPU SLIC (gSLIC), CPU SPASM (cSPASM), and GPU SPASM (gSPASM), with different input image sizes. The *Cones* image from [SS03] is used.

5.4 Conclusion

In this chapter, we have implemented the cellular matrix model and its parallel SOM applications on GPU parallel computing platforms with CUDA. We have performed experiments on standard benchmarks and in comparison with state-of-the-art approaches. We have evaluated our GPU implementation of the parallel SOM TSP application, on different large-size TSP instances from different benchmarks. Our GPU SOM implementations run a lot faster than the CPU memetic SOM and Co-Adaptive Net, which are two neural network methods in the literature that perform experimental studies on large size benchmark instances. The execution time of our GPU implementation increases very slowly compared to either memetic SOM or Co-Adaptive Net, when the input instance size augments. We have conducted experiments on our GPU implementation of the structured meshing application and its counterpart CPU implementation. The execution time of GPU version for all six tested images is steady and no more than 0.36 seconds. The acceleration factor, which is the ratio of CPU version's execution time by GPU, varies from factor 5.84 to 8.33. We have also studied the relationship between execution time and input size, of GPU implementation and its counterpart CPU version, by carrying out experiments with four disparity maps, each at small, medium and large scales, respectively. The average acceleration factors of the three sets are 5.49, 12.68 and 39.74 respectively, as input size grows, which indicate augmentation of acceleration factor with input size. We have tested our SPASM algorithm utilizing two kinds of image attributes as the density distributions for cluster center initialization with the on-line SOM: image gradient and disparity value. The SPASM algorithm's ability of generating adaptive segmentation with respect to user-specified density distribution has been demonstrated through these two tests and the comparison of their results. We have compared the SPASM algorithm with the state-of-the-art SLIC algorithm, on seven input images of growing sizes, setting different initial superpixel sizes. We have tested both GPU version and CPU version for both two algorithms. Compared with others, the execution time of GPU SPASM increases in a linear way with a very weak increasing coefficient, when the input image size augments. The acceleration factor of GPU SPASM against its CPU sequential counterpart version is 39.59 in average for the seven tested images. For all the three applications, our GPU parallel versions always produce substantial acceleration over their sequential counterparts. An important conclusion is that the execution time of our GPU implementations for all the three applications increases in a linear way with a very weak increasing coefficient according to input size. Such a conclusion is encouraging to solve very large scale problems in an efficient way, under the proposed cellular matrix model.

Chapter 6

Distributed Local Search for Elastic Image Matching

6.1 Introduction

Based on the cellular matrix model, this chapter focuses on the design of a parallel local search algorithm applied to the elastic image matching problem formulated in our grid matching framework. We propose the *distributed local search* (DLS) algorithm, which is a parallel formulation of a local search procedure in an attempt to follow the spirit of the standard local search metaheuristics. Starting from its location in the cellular matrix, each processor locally acts on the data. It can execute local evaluation, perform neighborhood search, and select local improvement moves to execute. Applications of different operators for solution diversification are possible in a similar way to *variable neighborhood search* (VNS). Mutual exclusion between threads is guaranteed by the cellular matrix decomposition that allows independent moves of internal cell pixels. The solution results from the many operations simultaneously performed on the distributed data. Classical drawbacks to address are related to cell frontier management and solution diversification. We propose some responses to these issues.

Firstly, we formulate a general energy function to be equivalent to the elastic image matching problems. Then, we present the DLS algorithm in detail. We provide the data structures based on the cellular matrix model, which also partitions the solution between independent neighborhoods mutually exclusive to each other. We explain the local evaluation on cell level, in order to guarantee the mutual exclusion for parallel local evaluations among threads. We propose two strategies to eliminate conflict operations on cell frontiers, retaining the strategy of “dynamic change of cell frontiers” as the main solution for frontier management in our proposed DLS framework.

Afterwards, we design two classes of move operators considering small neighborhood and large neighborhood respectively, and we apply different operators in the DLS algorithm, combining them under the VNS framework. We present the details of GPU implementations from host and device sides, describing data transfers, kernel calls, and the main kernel function of DLS loop.

6.2 Elastic Grid Matching

We instantiate the energy function of our grid matching formulation to be equivalent to the elastic image matching problems such as the visual correspondence problems presented in the background of Chapter 2. Here, we call this class of problems the *elastic grid matching* problem.

Given two input images with same size and same regular topology, one is a matcher grid $G_1 = (V_1, E_1)$ where a vertex is a pixel with a variable location in the plane, while the other is a matched grid $G_2 = (V_2, E_2)$ where vertices are pixels located in a regular grid. The goal of elastic grid matching is to find the matcher vertex locations in the plane, so that the following energy function

$$E(G_1) = \sum_{p \in V_1} D_p(p - p_0) + \lambda \cdot \sum_{\{p,q\} \in E_1} V_{p,q}(p - p_0, q - q_0) \tag{6.1}$$

is minimized, where p_0 and q_0 are the default locations of p and q respectively in a regular grid; D_p and $V_{p,q}$ are defined the same way as the energy function defined in Subsection 2.4.3 in Chapter 2. A label f_p in visual correspondence represents a pixel moving from its regular position into the direction of its homologous pixel, *i.e.* $f_p = p - p_0$. In the following sections, we will directly use the notations of labels as relative displacements, as usual with such problems. The direction of projection is from $V1$ to $V2$. Given $p \in V_1$, the selected homologous pixel in G_2 is the closest point projection c_p^{V2} into the regular grid. According to the direction of projection, local search iterations are in the form of "project into", "evaluate and select", then "move".

6.3 Distributed Local Search (DLS)

It has been proven that elastic image matching is NP-complete [KU03], and finding the global minimum for the energy function even with the simplest smoothness penalty, the piecewise constant prior, is NP-hard [BVZ01, Vek99]. Therefore, it is impossible

to rapidly compute the global minimum unless P=NP. In our work, we choose the local search metaheuristics to deal with the energy minimization problem.

Based on the cellular matrix model, we propose a parallel local search algorithm, called *distributed local search* (DLS), to implement many local search operations on different parts of the data in a distributed way. It is a parallel formulation of local search procedures in an attempt to follow the spirit of the standard local search metaheuristics. Starting from its location in the cellular matrix, each processor locally acts on the data located in the corresponding cell according to the cellular decomposition, in order to do local evaluation, perform neighborhood search, and select local improvement moves to execute. The many processes locally interact in the plane, making evolve some current solution into an improved one. The solution results from the many independent local search operations simultaneously performed on the distributed data in the plane.

Normally, a local search algorithm with single operator obtains local minima. In order to escape from local minima, we design several operators. Applications of different operators for diversification are possible in a similar way to the VNS. We firstly introduce the detailed DLS algorithm in the section, then we provide different operators and their application in the VNS framework, in the following two sections.

Note that the simple and popular *winner-takes-all* (WTA) approach [Zha13] corresponds to a single projection step, and that DLS can be seen as an extension of WTA according to the energy function used, as well as to the many projection iterations until no improvement moves are possible.

6.3.1 Data Structures

The data structures and direction of operations for DLS algorithms are illustrated by Figure 6.1. The input data set is deployed on the low level of both matcher grid and matched grid, represented as regular images in the figure. The honeycomb cells represent the cellular matrix level of operations. No buffered cells are necessary, since direct access to the data is sufficient using spiral cells. Each cell is a basic processor that handles a basic local search processing iteration with the three steps: neighborhood generation step (**get**); neighbor solution evaluation and selecting the best neighbor (**search**); then moving the matcher grid toward the selected neighbor solution (**operate**). The nature and size of specific moves and neighborhoods will depend on the type of operator used and the level of the cellular matrix. The higher is the level, the larger is the local cell/neighborhood.

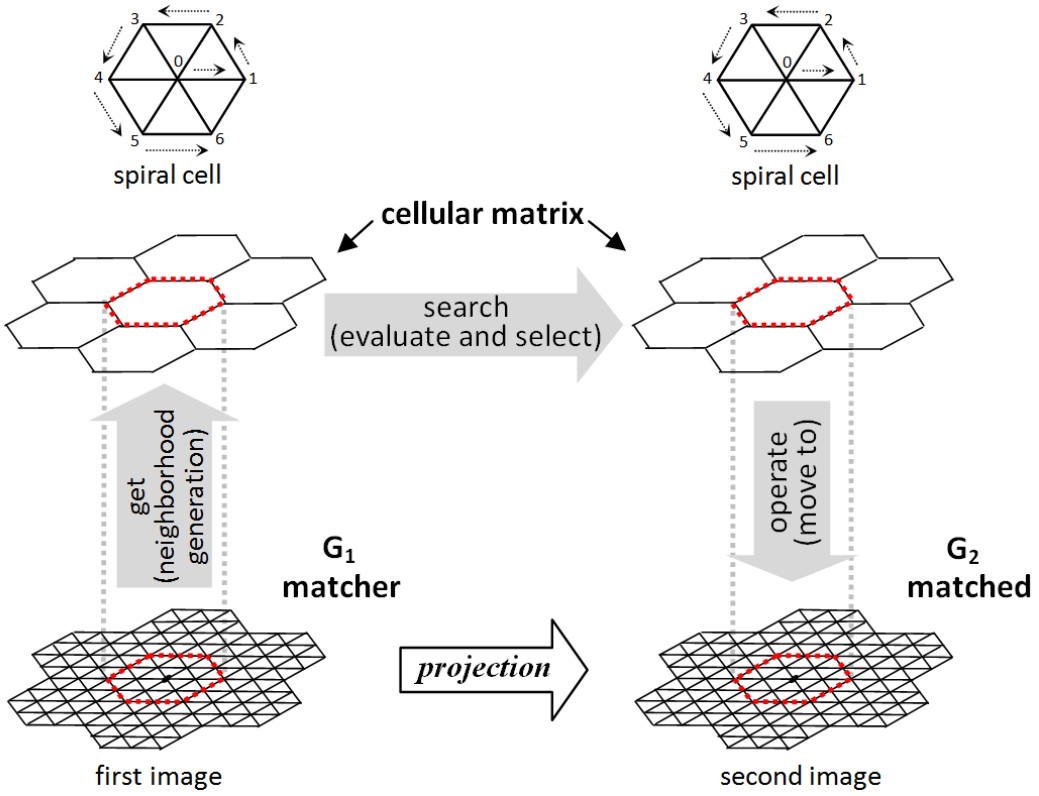


FIGURE 6.1: Basic projection for DLS.

6.3.2 Neighborhood Decomposition

In the cellular matrix model, a solution is composed of many sub-solutions from many cells. Each sub-solution is evolved from an initial sub-solution based on the distributed data in a cell. By partitioning the data and solution, the neighborhood structure is also partitioned at the same time.

Taking the pixel labeling problem of stereo matching for example, let us suppose that there are n pixels in the input image and k candidate labels (possible disparities). Then, the largest neighborhood structure includes k^n neighbors (solutions) which are all the possible labelings. After the cellular matrix decomposition, let us suppose that there are a constant number (C) of pixels in each cell, and there are n/C cells in the cellular matrix. Then for every cell, the largest neighborhood structure includes k^C neighbors. Therefore in DLS, the original exponential exploration in a large neighborhood ($O(k^n)$) is transformed into n/C simultaneously parallel exploration in a smaller neighborhood ($O(k^C)$). This kind of transformation has no impact on the result quality, since the local neighborhood move in a cell only contributes locally to the corresponding part of the energy function, without influence on the energy of the sub-solutions in other cells. This conclusion holds true as long as we can guarantee that, among the many local neighborhood moves that are executed in parallel triggering many parallel local

evaluations on the energy function, there are no neighborhood move conflict caused by more than one thread moving the same pixel at the same time. This conflict only happens on pixels that lie on cell frontiers. In the following two subsections, we explain the local evaluation principle and the strategies we propose to eliminate possible neighborhood move conflicts on cell frontiers.

6.3.3 Local Evaluation with Mutual Exclusion

The energy function of the problem serves as the fitness function of local search metaheuristics. Here, we describe the process of local evaluation in order to compute the energy improvement locally impacted by the pixels moves in a cell. According to the formulation of the energy function defined in Subsection 2.4.3, the energy is a summation of all pixel data costs $D_p(f_p)$ plus a weighted summation of all unordered pairs of smoothness costs $V_{p,q}(f_p, f_q)$. Under the cellular matrix decomposition, the energy for a given cell$_i$ is then defined as

$$E_i(f^i) = \sum_{p \in \mathcal{P}_i} D_p(f_p^i) + \lambda \cdot \sum_{\{p,q\} \in \mathcal{N}_i} V_{p,q}(f_p^i, f_q^i), \tag{6.2}$$

where $\mathcal{P}_i$ is a subset of $\mathcal{P}$ (the set of all pixels in the image) including all the pixels that locate in the cell$_i$; $\mathcal{N}_i$ is a subset of $\mathcal{N}$ (the set of all unordered pairs of neighboring pixels in the image) including all the unordered pairs of neighboring pixels in the cell$_i$; f^i is the labeling restricted to the cell$_i$; f_p^i and f_q^i are the labels of pixel p and q respectively. Then, the global energy function is

$$E(f) = \sum_{i=1}^{n/C} E_i(f^i), \tag{6.3}$$

where n/C is the number of cells in the cellular matrix model, such that

$$\begin{aligned} \mathcal{P}_1 \bigcup \mathcal{P}_2 \bigcup, ..., \bigcup \mathcal{P}_{n/C} = \mathcal{P}, \\ \mathcal{N}_1 \bigcup \mathcal{N}_2 \bigcup, ..., \bigcup \mathcal{N}_{n/C} = \mathcal{N}. \end{aligned} \tag{6.4}$$

Note that in Equation 6.2, the frontier pixels[1] are counted in f^i, $\mathcal{P}_i$, and $\mathcal{N}_i$. Therefore, the contributions of frontier pixels can be computed twice by adjacent cells, in the global energy function of Equation 6.3. This has no impact on the validity of the evaluation. The global energy change incurred by the move of cell$_i$ is given by

$$\Delta E(f, f') = E_i(f^i) - E_i(f^{i'}) \tag{6.5}$$

[1] We refer to frontier pixels as the pixels that locate on the cell frontiers, according to the cellar matrix partition of the image.

where f, f' are the labelings before and after the move respectively. Then, a local move that improves the cell local cost necessarily improves the global energy, as long as the frontier pixels of this cell are not moved by other threads at the same time. We call this situation *mutual exclusion*, and we propose a specific strategy for the mutual exclusion guarantee, called "dynamic change of cell frontiers" and explained in the next section.

Equation 6.2—Equation 6.5 correspond to the local evaluation on the **cell level**. A finer-grained local evaluation could be carried out on the **pixel level**. Since a single-pixel move only changes the data cost of this pixel and four pairs of smoothness costs (suppose this pixel is not on image border), the energy difference introduced by the move of a single pixel p at the position (x, y) can be computed by

$$\Delta E(f, f') = D_p(f_p) + \lambda \cdot \sum_{\{p,q\} \in \hat{\mathcal{N}}_p} V_{p,q}(f_p, f_q) - D_p(f'_p) - \lambda \cdot \sum_{\{p,q\} \in \hat{\mathcal{N}}_p} V_{p,q}(f'_p, f_q), \quad (6.6)$$

where f_p, f'_p are the labels of p before and after the move respectively; $\hat{\mathcal{N}}_p$ is a set of four pairs consisting of p with its upper, lower, left, and right neighboring pixel respectively, i.e.

$$\hat{\mathcal{N}}_p = \{\{(x, y), (x, y-1)\}, \{(x, y), (x, y+1)\}, \{(x, y), (x-1, y)\}, \{(x, y), (x+1, y)\}\}. \quad (6.7)$$

Then in this case, the mutual exclusion corresponds to the situation that no other thread is moving either p or its four neighboring pixels while one thread is moving p.

In DLS, if a small move operator is employed, which only moves one pixel at a time, we could use either pixel-level local evaluation or cell-level local evaluation. But clearly, the pixel-level local evaluation is faster since it only computes on related pixels while the cell-level one will compute on all the pixels in the cell. If a large move operator is employed, which simultaneously moves multiple pixels in a cell, we should choose the cell-level local evaluation. The designs of small move operators and large move operators are given in Section 6.4.

6.3.4 Management of Cell Frontier Access

During the parallel operation, the coherence of local evaluation with mutual exclusion is violated by conflict operations. A conflict operation occurs when a same pixel or two neighboring pixels is/are being evaluated and moved simultaneously by two threads. Conflict operations only happen on frontier pixels, which are the pixels on the cell frontiers according to the cellular matrix partition of the image. This is because in our data structures, a cell contains its frontier pixels. Therefore, a frontier pixel is included

by more than one cell, and it might be handled by more than one thread at the same time. By contrast, a non-frontier pixel is exclusively included by only one cell, hence it is handled by one thread all the time.

We propose two strategies in order to eliminate the conflict operations in DLS. In the first strategy, called *dynamic change of cell frontiers* (DCCF), we limit the move to the internal pixels of a cell only. Cell frontier pixels remain at fixed locations, and they are not concerned by local moves so that exclusive access of the thread to its internal region delimited by the cell, is guaranteed. A problem that arises is how to manage cell frontier pixels and make them participate in the optimization process. As a solution, the cellular matrix decomposition is dynamically changeable from the CPU side before the application of a round of DLS operations. At different moments, the cellular matrix decomposition slightly shifts on the input image in order to change the cell frontiers and consequently the fixed pixels. For a given cellular matrix decomposition, cell frontier pixels are then fixed and not allowed to be moved by current DLS operations.

In DLS, if a small move operator is employed, which only moves one pixel at a time, we only need the pixel-level local evaluation for fast execution. In this case, even though the DCCF strategy still works, we propose another strategy, called *synchronized execution*, to guarantee the mutual exclusion on pixel level in a faster way. In this strategy, cell frontier pixels are not fixed, and the cellular matrix decomposition is constant. DLS operations for pixels in the same cell are carried out in a deterministic order. Parallel operations among different cells are highly synchronized, and they are performed on pixels far away from each other.

We have implemented the two strategies. The synchronous strategy allows for faster execution with only single pixel moves. The dynamic change strategy allows for more complex operators in larger neighborhood, moving a set of pixels instead of a single pixel. This strategy is then retained as the main solution for cell frontier management in our proposed DLS framework. We now detail the implementations of the two strategies.

6.3.4.1 Dynamic Change of Cell Frontiers (DCCF)

In this strategy, we fix cell frontier pixels and do not allow any DLS move operations on them. Note that the cell frontier pixels are decided by the cellular matrix decomposition. Then, if we change the cellular matrix decomposition, the cell frontier pixels change at the same time: some previously fixed pixels become free to move while some previously unfixed pixels turn to fixed. After a few times of changing the cellular matrix decomposition, all pixels can be made participate in the optimization process. The

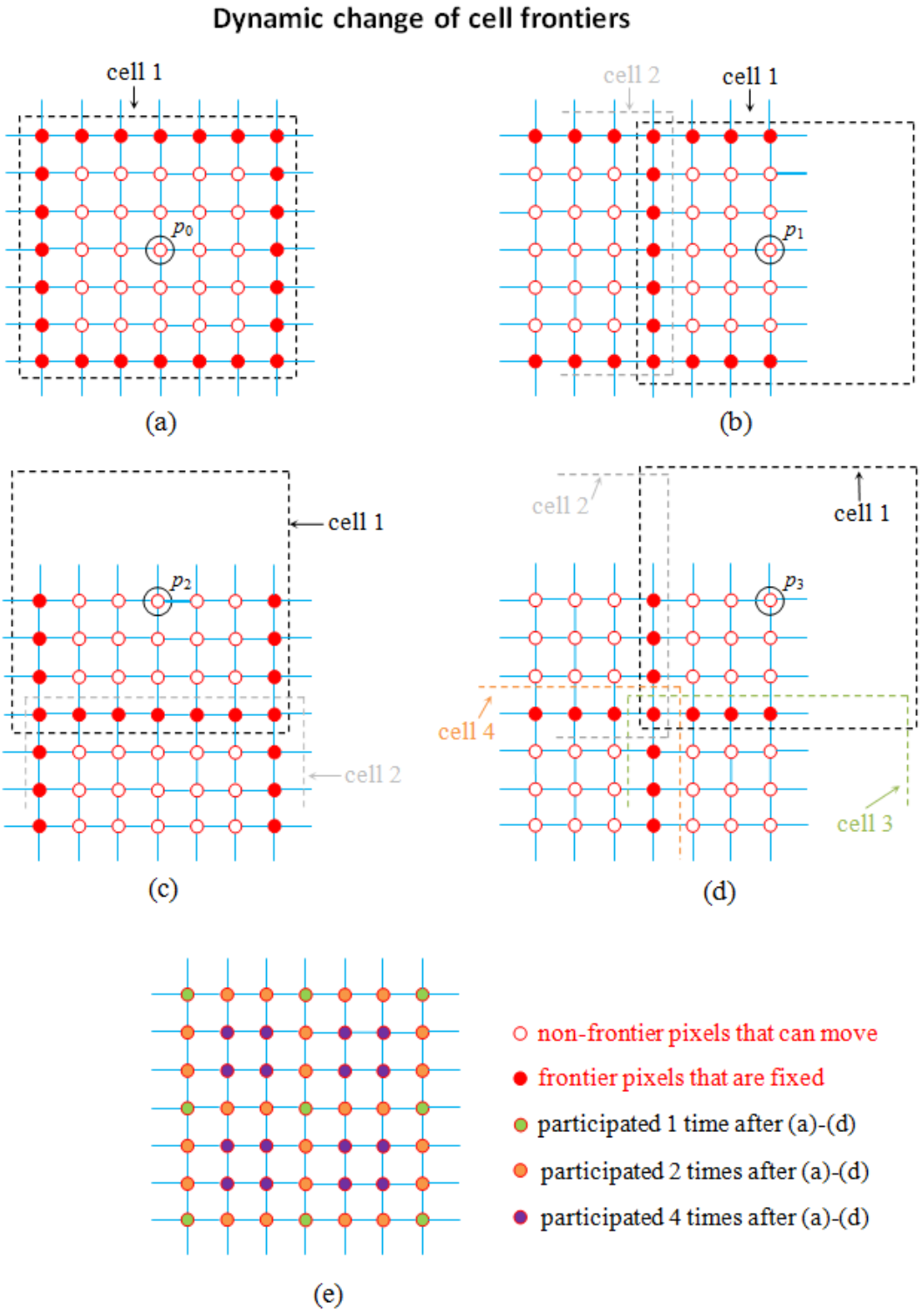


FIGURE 6.2: An example of dynamic change of cell frontiers. Red circles represent pixels. Rectangular partition delimits pixels into different cells of the cellular matrix decomposition. In (e) different colors mean different times that pixels have participated in the optimization process after the four DLS operation periods with the four cellular matrix decompositions of (a)—(d).

change of the cellular matrix decomposition dynamically happens during the whole DLS algorithm.

The DCCF can be implemented by simply shifting the cellular matrix decomposition in the plane. Such an example is illustrated in Figure 6.2. The center position and the cell radius R together decide the cellular matrix decomposition. In (a), the center position is the pixel p_0 denoted by a black circle, and the cell radius R is 3. In this example, we only analyze the 49 pixels that locate in the center cell according to the cellular matrix decomposition defined by (a). In (a)—(d), the red pixels are cell frontier pixels which are fixed, according to the respective cellular matrix decompositions. Compared with (a), the cellular matrix decomposition in (b) is shifted in the horizontal direction with the new center position p_1 which is three-pixel right of p_0; the cellular matrix decomposition in (c) is shifted in the vertical direction with the new center position p_2 which is three-pixel up of p_0; the cellular matrix decomposition in (d) is shifted in the lower-left-to-upper-right diagonal (secondary diagonal) direction with the new center position p_3 which is three-pixel upper-right of p_0. So, starting from the cellular matrix decomposition of (a) and then after a round of the three shifts of (b)—(d), any of the 49 pixels becomes unfixed for at least one time, so as to participate in the optimization process for at least one time. As indicated in (e), some of the pixels have participated for two times, while some of the pixels have participated for four times.

In the DCCF strategy, multiple pixels in one cell can be moved together as long as they are not fixed. Also, pixels at arbitrary positions could be moved at arbitrary period of the DLS process. These properties increase the various possibilities compared with the synchronized execution strategy. However, additional costs for storing and dynamically switching different cellular decompositions are necessary. This method of cellular matrix changes is retained for our framework since it is more general than the synchronous strategy.

6.3.4.2 Synchronized Execution

In this strategy, cell frontier pixels are not fixed, and the cellular matrix decomposition is constant. In every cell, operations are performed one by one on pixels in a deterministic order. The order is the same for different cells. An example of the synchronized execution is illustrated in Figure 6.3. The DLS operations in a cell consist of 49 execution steps: one pixel is dealt with at each step. Starting for the center pixel, operations are carried out on pixels one by one in a counter-clockwise order until all the 49 steps are finished. To guarantee that no conflict occurs, execution steps in different cells are synchronized. Thus, parallel DLS operations among different cells

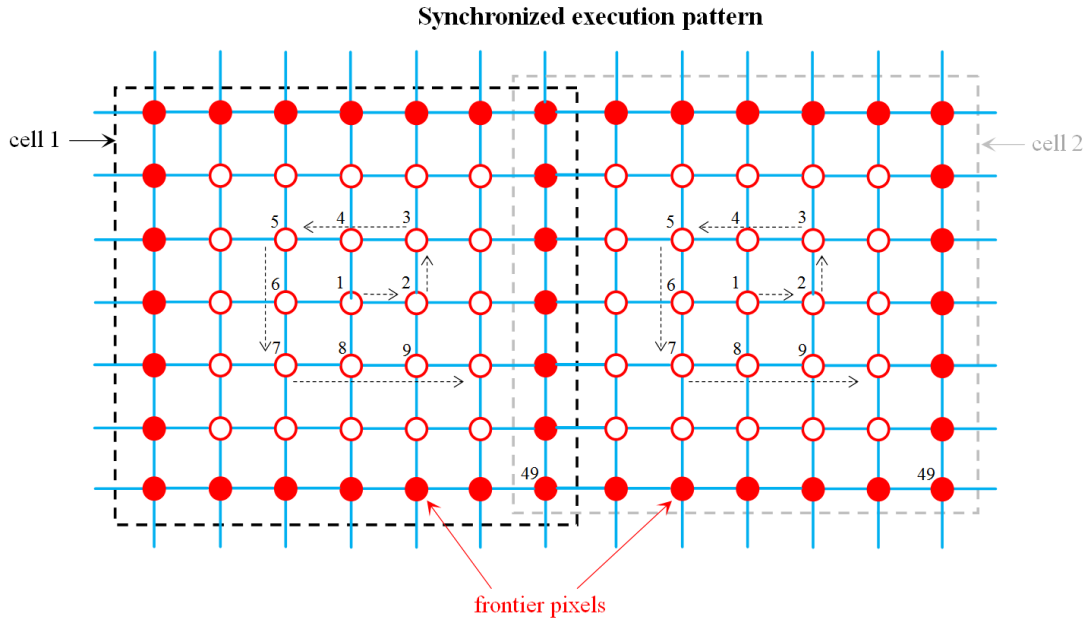


FIGURE 6.3: Synchronized execution pattern. Red circles represent pixels. Numbers and dash line arrows indicate the order of execution steps. Rectangular partition delimits pixels into different cells in the cellular matrix decomposition.

are performed at a same pace. At a given time, the DLS operations are at the same execution step in all cells, while the distributed pixels that are being handled in the same step are far way from each other so that no conflict will ever happen.

The synchronized execution strategy is specially designed for the fast pixel-level local evaluation with mutual exclusion, when only small move[2] operators are employed in the DLS. However, the drawback of this strategy is that it limits the variety of operator behaviors. For example, if we want to make a move on an arbitrary pixel in a cell meanwhile make another move on an arbitrary pixel in another cell, or if we want to make large moves[3] on some randomly picked pixels in a cell, these situations can not be allowed. Hence, this strategy appears to be too restrictive even if easy to implement and fast. By contrast, the DCCF strategy can eliminate conflict operations without sacrificing the diversification of DLS operator behaviors.

6.4 Neighborhood Operators

In this section, we present different neighborhood operators that we design for the DLS algorithm applied to the elastic grid matching. We use the notations of labeling problems to present these operators. Move operations in a given neighborhood

[2] A small move operation only moves one pixel at a time.
[3] A large move operation simultaneously moves multiple pixels in a cell.

structure correspond to changing labels of pixels in the corresponding labeling space. Operators are classified between small moves and large moves. In the first case, only a single pixel from the cell moves at a time; in the second case, larger sets of pixels from a cell can simultaneously move.

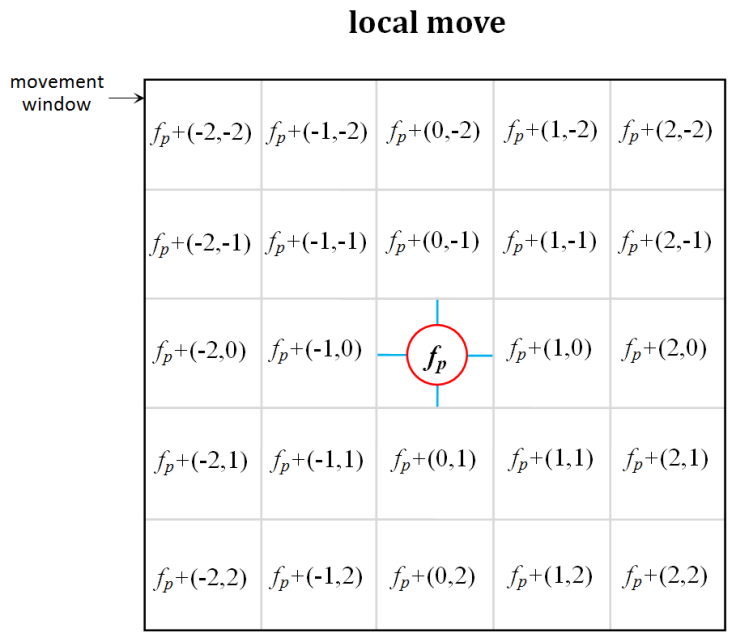
6.4.1 Small Move Operators

In a move operation, if only one pixel moves meaning that only one pixel's label is changed, this kind of move operation is called *small move* operation. We designed two small move operators: *local move operator* and *propagation operator*.

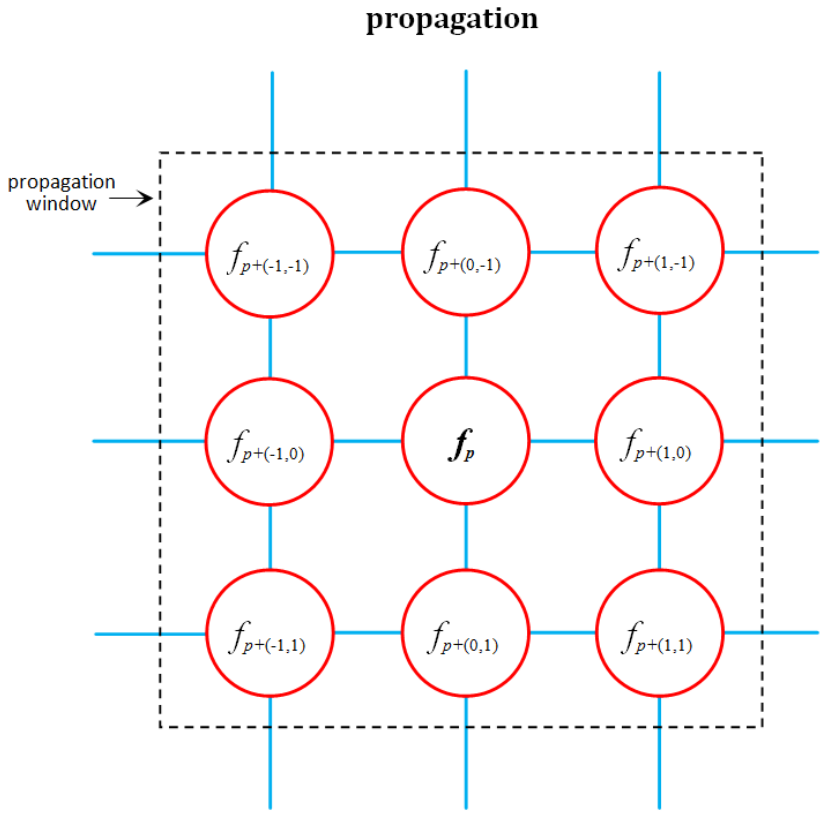
6.4.1.1 Local Move Operator

A move operation can be viewed as moving the considered pixel around its current position within the range of a given window. Then, a local move operator applies an increment/decrement to the current label of the considered pixel. An example of the neighborhood in the matched grid, where the movement takes place, is presented in Figure 6.4 (a), where the current label of the moving pixel p (center red circle) is f_p. All the candidate labels within a squared movement window with radius of 2 are listed. Each candidate label, corresponding to a position in the window, is obtained by adding an incremental/decremental displacement to the current label. Twenty-four candidate labels are indicated in Figure 6.4 (a). The center position of the window corresponds to the position of the considered pixel. Therefore, the local move operations can be viewed as moving the considered vertex (pixel) in the matcher grid, around its current position in the plane and within the range of the movement window.

Two parameters of the local move operator are the $radius$ of the movement window and the $scale$ factor of the movement. The $radius$ defines the movement range of the operator while $scale$ defines the moving step size. In the example of Figure 6.4 (a), $radius = 2$ and $scale = 1$. Here, $scale = 1$ means that the movements are based on the pixel wise precision. A move from f_p to $f_p + (-1, 0)$ means moving the considered pixel to the left-hand side by one pixel. If we set $scale$ less than 1, for example $scale = 1/2$, then the movements are based on the sub-pixel wise precision, and a move from f_p to $f_p + (-1, 0)$ means moving the considered pixel to the left-hand side by half pixel. In the example of Figure 6.4 (a), the movement window is a squared window. It could also be hexagonal window or rhombus window in our implementation, independently of the particular topology.



(a)



(b)

FIGURE 6.4: Two small move operators. Red circles represent pixels. f represents label. In (a), small squares in the movement window represent positions in the plane.

6.4.1.2 Propagation Operator

A propagation operator takes the labels of the considered pixel's neighboring pixels, as the candidate labels, and it replaces the current label with the best one found in the propagation window. It "propagates" the good label from the neighboring pixels within the propagation window, to the considered pixel. An example of propagation window with radius 1 is presented in Figure 6.4 (b), where the grid represents the matcher grid, and the current label of the considered pixel p (center red circle) is f_p. All the labels of the 8 neighboring pixels are within a squared propagation window with radius 1. Note that in the propagation operator, the definition of neighboring pixels are not only restricted to the standard 4-connected neighborhood system (as in the smoothness energy term), and all the pixels locate in the propagation window are neighboring pixels of the center considered pixel. The propagation operator can be viewed as moving the considered vertex (pixel) in the matcher grid, toward the current position (in the plane) of its neighboring vertex (in the topological grid) in order to be next to the neighboring vertex in the plane. This operator respects the smoothness constrains in visual correspondence applications that pixels belonging to the same object tend to move together.

The $radius$ of the propagation window is the only parameter of the operator. Normally a radius of 1 which includes 8 neighboring pixels is enough. Note that in our implementations, the propagation window could also be hexagonal window or rhombus window.

6.4.2 Large Move Operators

Large move operators consider multiple pixels. We design two groups of large move operators: *random pixels operators* and *random window operators*.

6.4.2.1 Random Pixels Move Operator

A *random pixels move* operator randomly picks several pixels in the considered cell, and it assigns a same candidate label to these pixels. An exemplary move of such an operator from the original labeling as shown in Figure 6.5, is illustrated in Figure 6.6 (a) where five randomly picked pixels in the black-line circles are the considered pixels, and they are all assigned with label of 3. The operator can be viewed as moving the considered vertices (pixels) in the matcher grid together toward a same direction, where

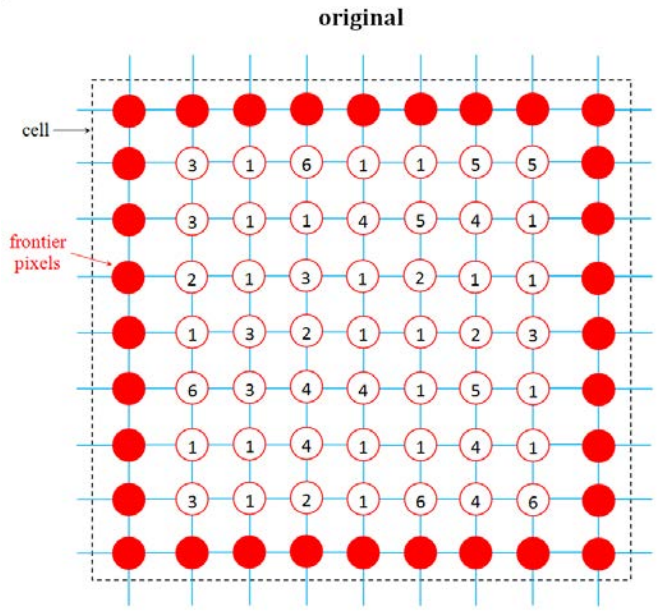


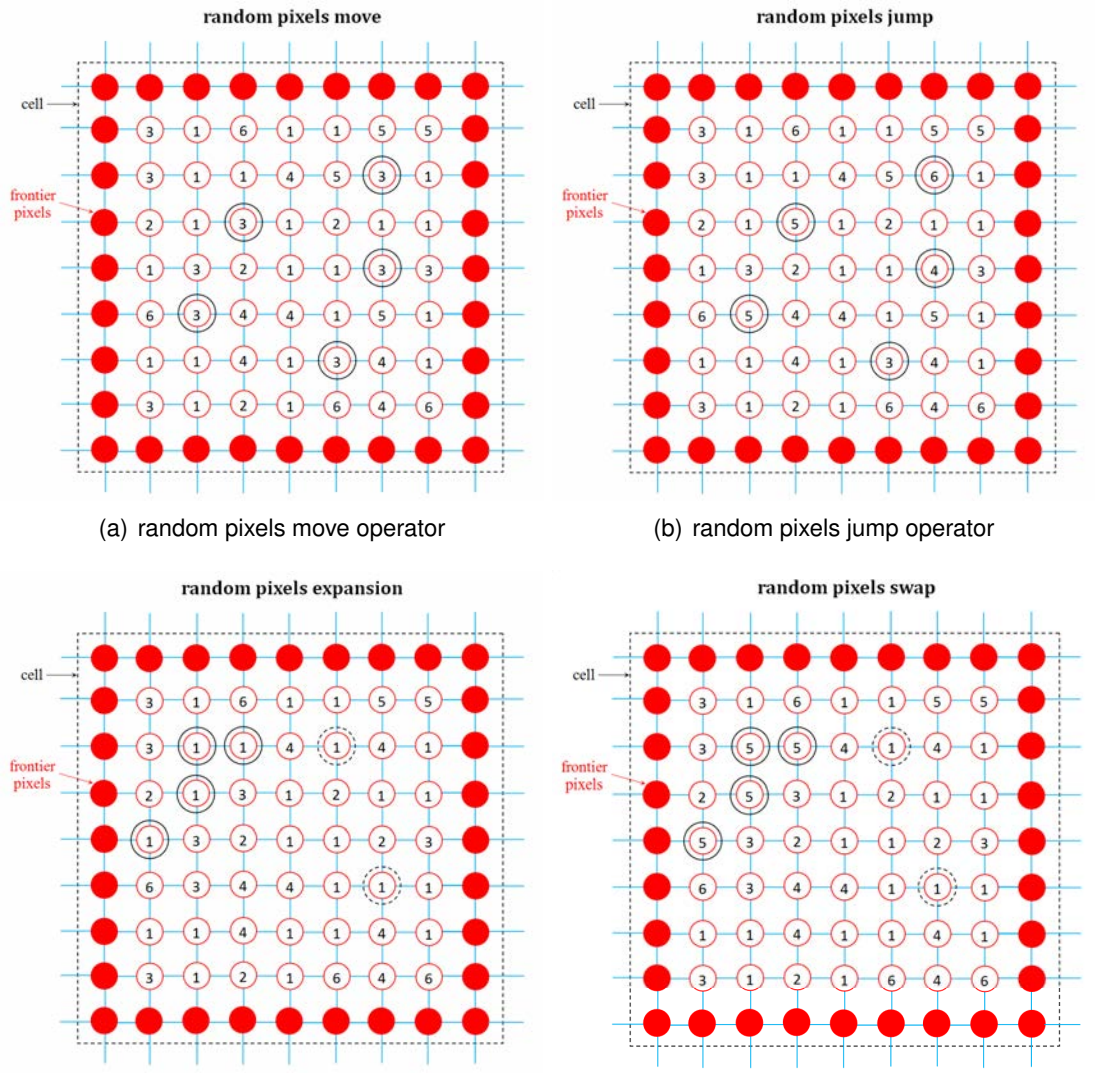
FIGURE 6.5: A labeling example. Vertices represent pixels. Different numbers represent different labels. Rectangular partition delimits pixels into different cells in the cellular matrix model. Note that this labeling example is used as the original labeling in order to explain different large move operators in Figures 6.6 and 6.7.

each considered vertex is moved with the same displacement from its original position in the plane before any movement.

The only parameter is $pickedNumber$ which indicates the number of pixels the operator randomly picks. The value of $pickedNumber$ should be no larger than the cell size (the number of pixels in a cell).

6.4.2.2 Random Pixels Jump Operator

A *random pixels jump* operator picks pixels in the same way as the random pixels move operator, and it applies a same increment/decrement to the current labels of the considered pixels. An exemplary move of such an operator from the original labeling in Figure 6.5, is illustrated in Figure 6.6 (b) where five randomly picked pixels in the black-line circles are the considered pixels, and their labels are all added by 2. The operator can be viewed as moving the considered vertices (pixels) in the matcher grid together toward a same direction, where each considered vertex is moved with the same displacement from its current position in the plane. The difference between the random pixels move operator and the random pixels jump operator is that the former operator moves vertices based on their original positions in the plane before any movement, while the latter operator moves vertices based on their current positions in



(a) random pixels move operator

(b) random pixels jump operator

(c) random pixels expansion operator

(d) random pixels swap operator

FIGURE 6.6: Example of random pixels operators. Vertices represent pixels. Different numbers represent different labels. Rectangular partition delimits pixels into different cells in the cellular matrix model. Note that the move operations by respective operators, are all based on the original labeling in Figure 6.5.

the plane. The only parameter is $pickedNumber$ which has exactly the same function as in the random pixels move operator.

6.4.2.3 Random Pixels Expansion Operator

A *random pixels expansion* operator randomly picks two groups of pixels, where pixels in the same group have the same label. It "expands" the label of one group to the other, setting the labels of all the pixels in the second group with the same label as the first group. An exemplary move of such an operator from the original labeling in Figure 6.5,

is illustrated in Figure 6.6 (c). Four randomly picked pixels with label of 1 in the black-line circles, are the first group of considered pixels; two randomly picked pixels with label of 5 in the black-dash-line circles, are the second group of considered pixels. In Figure 6.6 (c), the operator expands the label of the first group to the second group, setting the labels of the two considered pixels in the second group with label of 1.

The only parameter is $maxPickedNumber$ which indicates the max number of pixels the operator is allowed to randomly pick for each group. The value of $maxPickedNumber$ should be no larger than the cell size. Then for a given group, the actual number of considered pixels need to be picked is determined by randomly choosing a number from 1 to $maxPickedNumber$. This operator is inspired by the expansion move in some binary graph-cut implementations (see next operator).

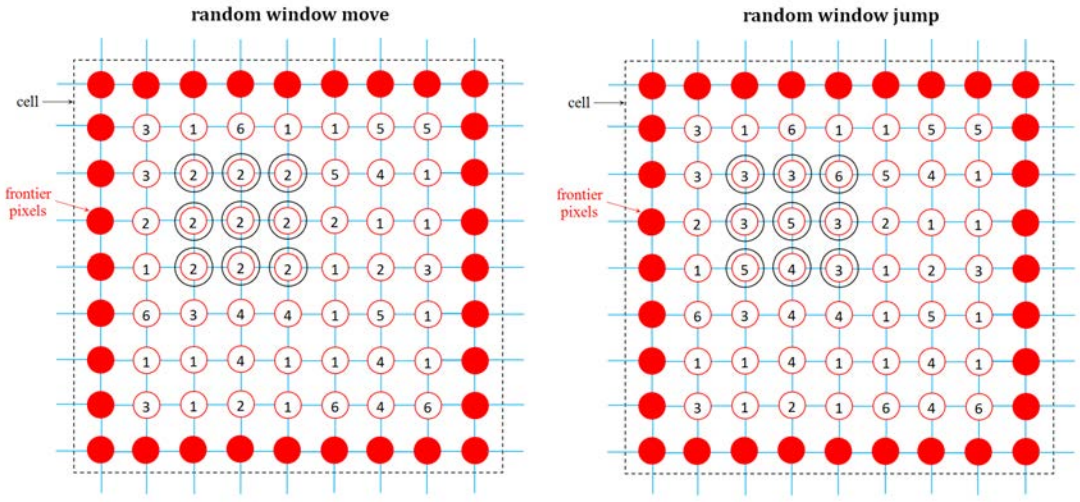
6.4.2.4 Random Pixels Swap Operator

A *random pixels swap* operator picks pixels in the same way as the random pixels expansion operator. It "swaps" the labels of the two groups, setting the labels of all the pixels in the second group with the label of the first group, meanwhile setting the labels of all the pixels in the first group with the label of the second group. An exemplary move of such an operator from the original labeling in Figure 6.5, is illustrated in Figure 6.6 (d). Four randomly picked pixels with label of 1 in the black-line circles, are the first group of considered pixels; two randomly picked pixels with label of 5 in the black-dash-line circles, are the second group of considered pixels. In Figure 6.6 (d), the operator sets the labels of the two considered pixels in the second group with label of 1, and it sets the labels of the four considered pixels in the first group with label of 5. The only parameter is $maxPickedNumber$ which has exactly the same function as in the random pixels expansion operator.

The designs of random pixels expansion operator and random pixels swap operator are inspired by the α-*expansion* move and the α-β-*swap* move in [BVZ01, Vek99], where the authors use these two moves to find local minimum for the labeling problem by binary graph cuts [FF62, BK04].

6.4.2.5 Random Window Move Operator

A *random window move* operator picks a fixed-sized window of pixels at a random position within the considered cell, and it assigns a same candidate label to all the pixels in this picked window. An exemplary move of such an operator from the original labeling in Figure 6.5, is illustrated in Figure 6.7 (a) where a square window with radius



(a) random window move operator (b) random window jump operator

FIGURE 6.7: Example of random window operators. Vertices represent pixels. Different numbers represent different labels. Rectangular partition delimits pixels into different cells in the cellular matrix model. Note that the move operations by respective operators, are all based on the original labeling in Figure 6.5.

of 1 is randomly picked, containing the nine considered pixels denoted by black-line circles. The operator assigns the labels of the nine considered pixels with label of 2. The operator can be viewed as moving a window of vertices (pixels) in the matcher grid together toward a same direction, where each considered vertex is moved with the same displacement from its original position in the plane before any movement.

The only parameter is $pickedWindowRadiu$ which indicates the radius of the randomly picked window of considered pixels. In Figure 6.7 (a) $pickedWindowRadiu$ is set to 1. Note that in our implementations the window could also be hexagonal window or rhombus window.

6.4.2.6 Random Window Jump Operator

A *random window jump* operator picks pixels in the same way as the random window move operator, and it applies a same increment/decrement to the current labels of all the pixels in this picked window. An exemplary move of such an operator from the original labeling in Figure 6.5, is illustrated in Figure 6.7 (b). The labels of the nine considered pixels are all added by 2. The operator can be viewed as moving a window of vertices (pixels) in the matcher grid together toward a same direction, where each considered vertex is moved with the same displacement from its current position in the plane. The difference between the random window move operator and the random window jump operator is similar to the difference between the random pixels move

operator and the random pixels jump operator: the moves of the former are based on original positions in the plane before any movement, while the moves of the latter are based on current positions in the plane. The only parameter of the random window jump operator is $pickedWindowRadiu$ which has exactly the same function as in the random window move operator.

6.5 DLS with Multiple Operators Under VNS Framework

We present the main GPU CUDA program that implements our DLS algorithm. The kernel calling sequence from the CPU side allows applications of different operators in the spirit of VNS and manages dynamic change of cellular matrix frontiers. According to our previous experiments, the repartition of tasks between host (CPU) and device (GPU) is actually the best compromise we found to exploit the GPU CUDA platform at a reasonable level of computation granularity.

6.5.1 DLS Execution Pattern from Host Side

The flow chart executed from the CPU side is presented in Figure 6.8. The data transfer between the CPU side and the GPU side only occurs at the beginning and the end of the algorithm. The two kernels that are called from the CPU side and executed on GPU are: the random number generation kernel and the DLS kernel.

On the GPU side, random numbers are needed for random move operators. The random numbers are generated in advance by the random number generation kernel stated in Subsection 3.5.4. This kernel is regularly called during the algorithm according to the random number generation rate.

It is the CPU side that controls DLS kernel calls with different operators executed within the dynamic change of cell frontiers (DCCF) pattern for frontier cells management. With several neighborhood operators in hand, we use them under the VNS framework in order to enhance the solution diversification.

6.5.2 DLS Main Kernel

The generic loop template is instantiated with one single DLS operator working, based on one cellular matrix decomposition, as illustrated in Kernel 11. In the input parameter list, $currentCellularMatrix$ indicates which cellular matrix decomposition

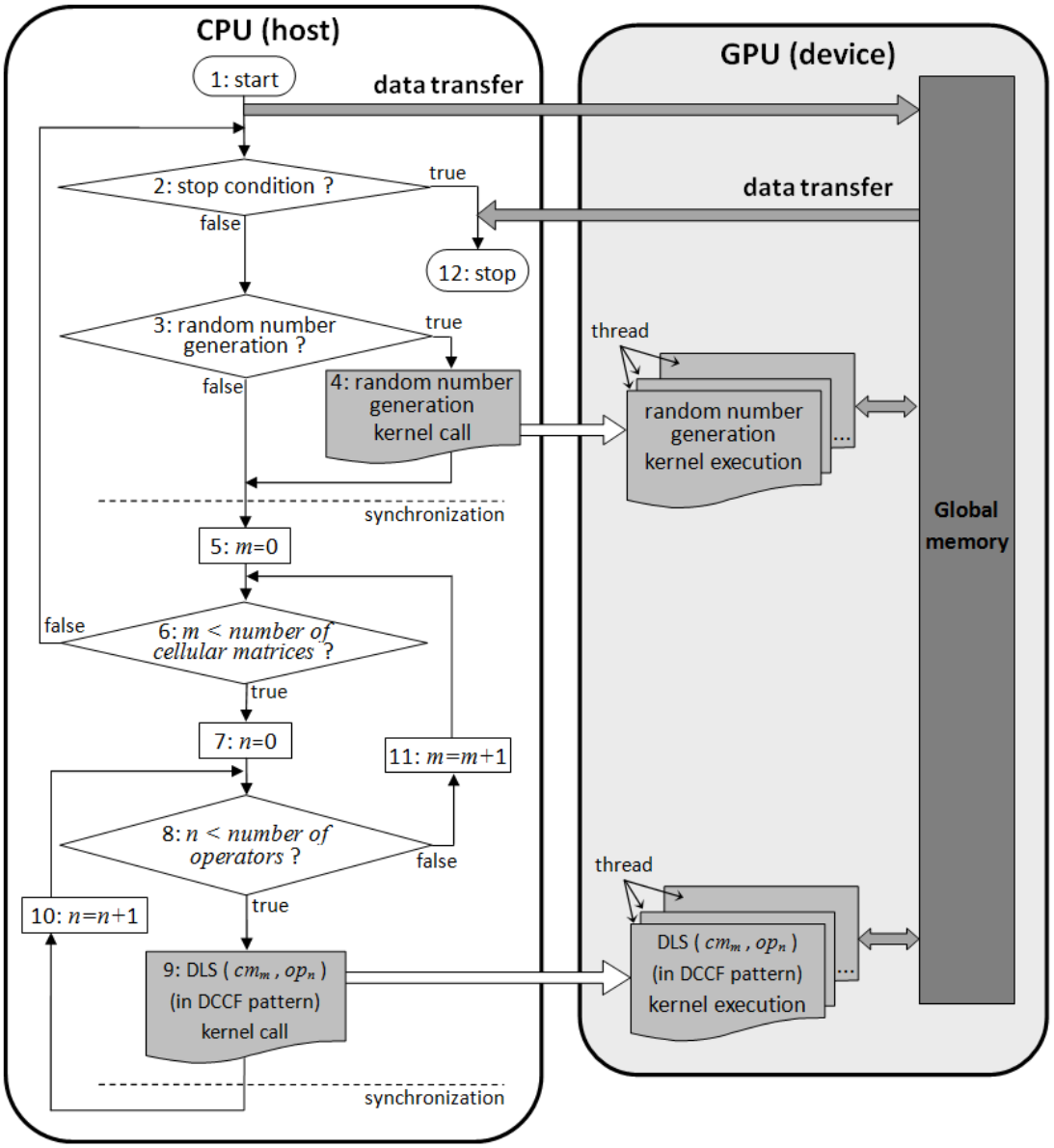


FIGURE 6.8: Flowchart of DLS implementation.

is currently used so that the $FGet.initialize$ function can set corresponding cell frontier pixels fixed. The $FGet.get$ function picks one pixel or multiple pixels that are not fixed. Based on the picked pixel/pixels, all the neighbor solutions G'_1 will be sequentially generated by $FNeighborhood.get$, and then be evaluated by $FEnergy.CellLevelEvaluation$. Cell-level local evaluation is employed in order to allow multiple pixels to move together. If the neighbor solution is better, the procedure replaces the current solutions G_1 with G'_1, through $FNeighborhood.move$. Note that in our practical implementation, the contribution after the neighborhood move is evaluated by $FEnergy.CellLevelEvaluation$, as indicated in Line 8 of Kernel 11, but the energy before the neighborhood move is not necessarily computed every time. It only needs to be done at the beginning of the DLS loop and stored as the current energy of the

Kernel 11: DLS loop in the DCCF pattern.

```
// For each cell_i of cellular matrix WD × HD, do:
Input: G_1, G_2, cell_i, FGet, FNeighborhood, FEnergy, currentCellularMatrix
1  begin
2      FGet.initialize(cell_i, currentcellularmatrix); ;
3      while FGet.next(cell_i) do
4          pixels ← FGet.get(cell_i, G_1); ;
5          FNeighborhood.initialize(cell_i, pixels); ;
6          while FNeighborhood.next(cell_i) do
7              G'_1 ← FNeighborhood.get(cell_i, pixels); ;
8              if FEnergy.CellLevelEvaluation(G'_1, G_2, cell_i) <
               FEnergy.CellLevelEvaluation(G_1, G_2, cell_i) then
9                  G_1 ← FNeighborhood.move(cell_i, pixels, G'_1); ;
10     return; ;
```

cell. Then in each execution of Line 8, the new energy after the neighborhood move, computed by $FEnergy.CellLevelEvaluation$, is compared with the current energy, and it replaces the current energy if the new energy is lower.

6.6 Conclusion

We have proposed the distributed local search (DLS) algorithm to handle elastic image matching. The GPU CUDA implementation consists of an instantiation of the main projection loop on device side, together with a kernel calling sequence on different operators and cellular matrix decompositions controlled from the CPU side. We have provided a solution to the problem of frontier pixel access by the use of dynamic changes of cell frontiers in the cellular matrix decomposition. Evaluations can be performed locally inside a cell, then guaranteeing a true evaluation of the global energy function. We have detailed operators in two classes: the first class contains the single pixel move operators (small neighborhood); the second class contains large move operators (large neighborhood). We have presented the details of algorithms from host and device sides, describing data transfers, kernel calls, and the main kernel function of DLS loop. The next chapter will deal with experimental evaluations of performances on standard elastic grid matching problems.

Chapter 7

Experimental Study of DLS to Visual Correspondence

7.1 Introduction

In this chapter, we experiment on the DLS algorithm applied to visual correspondence applications. First, we apply DLS to stereo matching, since comparative evaluations on energy minimization is relatively easy due to the many state-of-the-art methods that are freely available in software form. We evaluate operators individually, with different parameter settings. We also evaluate different combinations of operators, and different initializations, under the VNS framework we proposed. Then, we turn to comparative evaluations with *graph-cut* methods, *belief-propagation* methods and others. Comparative evaluations are performed according to the growing size of instances in order to evaluate the general evolution tendency of the GPU acceleration factor. All the analyses are performed on the stereo matching problems. Afterwards, we apply DLS to optical flow at the end of the chapter, with reporting visual results and ground truth evaluations performed on standard benchmarks. In that case, will be mentioned the problem of the correlation between energy minimization and ground truth evaluation.

7.2 Evaluation of Operators

In this section, we test different DLS operators with stereo matching application. We follow in the footsteps of Boykov et al. [BVZ98], Tappen and Freeman [TF03], and Szeliski et al. [SZS+08], using a simple energy function for stereo, applied to

benchmark images from the widely used Middlebury stereo data set [SS03]. The labels are the disparities, and the data costs are the absolute color differences between corresponding pixels for each disparity, as defined in Equation 2.14. For the smoothness term in energy function, we use a truncated linear cost as the piecewise smooth prior defined in Equation 2.21 where $u_{p,q} = 1$, $C = 2$, and $\| \cdot \|$ is the L1 norm. The smoothness weight parameter λ is set to 20.

We use DLS with the *dynamic change of cell frontiers* (DCCF) pattern. Since the purpose of the evaluation of operators is to analyze the potentials of different operators in terms of minimizing the energy function, we set the stop condition of DLS to convergence, in order to get the most out of operators. To be more specific, we stop the algorithm after 10 external iterations without lower energy found in any cell (by any thread). Note that this stop condition may significantly prolong the execution time of the algorithm, and it is only for the experimental purpose of this section. More often, we should set a reasonable number of external iterations in order to achieve a good compromise between energy and execution time, just like what we do during the experiments of the other sections in this chapter.

Instead of reporting the absolute energy values, we report the percentage deviation to the best known solution (lowest energy) as the $\%PDB$ value. We choose the best solution from the executions of several state-of-the-art energy minimization methods[1] from the literature, including the *tree-reweighted message passing* (TRW) [WJW05, SZS+08], *graph cuts* (GC) [BVZ01, BK04, SZS+08], and *belief-propagation* (BP) [Pea14, TF03, SZS+08]. Then, we compute the $\%PDB$ value as $\%PDB = (\text{DLS energy} - \text{lowest energy}) \times 100/\text{lowest energy}$.

7.2.1 DLS with Single Operator

In Figure 7.1, are reported the experimental results of DLS with only one operator, on the *Tsukuba* (384×288) image benchmark with the maximum number of labels (representing disparities) set to 16. We test the *random window move* operator, the *random pixels move* operator, and the *random pixels expansion swap* operator[2], respectively. Results show that for the random pixels move operator, larger $pickedNumber$ values lead to faster convergences toward higher energies, while smaller $pickedNumber$ values lead to slower convergences toward lower energies. For the random pixels expansion swap operator, the $maxPickedNumber$ value has more impact on energy than

[1] For all the tested energy minimization algorithms, we use the original codes from *http://vision.middlebury.edu/MRF/code/*.

[2] The *random pixels expansion swap* operator combines the *random pixels expansion* operator and the *random pixels swap* operator together, selecting the best move of these two operators to act.

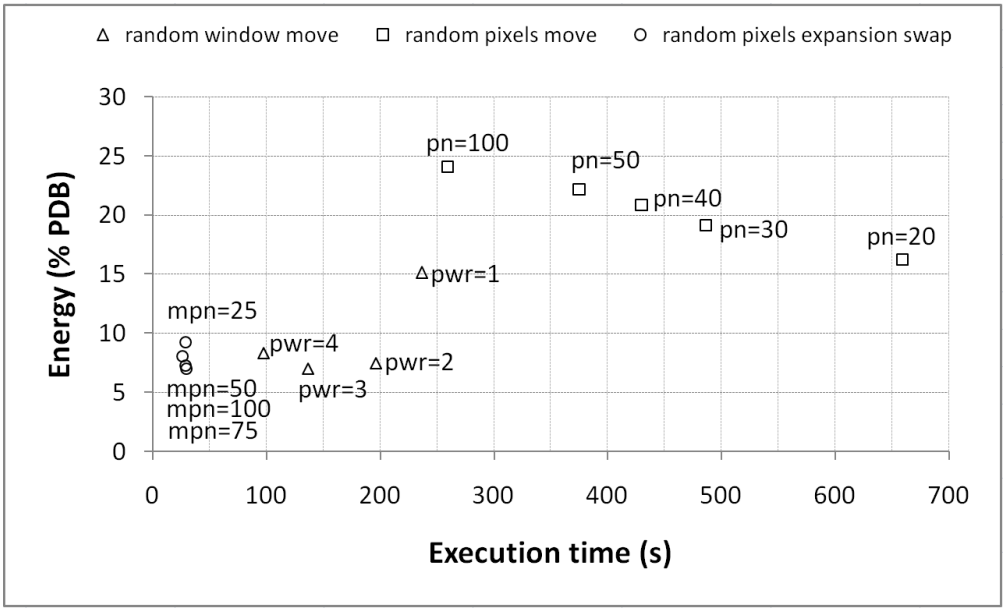


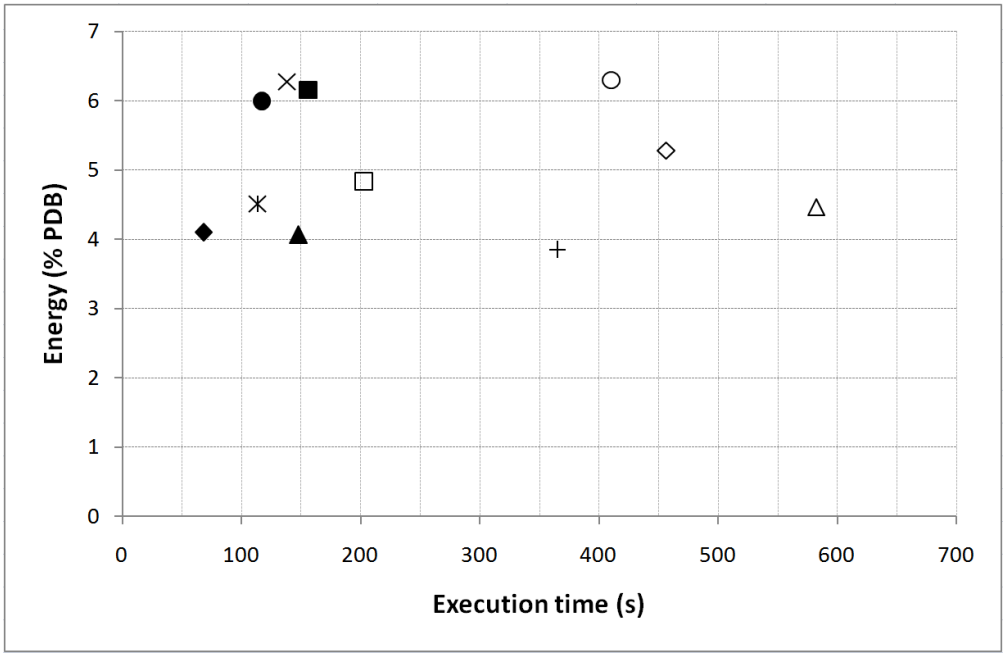
FIGURE 7.1: DLS with single operator on the *Tsukuba* benchmark. The x-axis shows execution time and the y-axis shows the $\%PDB$ value computed according to the lowest energy found by the TRW approach [WJW05, SZS+08]. All the results are mean values over 10 runs. **Acronyms**: "pwr" stands for $pickedWindowRadiu$; "pn" stands for $pickedNumber$; "mpn" stands for $maxPickedNumber$.

execution time, and it produces lowest energy with $maxPickedNumber$ value set to 75. The random window move operator produces lowest energy with $maxPickedNumber$ value set to 3, among the four $pickedWindowRadiu$ values. Best compromises between energy and execution time look provided by the random pixels expansion swap operator.

7.2.2 DLS with Combination of Operators

We also test DLS with combination of multiple operators under the proposed VNS framework as stated in Subsection 6.5. At each external iteration, different operators are applied to the DLS kernel one by one. The eleven combinations of operators that we have tested are listed in Figure 7.2 (b), and their experimental results on the *Tsukuba* (384×288) image benchmark are depicted in Figure 7.2 (a).

In principle, more operators tend to lead to better diversification for solution, hence lead to lower energies. This is supported by the result of the most complex combination—"local move + propagation + random window move + random pixels move + random pixels expansion swap"—which is composed of 5 operators and produces the lowest energy. Another observation is that the combinations which include the random pixels expansion swap operator are more likely to find lower energies than other combinations. We think this is due to the highly stochastic nature of the random pixels expansion

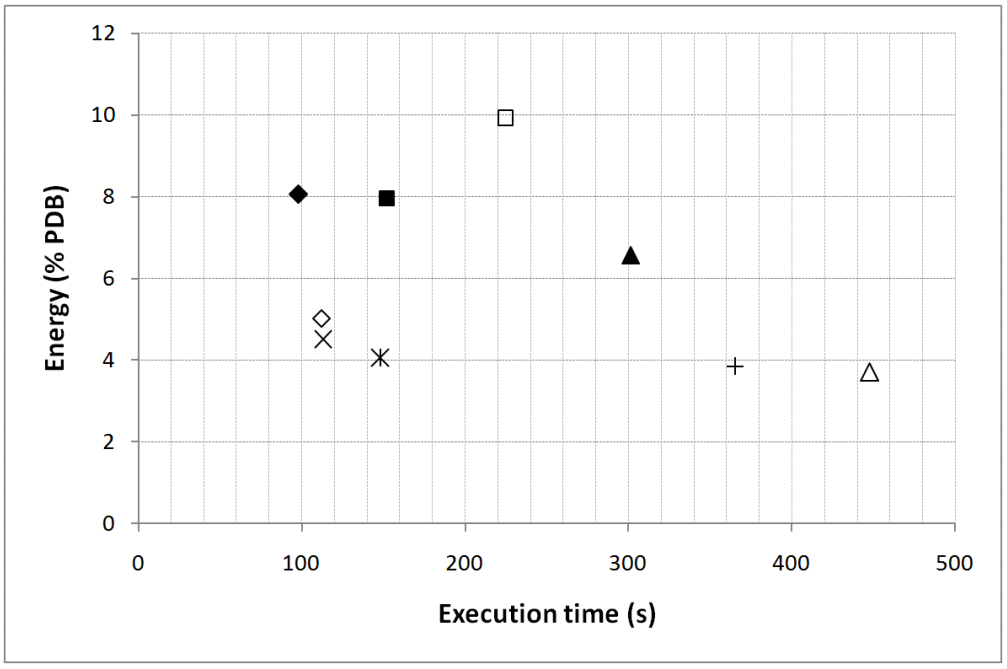


(a) results

symbol	number of operators	operator names
×	2	local move + random window move
✳	2	local move + random pixels expansion swap
●	2	propagation + random window move
◆	2	propagation + random pixels expansion swap
■	3	local move + propagation + random window move
▲	3	local move + propagation + random pixels expansion swap
○	2	random window move + random pixels move
◇	2	random pixels move + random pixels expansion swap
□	2	random window move + random pixels expansion swap
△	3	random window move + random pixels move + random pixels expansion swap
+	5	local move + propagation + random window move + random pixels move + random pixels expansion swap

(b) combinations of operators

FIGURE 7.2: DLS with different operator combinations, on the *Tsukuba* benchmark. The x-axis shows the execution time and the y-axis shows the $\%PDB$ value computed according to the lowest energy found by the TRW approach. All the results are mean values over 10 runs.



(a) results

symbol	init	number of operators	operator names
◆	0	2	local move + random pixels expansion swap
■	0	3	local move + propagation + random pixels expansion swap
▲	0	5	local move + propagation + random window move + random pixels move + random pixels expansion swap
◇	random	2	local move + random pixels expansion swap
□	random	3	local move + propagation + random pixels expansion swap
△	random	5	local move + propagation + random window move + random pixels move + random pixels expansion swap
×	WTA	2	local move + random pixels expansion swap
✳	WTA	3	local move + propagation + random pixels expansion swap
+	WTA	5	local move + propagation + random window move + random pixels move + random pixels expansion swap

(b) combinations of operators

FIGURE 7.3: DLS with different solution initializations, on the *Tsukuba* benchmark. The x-axis shows the execution time and the y-axis shows the $\%PDB$ value computed according to the lowest energy found by the TRW approach. All the results are mean values over 10 runs.

swap operator, where two randomly picked groups of pixels are considered. Also, the combination of "propagation + random pixels expansion swap" seems like a good choice, since it produces relatively lower energy than most of the others, with the shortest execution time. Therefore, we use this combination of operators for our comparative experiments between DLS and other state-of-the-art energy minimization methods, reported in Section 7.4.

7.3 Influence of Solution Initialization

The results obtained by local search metaheuristics are usually sensitive to the initial state of solution, especially in high-dimensional spaces with non-convex energies (such as those that arise in vision) due to the huge number of local minima [SZS+08]. We experiment on three different solution initializations: (1) "0 init" where DLS is started with all labels set to 0; (2) "random init" where DLS is started with each label set to a random value from 0 to the maximum label value (the largest disparity); (3) "WTA init" where we assign each pixel the label with the lowest data cost in a winner-take-all round. Results are reported in Figure 7.3. In each initialization case, we test three combinations with 2, 3, and 5 operators respectively, as listed in Figure 7.3 (b). According to the results shown in Figure 7.3 (a), "0 init" takes the least execution time in average, while "random init" takes the most execution time in average. "WTA init" produces the lowest energies in average. In all the following experiments on stereo matching problems, we will use "WTA init" as the solution initialization.

7.4 Trace Execution of Different Methods

We compare execution traces of different methods, on the stereo matching benchmarks, in order to better gauge their respective behaviors with respect to energy minimization as the number of iterations grows. We compare DLS with six other methods[3]: *iterated conditional modes* (ICM) [Bes86] which is an old approach using a deterministic "greedy" strategy to find a local minimum; sequential *tree-reweighted message passing* (TRW-S) [SZS+08] which is an improved version of the original tree-reweighted message passing algorithm [WJW05]; Belief-Propagation-S and Belief-Propagation-M [SZS+08] which are two updated version of the max-product *loopy belief propagation* (LBP) implementation of [TF03]; Graph-Cut-swap and Graph-Cut-expansion which are

[3]For all the tested energy minimization algorithms, we use the original codes from *http://vision.middlebury.edu/MRF/code/*.

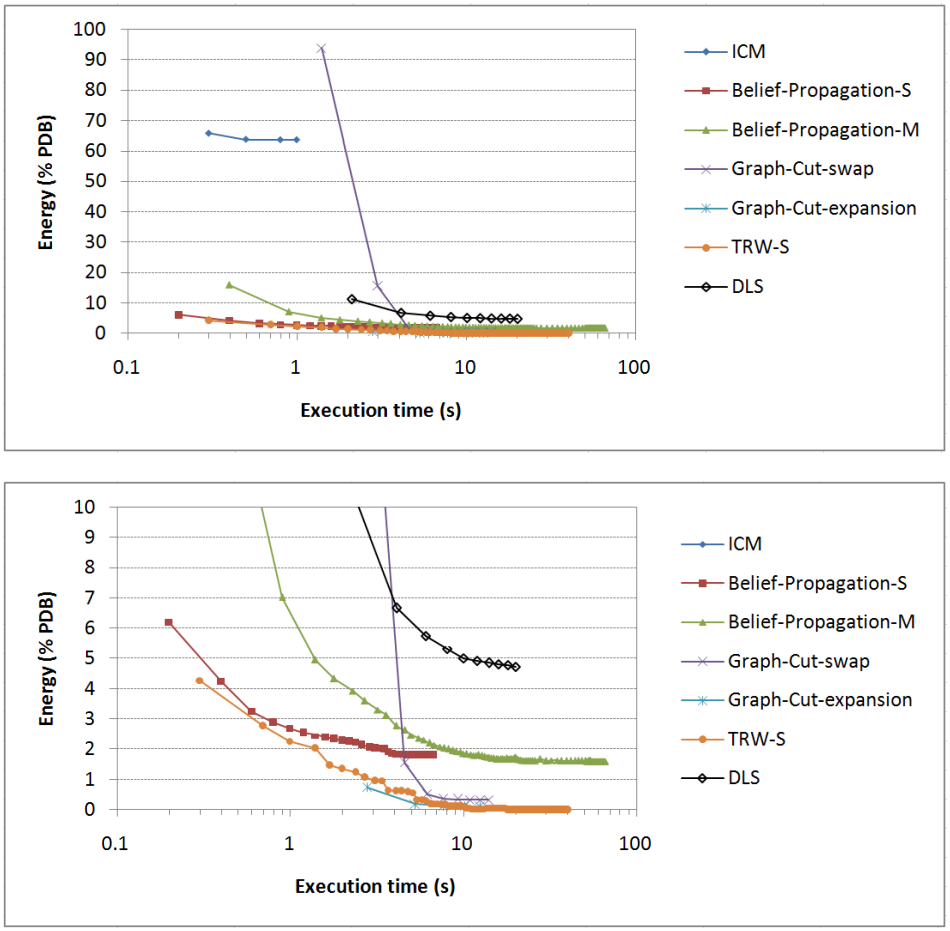


FIGURE 7.4: Different energy minimization methods on *Tsukuba* (384×288) stereo matching benchmark with large disparity range. The disparity range is set to 64 pixels. Note that the execution time is shown on the x-axis using a log scale. The lower chart is a zoom-in of the upper chart.

two graph cuts (GC) based algorithms proposed in [BVZ01] and implemented in [BK04]. For DLS, we use the operator combination of "propagation + random pixels expansion swap". We run DLS for 100 iterations and report both the energy and the execution time after each 10 iterations. Other methods have no preset number of iterations and they stop running after no lower energy can be found. Among these methods, results of ICM are reported after each 5 iterations, while results of the other methods are reported iteration by iteration. For each experiment group, the $\%PDB$ value is computed according to the lowest energy found among all the tested algorithms in the current experiment group.

We firstly report the trace execution for a small size benchmark *Tsukuba* (384×288) with a large (relatively large compared with image size) disparity range of 64 pixels. Results are plotted in Figure 7.4, where the lower chart is a zoom-in of the upper chart. The lowest energy is obtained by TRW-S. Here, DLS generates a relatively higher energy compared with other methods (except for ICM) but with $\%PDB$ value

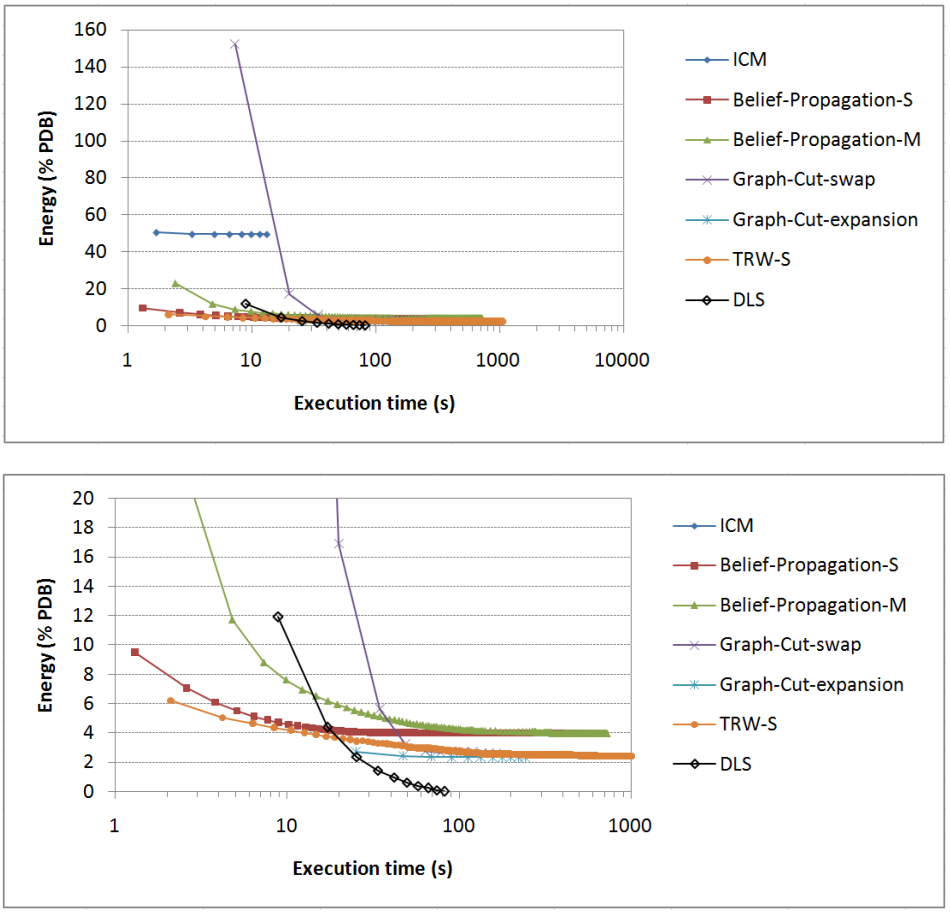


FIGURE 7.5: Different energy minimization methods on half size *Teddy* (900×750) stereo matching benchmark with small disparity range. The disparity range is set to 64 pixels. Note that the execution time is shown on the x-axis using a log scale. The lower chart is a zoom-in of the upper chart.

less than $\%5$. We then experiment on a large size benchmark *Teddy* (900×750) with a small (relatively small compared with image size) disparity range of 64 pixels. Results are plotted in Figure 7.5. In this case, the lowest energy is obtained by DLS. All the other methods generate higher energies with $\%PDB$ value larger than $\%2$. Afterwards, we experiment on the same *Teddy* (900×750) benchmark with a large disparity range of 128 pixels. At this time, the Belief-Propagation-S, Belief-Propagation-M, and TRW-S algorithms fail to run on our station[4], due to the large amount of memory needed for the large disparity range. But we report the results of other methods anyway in Figure 7.6, where Graph-Cut-expansion generates the lowest energy, and DLS generates a relatively higher energy compared with the other two graph cuts based methods (Graph-Cut-expansion and Graph-Cut-swap) but with $\%PDB$ value less that $\%5$. From the results, it seems that DLS can find lower energy within a small disparity range, when compared with other methods. Meanwhile, when operating on a large

[4]The detailed information of our experiment platforms is reported in the introduction section of Chapter 5.

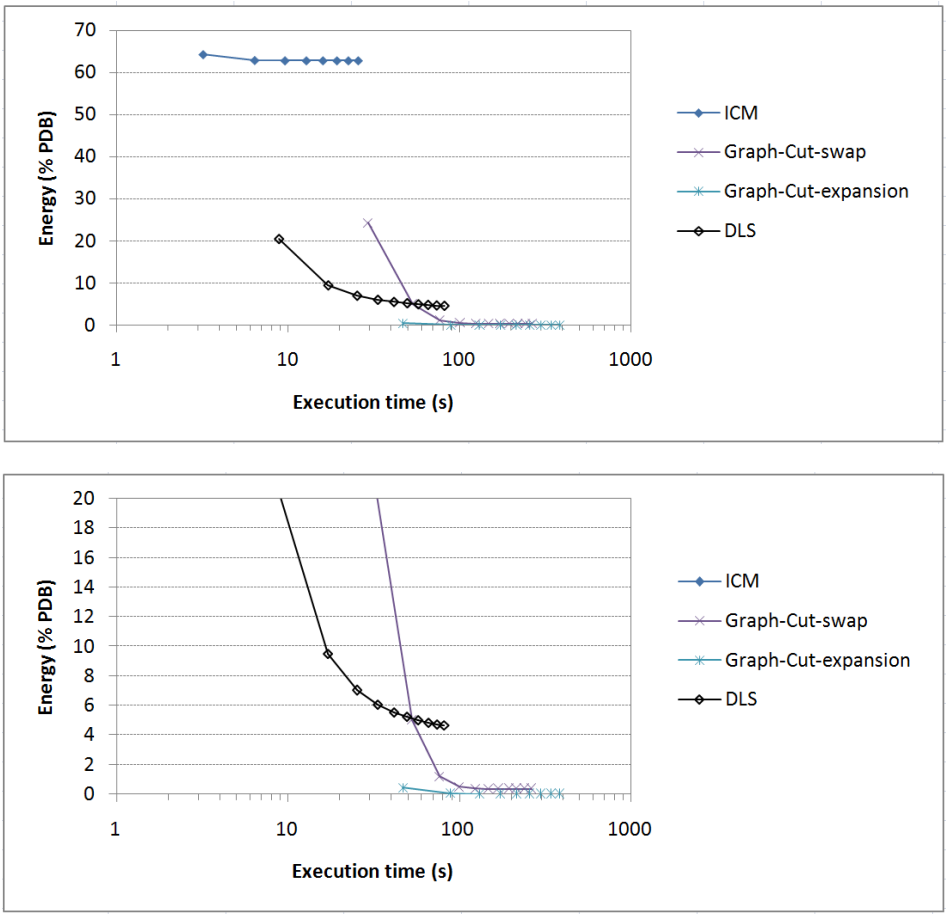
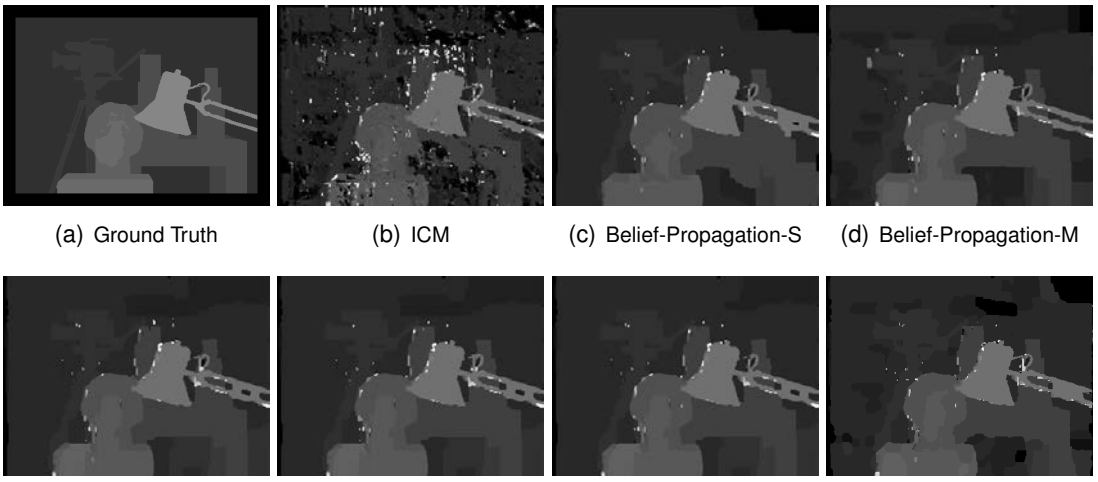


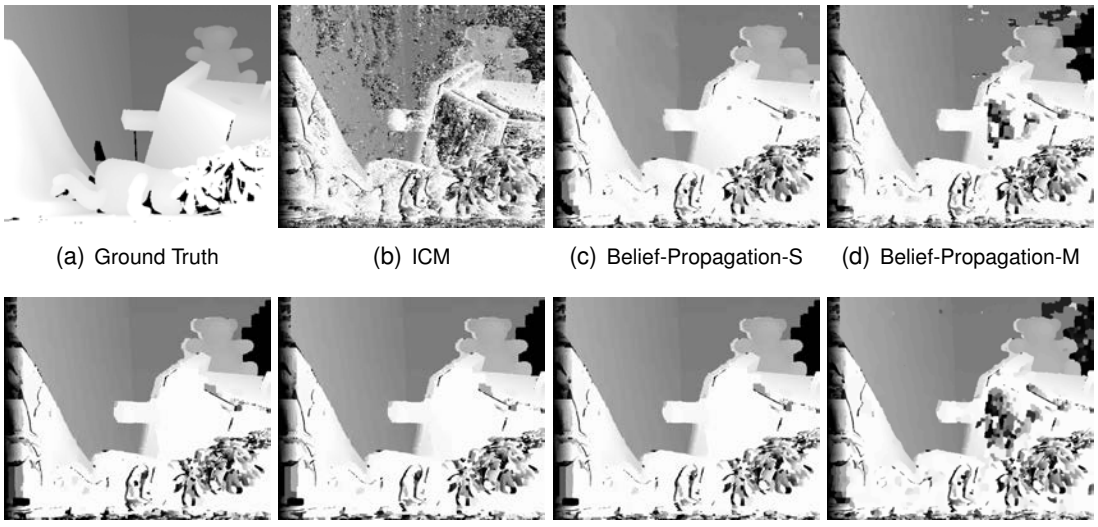
FIGURE 7.6: Different energy minimization methods on half size *Teddy* (900×750) stereo matching benchmark with large disparity range. The disparity range is set to 128 pixels. Note that the execution time is shown on the x-axis using a log scale. The lower chart is a zoom-in of the upper chart.



(a) Ground Truth (b) ICM (c) Belief-Propagation-S (d) Belief-Propagation-M

(e) Graph-Cut-Swap (f) Graph-Cut-Expansion (g) TRW-S (h) DLS

FIGURE 7.7: Disparity maps for *Tsukuba* (384×288) benchmark obtained with different energy minimization methods. The disparity range is set to 64 pixels.



(a) Ground Truth (b) ICM (c) Belief-Propagation-S (d) Belief-Propagation-M

(e) Graph-Cut-Swap (f) Graph-Cut-Expansion (g) TRW-S (h) DLS

FIGURE 7.8: Disparity maps for half size *Teddy* (900×750) benchmark obtained with different energy minimization methods. The disparity range is set to 64 pixels.

disparity range, there is still a gap between DLS and other state-of-the-art global methods for energy minimization, such as the graph cuts based algorithms and the belief propagation based algorithms.

In Figure 7.7 and Figure 7.8, are displayed the disparity maps for the two tested benchmarks. Note that during our experiments, we choose the stereo matching application but only view it as an energy minimization problem, just focusing on minimizing energies. The disparity maps obtained from all the tested methods are the raw results after energy minimization, without any additional post-treatments such as left-right consistency check, occlusion detection, or disparity smoothing, which are all treatments specific to stereo matching in order to minimize the errors compared with ground truth disparity maps. Moreover, as pointed out in [SZS+08], the ground truth solution may not always be strictly related to the lowest energy.

7.5 Acceleration Factors According to Problem Size

In this section, we focus on the performance of DLS when input size augments. We experiment on the *Teddy* benchmark with six sizes from the smallest size of 630×525 to the largest size of 1620×1350. We uniformly set the disparity range to 128 pixels, for all the sizes. We use DLS with the operator combination of "propagation + random pixels expansion swap". We denote this DLS implementation as DLS-GPU. We also test the counterpart CPU sequential version which is denoted by DLScpu. Moreover, we test ICM, Graph-Cut-swap, and Graph-Cut-expansion. For Belief-Propagation-S,

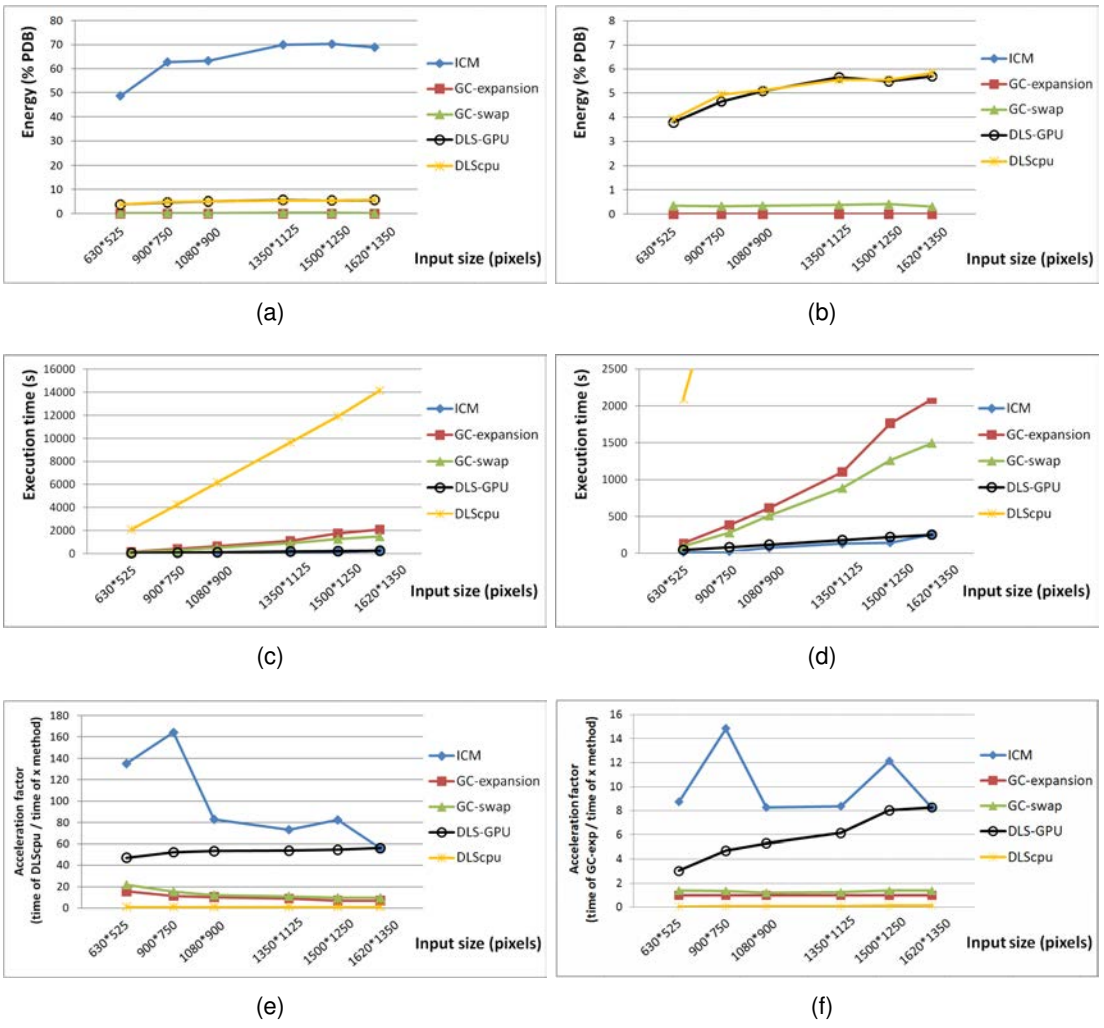


FIGURE 7.9: Stereo matching on *Teddy* benchmarks with different sizes. (b) and (d) are zoom-in parts of (a) and (c). The plot in (e) illustrates the acceleration factor computed according to the slowest method (DLScpu), while the plot in (f) illustrates the acceleration factor computed according to the method (GC-expansion) that gets the lowest energy. For the stop condition, the ICM method, and the graph cuts based methods (GC-expansion and GC-swap) run until convergence when no lower energy can be further found; the DLS methods (DLS-GPU and DLScpu) run for 100 iterations. The disparity range is set to 128 pixels.

Belief-Propagation-M, and TRW-S, they fail to run on our station with the image size of 900$\times$750 and larger, due to the large amount of memory needed for the disparity range with 128 pixels. Therefore, we do not consider these three methods in this section.

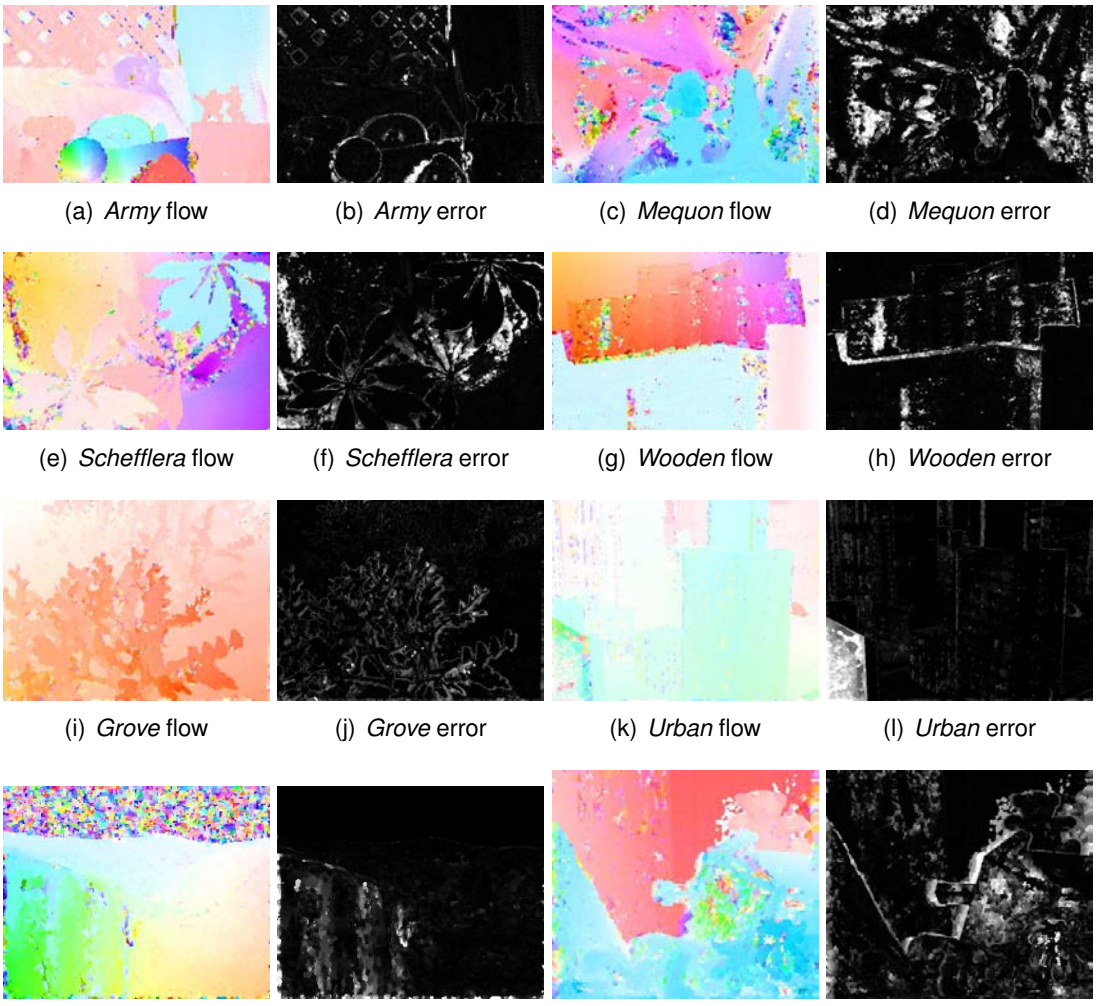
The results of different methods are reported in Figure 7.9. The charts in the second column of the figure correspond to zoom-in of the first column. The results of ICM, Graph-Cut-swap, and Graph-Cut-expansion are obtained after convergence (no lower energy can be further found); the results of DLS-GPU and DLScpu are obtained after 100 iterations. From top to bottom are reported the energy value as $\%PDB$, the execution time, and the acceleration factor of each method relative to

the slowest method (DLScpu) and the method (GC-expansion) that gets the lowest energy, respectively. For each method, the acceleration factor in Figure 7.9 (e) is computed by (execution time of DLScup / execution time); the acceleration factor in Figure 7.9 (f) is computed by (execution time of GC-expansion / execution time). The Graph-Cut-expansion (GC-expansion) method generates the lowest energies. Hence, all $\%PDB$ values are computed based on the lowest energy found by the Graph-Cut-expansion method. The ICM method runs fastest but generates very high energies with $\%PDB$ values in the range of 60%—70%. DLS-GPU runs a little slower than ICM but generates much lower energies with more acceptable $\%PDB$ values around 6%. An important observation is that, among all the tested methods, only the DLS-GPU has an acceleration factor which increases according to the augmentation of input size. This means that further improvement could be carried on only by the use of multi-processor platform with more effective cores. Whereas, we think that DLS with the required combination of operators looks promising in that it generates new intermediate compromises between time and quality. Recall that some methods have failed to be applied to large size images. Also, the acceleration factor for graph-cut methods over the DLScpu clearly decreases as the image size increases. Larger experiments on larger benchmark sets with large size images should be conducted near in the future.

7.6 Experimental Results on Optical Flow Benchmarks

In this section, we employ DLS for optical flow applications. In this application, the DLS is applied in the synchronized execution pattern, using only small move operators for fast computation. However, we use a complex energy function where the data term is a window based cost aggregation of the matching cost defined by Equation 2.17. For the smoothness term, we use a linear cost as the piecewise smooth prior and set the smoothness weight parameter λ (see Equation 2.12) to 0.5. We use DLS with the operator combination of "propagation + local move" under VNS framework. In order to obtain flow values with sub-pixel precision, we set the $scale$ parameter of *local move* operator to $1/8$, and access sub-pixel positions in input images through bilinear interpolation. We start DLS with random initialization where the flow value of each pixel is a random value within a certain range. For the occlusion detection, we apply a left-right cross checking procedure and use a weight median filter [HRB+13] to fill invalid pixels as post-processing. We test our method with the Middlebury data set [BSL+11] of optical flow[5]. The visualized flow value results are shown in Figure 7.10. Detailed evaluations compared with the top 5 optical flow methods in the Middlebury rank (up

[5] http://vision.middlebury.edu/flow/data/



(a) *Army* flow (b) *Army* error (c) *Mequon* flow (d) *Mequon* error

(e) *Schefflera* flow (f) *Schefflera* error (g) *Wooden* flow (h) *Wooden* error

(i) *Grove* flow (j) *Grove* error (k) *Urban* flow (l) *Urban* error

(m) *Yosemite* flow (n) *Yosemite* error (o) *Teddy* flow (p) *Teddy* error

FIGURE 7.10: DLS optical flow visualization results on Middlebury benchmarks. The error is average endpoint error evaluated by the Middlebury website.

to October 2015) are reported in Table 7.1 and Table 7.2. The average execution time of DLS for the eight tested benchmarks is 3.1 seconds on our GPU platform, compared with the average execution time of 573.07 seconds from the sequential CPU counterpart version, with an average acceleration factor of 173.

It is a well-known fact that the minimum energy level does not necessarily correlate to the best real-case matching. Here, we only address energy minimization discarding too much complex post-treatments necessary for the "true" ground truth matching. We conclude that a gap in quality remains to be covered with DLS applied to optical flow yet. In order to improve the matching quality in terms of minimizing the errors to ground truth only, specially designed terms for detecting typical situations in vision, such as occlusion, slanted surfaces, and the aperture problem, need to be added in the formulation of energy function. Furthermore, more complex post-treatments for invalid flow value fixing and smoothing should also be considered.

TABLE 7.1: Middlebury optical flow evaluation results (Part 1).

Average endpoint error	Army (Hidden texture)			Mequon (Hidden texture)			Schefflera (Hidden texture)			Wooden (Hidden texture)		
	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext
NNF-Local [Mid15a]	0.07	0.20	0.05	0.15	0.51	0.12	0.18	0.37	0.14	0.10	0.49	0.06
PMMST [Mid15a]	0.09	0.21	0.07	0.18	0.51	0.16	0.21	0.42	0.17	0.10	0.33	0.08
OFLAF [KLL13]	0.08	0.21	0.06	0.16	0.53	0.12	0.19	0.37	0.14	0.14	0.77	0.07
MDP-Flow2 [XJM12]	0.08	0.21	0.07	0.15	0.48	0.11	0.20	0.40	0.14	0.15	0.80	0.08
NN-field [CJL+13]	0.08	0.22	0.05	0.17	0.55	0.13	0.19	0.39	0.15	0.09	0.48	0.05
DLS	0.27	0.68	0.24	1.64	1.75	2.05	1.05	1.49	1.60	0.77	2.00	0.80

TABLE 7.2: Middlebury optical flow evaluation results (Part 2).

Average endpoint error	Grove (Synthetic)			Urban (Synthetic)			Yosemite (Synthetic)			Teddy (Stereo)		
	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext
NNF-Local [Mid15a]	0.41	0.61	0.21	0.23	0.66	0.19	0.10	0.12	0.17	0.34	0.80	0.23
PMMST [Mid15a]	0.51	0.74	0.28	0.24	0.65	0.20	0.11	0.12	0.17	0.37	0.74	0.35
OFLAF [KLL13]	0.51	0.78	0.25	0.31	0.76	0.25	0.11	0.12	0.21	0.42	0.78	0.63
MDP-Flow2 [XJM12]	0.63	0.93	0.43	0.26	0.76	0.23	0.11	0.12	0.17	0.38	0.79	0.44
NN-field [CJL+13]	0.41	0.61	0.20	0.52	0.64	0.26	0.13	0.13	0.20	0.35	0.83	0.21
DLS	1.15	1.57	1.07	2.45	2.68	1.47	0.46	0.32	1.21	2.26	2.86	4.28

7.7 Conclusion

We have experimented on the DLS algorithm applied to visual correspondence applications including stereo matching and optical flow. The main encouraging result is that the DLS on stereo matching seems to be the only method that provides an increasing acceleration factor as the instance size augments, for a result of quality about 4%—6% deviation to the best known energy value. For all the other approaches, the acceleration factor, against the slowest sequential version of DLS, is decreasing, except for the ICM method, which however only produces poor result of about 70% deviation to the best known energy. Graph cuts based algorithms and belief propagation based algorithms are well-performing approaches concerning quality, however the computation time increases quickly along with the instance size. That is why we hope for further improvements or improved accelerations of the DLS approach with the availability of new multi-processor platforms with more independent cores. Results of optical flow are evaluated according to ground truth, not the energy value. It should follow that many tricks are certainly not yet implemented to make energy minimization coincide to ground truth evaluation. Results on more benchmarks should be carried on, together with better adjustment of the method to ground truth evaluation and related modified energy functions.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Parallel implementation models based on data duplication look adapted to standard peer-to-peer multi-processors or networks of workstations, since the level of granularity allows communications by message passing. According to the data decomposition paradigm, GPU platform looks more suited to the low level of granularity and the many local interactions that take place, because all processors (threads) have direct accesses to global memory with high throughput. In this thesis, we have tried to implement models of "data decomposition, control decentralized, global memory" type, on GPU platforms. Certainly, granularity of the parallel computation depends on many compromises between the complexity of the treatments and the requirement for non divergent codes executed on GPU. The CPU side plays a central role in the management of kernel calls. However, we have paid a particular attention so that most of the optimization operations are done in parallel. The model we have proposed is called cellular matrix and it partitions data, defines the level of computation granularity, and allows generic and systematic association of processing units to the data deployed in the Euclidean plane. Based on the cellular matrix model, we can perform efficient spiral search with constant time $O(1)$ in average for uniform distribution.

We have provided a generic framework for Euclidean grid matching problems based on the cellular matrix model. We have presented topological *k*-means problems formulated in the framework, and the related parallel *k*-means procedures of the SOM algorithm. We have proposed the *superpixel adaptive segmentation map* (SPASM) algorithm, as a superpixel image segmentation algorithm, which combines the parallel on-line SOM structured mesh generation and the parallel batch SOM clustering. The SPASM algorithm can generate segmentation where the distribution/density of superpixels

coincides with the distribution of some specified attribute of input image, such as edges, textures, and depths.

We have also proposed the *distributed local search* (DLS) algorithm and applied it to visual correspondence applications. In this case, the considered pixel labeling problems can be viewed as Euclidean grid matching problems and formalized under an energy minimization framework. Then, we have dealt with the problems as combinatorial optimization problems by minimizing the energy function through the proposed DLS algorithm. In DLS, classical drawbacks to address are related to cell frontier management and solution diversification. In order to eliminate the conflicting operations near cell frontiers, we have proposed two execution patterns for the DLS algorithm: *synchronized execution* pattern and *dynamic change of cell frontiers* pattern. In order to enhance the solution diversification, we have designed different neighborhood operators including two small move operators and six large move operators, so that we can jointly use multiple operators in the DLS algorithm, combing them under a *variable neighborhood search* (VNS) framework.

We have performed experiments on GPU platform with CUDA implementations. For all the three *k*-means applications, our GPU parallel versions always produce substantial acceleration over their sequential counterparts. We have evaluated our GPU implementation of the parallel SOM TSP application, on different large-size TSP instances from different benchmarks. These instances include 33 TSPLIB [Rei91] instances with sizes from 1000 cities to 85900 cities and 19 National TSP's [nat09] with sizes from 1621 cities to 71009 cities. Our GPU SOM implementations run a lot faster than the CPU memetic SOM and Co-Adaptive Net, which are two neural network methods in the literature that perform experimental studies on large size benchmark instances. The execution time of our GPU implementation increases very slowly compared to either memetic SOM or Co-Adaptive Net, when the input instance size augments.

We have conducted experiments on our GPU implementation of the structured meshing application and its counterpart CPU implementation. The execution time of GPU version for all six tested images is steady and little. The acceleration factor, which is the ratio of CPU version's execution time by GPU, varies from factor 5 to 8. We have also studied the relationship between execution time and input size of GPU implementation and its counterpart CPU version, by carrying out experiments with four disparity maps at small, medium and large scales, respectively. The average acceleration factors of the three sets are 5, 12 and 39 respectively, as input size grows, which indicate augmentation of acceleration factor with input size.

We have tested our SPASM algorithm utilizing two kinds of image attributes as the density distributions for cluster center initialization with the on-line SOM: image gradient and disparity value. The SPASM algorithm's ability of generating adaptive segmentation with respect to user-specified density distribution has been demonstrated through these two tests and the comparison of their results. We have compared the SPASM algorithm with the state-of-the-art SLIC algorithm, on seven input images of growing sizes, setting different initial superpixel sizes. We have tested both GPU version and CPU version for both two algorithms. Compared with others, the execution time of GPU SPASM increases in a linear way with a very weak increasing coefficient, when the input image size augments. The acceleration factor of GPU SPASM against its CPU sequential counterpart version is 39 in average for the seven tested images.

For DLS applications, we have carried out experiments on standard benchmarks and in comparison with state-of-the-art energy minimization approaches. Among all the tested methods, only DLS has an acceleration factor (either against the slowest sequential version of DLS or against the graph cuts based method that gets the lowest energy) which is increasing according to the augmentation of input size, for a result of quality about 4%—6% deviation to the best known energy value. We think that the results imply the potentiality of the approach to substantially improve its performance in the future as more physical cores will be available in GPU. We have applied DLS to optical flow applications and tested eight benchmarks from the Middlebury data set. The average execution time is about 3 seconds on our GPU platform, compared with the average execution time of 573 seconds of the CPU sequential counterpart version, with an average acceleration factor of 173.

8.2 Future Work

Several future works can be considered as extensions of this thesis. The potential of DLS could be further verified through systematical experiments on large size benchmarks, as a continuation of this work on the visual correspondence problems. Particularly, we will concentrate on the optical flow experiments for which a large set of operators have been designed in the DLS framework.

We expect that the DLS approach could yield a new standard metaheuristic based on parallel variable neighborhood search. An attention will be paid to the development of robust and generic software with object orienting programming (OOP) and meta-programming, following the continuation of the work already done for the cellular matrix in different topologies, using functors and template codes.

The cellular matrix model could be further improved by the introduction of dynamic load balancing properties. It could be possible, for example, to compute adapted cells by *k*-means tools for balanced covering of brute data. Also, recursive decomposition for computation reduction should be addressed by the means of plane decomposition with recursive grids, in a way similar to quad-tree, in different topologies. This could allow the use of new compute capability GPUs with recursive kernel calls.

As standard tools for image processing and vision, parallel *k*-means and elastic matching approaches could be combined in different ways to address more complex and composite problems. For example, a possible formulation could be double-matching, using bi-directional projection with two grids with both variables and inputs. Also, different sequential steps in image processing could be merged into more integrated sequences dealing with grids of same nature. For example, one could compute disparity maps at the same time as generating some compressed representations with superpixels, meshing or clustering.

Implementations on different platforms such as *clusters*, and possibly configurable systems as *field-programmable gate array* (FPGA), or specific systems as *application-specific integrated circuit* (ASIC), and other *system on chip* (SoC) architectures, could be envisaged in the future. These applications could allow better matching of the algorithms and data structures based on our conceptual cellular matrix model, to hardware architectures in the future.

Appendix A

Experimental Results

TABLE A.1: Experimental results of 33 TSPLIB instances with sizes larger than 1000 cities. (Part 1)

		GPU-SOM-1[a]			GPU-SOM-2[b]			Memetic SOM		
Instance	Optimal	%PDB	%PDM	Sec	%PDB	%PDM	Sec	%PDB	%PDM	Sec[c]
dsj1000	18659688	6.52	7.90	9.76	5.96	6.81	19.92	4.73	6.46	61.31
pr1002	259045	4.52	6.15	6.15	4.60	5.70	11.60	4.17	4.78	55.76
u1060	224094	5.39	6.76	6.19	5.70	6.05	6.82	4.04	5.12	67.94
vm1084	239297	6.57	8.29	4.02	6.41	7.99	8.58	4.95	5.86	68.53
pcb1173	56892	7.77	8.63	5.64	6.91	7.67	6.98	6.97	7.50	70.53
d1291	50801	10.50	12.81	4.99	11.13	12.62	7.85	7.54	9.66	81.58
rl1304	252948	9.29	12.99	6.04	9.44	12.85	12.32	7.76	10.00	78.64
rl1323	270199	9.16	12.19	6.88	9.40	11.27	14.76	8.80	9.45	82.52
nrw1379	56638	5.82	6.43	6.25	4.07	4.72	11.12	3.96	4.61	80.42
fl1400	20127	15.84	23.68	1.33	8.59	18.15	1.82	3.46	4.32	234.54
u1432	152970	6.58	7.19	7.78	5.93	6.62	19.67	4.07	5.02	96.78
fl1577	22249	15.02	18.92	1.61	10.72	15.39	2.97	15.87	17.46	107.54
d1655	62128	10.43	11.74	6.28	7.33	10.86	10.14	8.00	9.60	99.40
vm1748	336556	6.86	7.76	5.02	7.33	8.13	14.18	5.30	6.68	132.86
u1817	57201	11.06	11.70	4.80	9.59	10.86	10.82	8.25	9.68	125.03
rl1889	316536	8.38	11.35	5.01	9.92	11.30	15.75	8.56	9.54	129.02
d2103	80450	19.08	20.28	6.28	17.84	19.17	12.92	15.54	19.15	132.50
Average		10.52	12.69	32.26	8.55	10.20	382.52	7.71	8.83	1294.72
Average	$(N \geq 10000)$	13.09	15.92	127.25	9.36	10.05	1718.56	8.35	8.77	5426.05

[a] SSR=10. [b] SSR=*infinite*.

[c] Time per run in AMD Athlon (2000MHz) seconds. [d] Best known solution.

TABLE A.2: Experimental results of 33 TSPLIB instances with sizes larger than 1000 cities. (Part 2)

		GPU-SOM-1[a]			GPU-SOM-2[b]			Memetic SOM		
Instance	Optimal	%PDB	%PDM	Sec	%PDB	%PDM	Sec	%PDB	%PDM	Sec[c]
u2152	64253	10.91	11.76	4.89	9.41	10.54	11.34	8.37	10.43	144.56
u2319	234256	3.00	3.37	10.83	2.80	3.33	43.49	1.50	1.72	191.37
pr2392	378032	7.67	8.23	8.68	7.01	8.03	30.60	6.32	7.04	161.54
pcb3038	137694	8.07	8.63	7.31	7.49	7.92	24.28	7.10	7.88	195.26
fl3795	28772	20.17	25.24	1.89	7.25	14.58	2.41	13.66	16.13	389.22
fnl4461	182566	6.00	6.80	9.98	4.72	5.08	34.35	5.13	5.62	330.24
rl5915	565530	14.03	15.56	9.06	13.14	14.01	51.39	12.02	12.94	465.58
rl5934	556045	14.10	16.01	9.43	12.72	13.32	52.41	11.68	13.02	478.35
pla7397	23260728	12.68	17.05	17.77	11.38	13.41	154.92	9.11	10.19	682.31
rl11849	923288	15.33	17.50	14.15	11.60	12.24	154.97	10.71	11.49	1234.15
usa13509	19982859	18.35	24.61	25.78	10.22	11.28	162.73	7.11	7.62	1579.27
brd14051	469388[d]	7.46	8.08	10.50	5.59	6.22	65.43	5.98	6.18	1459.43
d15112	1573084	8.14	8.68	17.09	6.54	6.84	138.94	5.56	5.95	1802.36
d18512	645244[d]	7.51	8.77	14.57	5.69	5.90	101.53	5.67	6.00	2084.02
pla33810	66050535[d]	15.60	18.12	68.64	14.11	15.56	3410.98	12.83	13.23	4788.57
pla85900	142383704[d]	19.24	25.65	739.99	11.74	12.32	7995.31	10.51	10.94	25034.37
Average		10.52	12.69	32.26	8.55	10.20	382.52	7.71	8.83	1294.72
Average	$(N \geq 10000)$	13.09	15.92	127.25	9.36	10.05	1718.56	8.35	8.77	5426.05

[a] SSR=10. [b] SSR=*infinite*.

[c] Time per run in AMD Athlon (2000MHz) seconds. [d] Best known solution.

TABLE A.3: Experimental results of 19 National TSPs with sizes larger than 1000 cities.

		GPU-SOM-1[a]			GPU-SOM-2[b]			Memetic SOM		
Instance	Optimal	%PDB	%PDM	Sec	%PDB	%PDM	Sec	%PDB	%PDM	Sec[c]
rw1621	26051	7.49	10.14	2.34	6.72	8.77	4.94	5.00	6.79	121.80
mu1979	86891	7.72	10.41	4.14	7.24	8.99	6.61	5.23	6.36	212.13
nu3496	96132	10.56	13.24	4.77	8.50	9.58	12.42	7.24	8.38	302.12
tzt6117	394718[d]	9.03	10.40	6.84	8.85	9.60	28.52	7.17	7.90	563.04
eg7146	172387[d]	12.86	15.39	8.25	8.61	11.52	16.69	6.08	7.84	1150.29
ym7663	238314[d]	11.70	20.72	8.04	9.38	10.73	19.69	7.44	8.45	829.82
pm8079	114857[d]	12.26	16.12	5.33	9.62	11.15	16.39	8.51	9.33	848.53
ei8246	206171[d]	6.99	9.97	7.94	6.65	7.22	32.11	6.55	6.84	713.90
ar9152	837520[d]	17.61	21.91	12.31	10.75	12.02	40.75	7.94	9.05	1036.86
ja9847	491924	14.10	23.48	19.18	13.86	15.41	23.03	8.94	9.41	1213.84
gr9882	300899	8.86	10.33	8.05	7.85	8.94	29.69	6.82	7.17	1017.28
kz9976	1061881[d]	10.16	11.92	11.87	9.49	11.85	57.48	7.18	7.72	1015.72
fi10639	520527[d]	8.76	10.87	9.09	7.59	7.96	47.54	6.66	6.93	1035.77
mo14185	427377[d]	8.42	8.75	11.43	7.56	8.11	45.92	6.49	6.95	1584.29
ho14473	177132[d]	14.13	23.83	6.77	13.01	13.58	28.55	7.54	7.98	1787.88
vm22775	569288[d]	14.18	17.15	17.71	10.24	13.00	50.17	7.14	7.75	3037.55
sw24978	855610[d]	9.51	10.81	14.72	8.25	9.09	101.40	7.06	7.39	3398.49
bm33708	959304[d]	9.44	10.13	21.18	7.92	8.42	105.56	6.84	7.41	5377.23
ch71009	4566578[d]	19.57	27.65	39.62	12.08	12.79	396.09	6.97	7.16	22193.83
Average		11.23	14.87	11.56	9.17	10.46	55.98	5.88	6.63	1905.47
Average	$(N \geq 10000)$	12.00	15.60	17.22	9.52	10.42	110.75	6.96	7.37	5487.86

[a] SSR=10. [b] SSR=*infinite*.

[c] Time per run in AMD Athlon (2000MHz) seconds. [d] Best known solution.

TABLE A.4: Experimental results of four sets of disparity maps at different scales.

input image	image size	matcher grid size (RC)	cellular matrix size (R)	SSR	iterations[g]	time (s)		%cost		
						GPU	CPU	GPU	CPU	AF[1]
tsukuba	384×288	64×48 (6)	20×15 (20)	3	1500	0.32	1.87	25.89	25.02	5.84
rocks1	425×370	71×62 (6)	22×19 (20)	3	1500	0.34	2.28	17.94	18.73	6.71
aloe	427×370	71×62 (6)	22×19 (20)	3	1500	0.30	2.50	27.15	28.70	8.33
venus	434×383	72×64 (6)	22×20 (20)	3	1500	0.32	2.40	21.55	19.62	7.50
teddy	450×375	75×63 (6)	23×19 (20)	3	1500	0.36	2.41	18.45	20.04	6.69
cones	450×375	75×63 (6)	23×19 (20)	3	1500	0.35	2.45	17.24	16.69	7.00
Average	**154926**					**0.33**	**2.32**	**21.37**	**21.47**	**7.03**
rocks1S	425×370	71×62 (6)	22×19 (20)	4	1500	0.66	3.32	20.03	18.91	5.03
aloeS	427×370	71×62 (6)	22×19 (20)	4	1500	0.51	3.56	27.45	28.02	6.98
conesS	450×375	75×63 (6)	23×19 (20)	4	1500	0.67	3.52	18.95	16.59	5.25
teddyS	450×375	75×63 (6)	23×19 (20)	4	1500	0.69	3.45	18.19	19.91	5.00
Average	**163185**					**0.63**	**3.46**	**21.16**	**20.86**	**5.49**
rocks1M	638×555	106×93 (6)	32×28 (20)	4	1500	1.05	8.33	21.65	18.37	7.93
aloeM	641×555	107×93 (6)	33×28 (20)	4	1500	0.57	9.36	26.39	26.90	16.42
conesM	900×750	150×125 (6)	45×38 (20)	4	1500	1.36	18.52	18.75	17.82	13.62
teddyM	900×750	150×125 (6)	45×38 (20)	4	1500	1.35	18.53	21.07	19.26	13.73
Average	**514961**					**1.08**	**13.69**	**21.97**	**20.59**	**12.68**
rocks1L	1276×1110	213×185 (6)	64×56 (20)	4	1500	2.18	47.36	21.73	19.53	21.72
aloeL	1282×1110	214×185 (6)	65×56 (20)	4	1500	0.95	53.19	25.64	24.72	55.99
conesL	1800×1500	300×250 (6)	90×75 (20)	4	1500	2.74	107.10	17.27	25.93	39.09
teddyL	1800×1500	300×250 (6)	90×75 (20)	4	1500	2.13	110.25	25.98	23.63	51.76
Average	**2059845**					**2.00**	**79.48**	**22.66**	**23.45**	**39.74**

[1]acceleration factor. [g]GPU version.

Appendix B

Publications

Journal

1. *Parallel Structured Mesh Generation with Disparity Maps by GPU Implementation*
   Hongjian WANG, Naiyu ZHANG, Jean-Charles CREPUT, Julien MOREAU, and Yassine RUICHEK
   IEEE Transactions on Visualization & Computer Graphics, vol.21, no. 9, pp. 1045-1057, Sept. 2015 (IF: 2.168)

Conference

1. *Massively Parallel Cellular Matrix Model for Self-organizing Map Applications*
   Hongjian WANG, Abdelkhalek MANSOURI, and Jean-Charles CREPUT
   Proc. of IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2015), Cairo, Egypt, Dec. 06-09, 2015

2. *Massively Parallel Cellular Matrix Model for SuperpixelAdaptive Segmentation Map*
   Hongjian WANG, Abdelkhalek MANSOURI, Jean-Charles CREPUT, and Yassine RUICHEK
   Proc. of 14th Mexican International Conference on Artificial Intelligence (MICAI 2015), Cuernavaca, Morelos, Mexico, Oct. 25-31, 2015 Advances in Artificial Intelligence and Its Applications, Lecture Notes in Computer Science, vol.9414, pp.325-336, 2015

3. *Parallel and Distributed Implementation Models for Bio-Inspired Optimization Algorithms*

Hongjian WANG, and Jean-Charles CREPUT
Proc. of First International Conference on Swarm Intelligence Based Optimization (ICSIBO 2014), Mulhouse, France, May 13-14, 2014
Swarm Intelligence Based Optimization, Lecture Notes in Computer Science, vol. 8472, pp. 68-79, 2014

4. *A Massive Parallel Cellular GPU Implementation of Neural Network to Large Scale Euclidean TSP*
Hongjian WANG, Naiyu ZHANG, and Jean-Charles CREPUT
Proc. of 12th Mexican International Conference on Artificial Intelligence (MICAI 2013), Mexico City, Mexico, Nov. 24-30, 2013 (**Best Student Paper Award**)
Advances in Soft Computing and Its Applications, Lecture Notes in Computer Science, vol. 8266, pp.118-129, 2013

5. *Cellular GPU Model for Structured Mesh Generation and Its Application to the Stereo-Matching Disparity Map*
Naiyu ZHANG, Hongjian WANG, Jean-Charles CREPUT, Julien MOREAU, and Yassine RUICHEK
Proc. of IEEE International Symposium on Multimedia (ISM 2013), Anaheim, California, USA, 2013

6. *A Near Real-Time Color Stereo Matching Method for GPU*
Naiyu ZHANG, Hongjian WANG, Jean-Charles CREPUT, Julien MOREAU, and Yassine RUICHEK
Proc. of Third International Conference on Advanced Communications and Computation (INFOCOMP 2013), Lisbon, Portugal, 2013

7. *Partial Demosaicing for Stereo Matching of CFA Images on GPU and CPU*
Naiyu ZHANG, Jean-Charles CREPUT, Hongjian WANG, Cyril MEURIE, and Yassine RUICHEK
Proc. of Third International Conference on Advanced Communications and Computation (INFOCOMP 2013), Lisbon, Portugal, 2013

Bibliography

[AK96] David Andre and John R Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In *Advances in genetic programming*, pages 317–337. MIT Press, 1996.

[ASS+12] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Susstrunk. Slic superpixels compared to state-of-the-art superpixel methods. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(11):2274–2282, 2012.

[AT02] Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 6(5):443–462, 2002.

[Bar89] Stephen T Barnard. Stochastic stereo matching over scale. *International Journal of Computer Vision*, 3(1):17–32, 1989.

[Ben02] Endika Bengoetxea. *Inexact Graph Matching Using Estimation of Distribution Algorithms*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec 2002.

[Bes86] Julian Besag. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 259–302, 1986.

[BK04] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(9):1124–1137, 2004.

[BLS13] Ilhem BoussaïD, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, 2013.

[BOL+09] Hongtao Bai, Dantong OuYang, Ximing Li, Lili He, and Haihong Yu. Max-min ant system on gpu with cuda. In *Innovative Computing, Information*

and Control, 2009 Fourth International Conference on, pages 801–804. IEEE, 2009.

[BSL+11] Simon Baker, Daniel Scharstein, JP Lewis, Stefan Roth, Michael J Black, and Richard Szeliski. A database and evaluation methodology for optical flow. *International Journal of Computer Vision*, 92(1):1–31, 2011.

[BvdM87] Elie Bienenstock and Christoph von der Malsburg. A neural network for invariant pattern recognition. *EPL (Europhysics Letters)*, 4(1):121, 1987.

[BVZ98] Yuri Boykov, Olga Veksler, and Ramin Zabih. Markov random fields with efficient approximations. In *Computer Vision and Pattern Recognition, 1998 IEEE Conference on*, pages 648–655. IEEE, 1998.

[BVZ01] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(11):1222–1239, 2001.

[BWY80] Jon Louis Bentley, Bruce W Weide, and Andrew C Yao. Optimal expected-time algorithms for closest point problems. *ACM Transactions on Mathematical Software (TOMS)*, 6(4):563–580, 1980.

[CB03] EM Cochrane and JE Beasley. The co-adaptive neural network approach to the euclidean travelling salesman problem. *Neural Networks*, 16(10):1499–1525, 2003.

[CHKK12] Jean-Charles Créput, Amir Hajjam, Abderrafiaa Koukam, and Olivier Kuhn. Self-organizing maps in population based metaheuristic to the dynamic vehicle routing problem. *Journal of Combinatorial Optimization*, 24(4):437–458, 2012.

[CHMR87] James P Cohoon, Shailesh U Hegde, Worthy N Martin, and D Richards. Punctuated equilibria: a parallel genetic algorithm. In *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*. Hillsdale, NJ: L. Erlhaum Associates, 1987., 1987.

[CJL+13] Zhuoyuan Chen, Hailin Jin, Zhe Lin, Sholom Cohen, and Ying Wu. Large displacement optical flow from nearest neighbor fields. In *Computer Vision and Pattern Recognition, 2013 IEEE Conference on*, pages 2443–2450. IEEE, 2013.

[CK09] Jean-Charles Créput and Abderrafiãa Koukam. A memetic neural network for the euclidean traveling salesman problem. *Neurocomputing*, 72(4):1250–1264, 2009.

[CMC+09] Tibério S Caetano, Julian J McAuley, Li Cheng, Quoc V Le, and Alex J Smola. Learning graph matching. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(6):1048–1058, 2009.

[CSS07] Timothee Cour, Praveen Srinivasan, and Jianbo Shi. Balanced graph matching. *Advances in Neural Information Processing Systems*, 19:313, 2007.

[CT03] Teodor Gabriel Crainic and Michel Toulouse. *Parallel strategies for meta-heuristics*. Springer, 2003.

[CT10] Teodor Gabriel Crainic and Michel Toulouse. Parallel meta-heuristics. In *Handbook of metaheuristics*, pages 497–541. Springer, 2010.

[DIM09] DIMACS TSP Challenge, 2009. http://dimacs.rutgers.edu/Challenges/TSP/.

[Dor92] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, 1992.

[DW87] Richard Durbin and David Willshaw. An analogue approach to the travelling salesman problem using an elastic net method. *Nature*, 326(16):689–691, 1987.

[FF62] LR Ford and Delbert Ray Fulkerson. *Flows in networks*, volume 1962. Princeton Princeton University Press, 1962.

[FH06] Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient belief propagation for early vision. *International Journal of Computer Vision*, 70(1):41–54, 2006.

[FL00] Alex A Freitas and Simon H Lavington. Data parallelism, control parallelism, and related issues. In *Mining Very Large Databases with Parallel Processing*, pages 71–78. Springer, 2000.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[FPS03] Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano. A scalable cellular implementation of parallel genetic programming. *Evolutionary Computation, IEEE Transactions on*, 7(1):37–53, 2003.

[GG84] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):721–741, 1984.

[GG15] Fatma Güney and Andreas Geiger. Displets: Resolving stereo ambiguities using object knowledge. In *Computer Vision and Pattern Recognition, 2015 IEEE Conference on*, pages 4165–4175, 2015.

[HC03] Karsten Hartelius and Jens Michael Carstensen. Bayesian grid matching. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(2):162–173, 2003.

[HM01] Pierre Hansen and Nenad Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449–467, 2001.

[HRB+13] Asmaa Hosni, Christoph Rhemann, Michael Bleyer, Carsten Rother, and Margrit Gelautz. Fast cost-volume filtering for visual correspondence and beyond. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(2):504–511, 2013.

[HS81] Berthold K Horn and Brian G Schunck. Determining optical flow. In *1981 Technical symposium east*, pages 319–331. International Society for Optics and Photonics, 1981.

[JM07] David S Johnson and Lyle A McGeoch. Experimental analysis of heuristics for the stsp. In *The traveling salesman problem and its variations*, pages 369–443. Springer, 2007.

[KLL13] Tae Hyun Kim, Hee Seok Lee, and Kyoung Mu Lee. Optical flow via locally adaptive fusion of complementary data costs. In *Computer Vision, 2013 IEEE International Conference on*, pages 3344–3351. IEEE, 2013.

[Koh82] Teuvo Kohonen. Clustering, Taxonomy, and Topological Maps of Patterns. *Pattern Recognition, 1982 Sixth International Conference on*, October 1982.

[Kon04] Zdenek Konfrst. Parallel genetic algorithms: Advances, computing trends, applications and perspectives. In *Parallel and Distributed Processing Symposium, 2004 18th International*, page 162. IEEE, 2004.

[KT15] Ryan Kennedy and Camillo J Taylor. Hierarchically-constrained optical flow. In *Computer Vision and Pattern Recognition, 2015 IEEE Conference on*, pages 3340–3348, 2015.

[KTP00] Constatine Kotropoulos, Anastasios Tefas, and Ioannis Pitas. Frontal face authentication using morphological elastic graph matching. *Image Processing, IEEE Transactions on*, 9(4):555–560, 2000.

[KU03] Daniel Keysers and Walter Unger. Elastic image matching is np-complete. *Pattern Recognition Letters*, 24(1):445–453, 2003.

[LL00] Raymond ST Lee and James NK Liu. Tropical cyclone identification and tracking system using integrated neural oscillatory elastic graph matching and hybrid rbf network track mining techniques. *Neural Networks, IEEE Transactions on*, 11(3):680–689, 2000.

[LYMD13] Jiangbo Lu, Hongsheng Yang, Dongbo Min, and Minh N Do. Patch match filter: Efficient edge-aware filtering meets randomized search for fast correspondence field estimation. In *Computer Vision and Pattern Recognition, 2013 IEEE Conference on*, pages 1854–1861. IEEE, 2013.

[ME07] Tomasz Malisiewicz and Alexei A Efros. Improving spatial support for objects via multiple segmentations. In *British Machine Vision Conference (BMVC)*, 2007.

[MG15] Moritz Menze and Andreas Geiger. Object scene flow for autonomous vehicles. In *Computer Vision and Pattern Recognition, 2015 IEEE Conference on*, pages 3061–3070, 2015.

[MH97] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[Mid15a] Middlebury. Middlebury Optical Flow Datasets., 2015. http://vision.middlebury.edu/flow/.

[Mid15b] Middlebury. Middlebury Stereo Datasets., 2015. http://vision.middlebury.edu/stereo/.

[Mla95] Nenad Mladenovic. A variable neighborhood algorithm-a new metaheuristic for combinatorial optimization. In *papers presented at Optimization Days*, page 112, 1995.

[MP76] David Marr and Tomaso Poggio. Cooperative computation of stereo disparity. *Science*, 194(4262):283–287, 1976.

[MS89] Bernard Manderick and Piet Spiessens. Fine-grained parallel genetic algorithms. In *Genetic algorithms, 1989 Third International Conference on*, pages 428–433. Morgan Kaufmann Publishers Inc., 1989.

[MSH+12] Sabine McConnell, Robert Sturgeon, Gregory Henry, Andrew Mayne, and Richard Hurley. Scalability of self-organizing maps on a gpu cluster using opencl and cuda. In *Journal of Physics: Conference Series*, volume 341, page 012018. IOP Publishing, 2012.

[nat09] National Travelling Salesman Problems, 2009. www.math.uwaterloo.ca/tsp/world/countries.html.

[NVI12] NVIDIA. CUDA C Programming Guide 4.2, CURAND Library, CUDPP library, Profiler User's Guide., 2012. http://docs.nvidia.com/cuda.

[NYYY07] Hung Dinh Nguyen, Ikuo Yoshihara, Kunihito Yamamori, and Moritoshi Yasunaga. Implementation of an effective hybrid ga for large-scale traveling salesman problems. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 37(1):92–99, 2007.

[OL96] Ibrahim H Osman and Gilbert Laporte. Metaheuristics: A bibliography. *Annals of Operations research*, 63(5):511–623, 1996.

[Pap77] Christos H Papadimitriou. The euclidean travelling salesman problem is np-complete. *Theoretical Computer Science*, 4(3):237–244, 1977.

[PC10] Martin Pedemonte and Hector Cancela. A cellular ant colony optimisation for the generalised steiner problem. *International Journal of Innovative Computing and Applications*, 2(3):188–201, 2010.

[Pea14] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014.

[PNC11] Martín Pedemonte, Sergio Nesmachnow, and Héctor Cancela. A survey on parallel ant colony optimization. *Applied Soft Computing*, 11(8):5181–5197, 2011.

[Ree93] Colin R Reeves. *Modern heuristic techniques for combinatorial problems*. John Wiley & Sons, Inc., 1993.

[Rei91] Gerhard Reinelt. Tsplib — a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[RL02] Marcus Randall and Andrew Lewis. A parallel implementation of ant colony optimization. *Journal of Parallel and Distributed Computing*, 62(9):1421–1432, 2002.

[RM03] Xiaofeng Ren and Jitendra Malik. Learning a classification model for segmentation. In *Computer Vision, 2003 Ninth IEEE International Conference on*, pages 10–17. IEEE, 2003.

[RR11] C Yuheng Ren and Ian Reid. gslic: a real-time implementation of slic superpixel segmentation. *University of Oxford, Department of Engineering, Technical Report*, 2011.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[SOSR+15] Jesús Sánchez-Oro, Marc Sevaux, André Rossi, Rafel Martí, and Abraham Duarte. Solving dynamic memory allocation problems in embedded systems with parallel variable neighborhood search strategies. *Electronic Notes in Discrete Mathematics*, 47:85–92, 2015.

[SS03] Daniel Scharstein and Richard Szeliski. High-accuracy stereo depth maps using structured light. In *Computer Vision and Pattern Recognition, 2003 IEEE Conference on*, volume 1, pages I–195. IEEE, 2003.

[ST85] Chien-Chung Shen and Wen-Hsiang Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *Computers, IEEE Transactions on*, 100(3):197–203, 1985.

[Sto97] Ivan Stojmenovic. Honeycomb networks: Topological properties and communication algorithms. *Parallel and Distributed Systems, IEEE Transactions on*, 8(10):1036–1042, 1997.

[Stü98] Thomas Stützle. Parallelization strategies for ant colony optimization. In *Parallel Problem Solving from Nature PPSN V*, pages 722–731. Springer, 1998.

[SZS+08] Richard Szeliski, Ramin Zabih, Daniel Scharstein, Olga Veksler, Vladimir Kolmogorov, Aseem Agarwala, Marshall Tappen, and Carsten Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(6):1068–1080, 2008.

[Tal09] El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

[TF03] Marshall F Tappen and William T Freeman. Comparison of graph cuts with belief propagation for stereo, using identical mrf parameters. In *Computer Vision, 2003 Ninth IEEE International Conference on*, pages 900–906. IEEE, 2003.

[TKP01] Anastasios Tefas, Constantine Kotropoulos, and Ioannis Pitas. Using support vector machines to enhance the performance of elastic graph

matching for frontal face authentication. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(7):735–746, 2001.

[Tom99] Marco Tomassini. Parallel and distributed evolutionary algorithms: A review. 1999.

[TWZ08] Yuichi Taguchi, Bennett Wilburn, and C Lawrence Zitnick. Stereo reconstruction with mixed pixels using adaptive over-segmentation. In *Computer Vision and Pattern Recognition, 2008 IEEE Conference on*, pages 1–8. IEEE, 2008.

[Vek99] Olga Veksler. *Efficient graph-based energy minimization methods in computer vision*. PhD thesis, Cornell University, 1999.

[VLMT13] The Van Luong, Nouredine Melab, and E-G Talbi. Gpu computing for parallel local search metaheuristic algorithms. *Computers, IEEE Transactions on*, 62(1):173–185, 2013.

[WJW05] Martin J Wainwright, Tommi S Jaakkola, and Alan S Willsky. Map estimation via agreement on trees: message-passing and linear programming. *Information Theory, IEEE Transactions on*, 51(11):3697–3717, 2005.

[WZC13] Hongjian Wang, Naiyu Zhang, and Jean-Charles Créput. A massive parallel cellular gpu implementation of neural network to large scale euclidean tsp. In *Advances in Soft Computing and Its Applications*, pages 118–129. Springer, 2013.

[WZC+15] Hongjian Wang, Naiyu Zhang, Jean-Charles Créput, Julien Moreau, and Yassine Ruichek. Parallel structured mesh generation with disparity maps by gpu implementation. *Visualization and Computer Graphics, IEEE Transactions on*, 21(9):1045–1057, 2015.

[XJM12] Li Xu, Jiaya Jia, and Yasuyuki Matsushita. Motion detail preserving optical flow estimation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(9):1744–1757, 2012.

[YKN+12] Masato Yoshimi, Takuya Kuhara, Kaname Nishimoto, Mitsunori Miki, and Tomoyuki Hiroyasu. Visualization of pareto solutions by spherical self-organizing map and it's acceleration on a gpu. *Journal of Software Engineering and Applications*, 5(3), 2012.

[Zha13] Naiyu Zhang. *Cellular GPU Models to Euclidean Optimization Problems: Applications from Stereo Matching to Structured Adaptive Meshing and*

Traveling Salesman Problem. PhD thesis, Université de Technologie de Belfort-Montbeliard, 2013.

[ZK07] C Lawrence Zitnick and Sing Bing Kang. Stereo for image-based rendering using image over-segmentation. *International Journal of Computer Vision*, 75(1):49–65, 2007.

[ZWC+13] Naiyu Zhang, Hongjian Wang, Jean-Charles Créput, Julien Moreau, and Yassine Ruichek. Cellular gpu model for structured mesh generation and its application to the stereo-matching disparity map. In *Multimedia, 2013 IEEE International Symposium on*, pages 53–60. IEEE, 2013.

Abstract:

In this thesis, we propose a parallel computing model, called *cellular matrix*, to provide answers to problematic issues of parallel computation when applied to Euclidean graph matching problems. These NP-hard optimization problems involve data distributed in the plane and elastic structures represented by graphs that must match the data. They include problems known under various names, such as geometric *k*-means, elastic net, topographic mapping, and elastic image matching. The Euclidean traveling salesman problem (TSP), the median cycle problem, and the image matching problem are also examples that can be modeled by graph matching.
The contribution presented is divided into three parts. In the first part, we present the cellular matrix model that partitions data and defines the level of granularity of parallel computation. We present a generic loop for parallel computations, and this loop models the projection between graphs and their matching. In the second part, we apply the parallel computing model to *k*-means algorithms in the plane extended with topology. The proposed algorithms are applied to the TSP, structured mesh generation, and image segmentation following the concept of superpixel. The approach is called *superpixel adaptive segmentation map* (SPASM). In the third part, we propose a parallel local search algorithm, called *distributed local search* (DLS). The solution results from the many local operations, including local evaluation, neighborhood search, and structured move, performed on the distributed data in the plane. The algorithm is applied to Euclidean graph matching problems including stereo matching and optical flow.

Keywords: cellular matrix, graph matching, *k*-means, local search, parallel algorithms, graphics processing unit (GPU)

Résumé :

Dans cette thèse, nous proposons un modèle de calcul parallèle, appelé *matrice cellulaire*, pour apporter des réponses aux problématiques de calcul parallèle appliqué à la résolution de problèmes d'appariement de graphes euclidiens. Ces problèmes d'optimisation NP-difficiles font intervenir des données réparties dans le plan et des structures élastiques représentées par des graphes qui doivent s'apparier aux données. Ils recouvrent des problèmes connus sous des appellations diverses telles que geometric *k*-means, elastic net, topographic mapping, elastic image matching. Ils permettent de modéliser par exemple le problème du voyageur de commerce euclidien, le problème du cycle médian, ainsi que des problèmes de mise en correspondance d'images.
La contribution présentée est divisée en trois parties. Dans la première partie, nous présentons le modèle de matrice cellulaire qui partitionne les données et définit le niveau de granularité du calcul parallèle. Nous présentons une boucle générique de calcul parallèle qui modélise le principe des projections de graphes et de leur appariement. Dans la deuxième partie, nous appliquons le modèle de calcul parallèle aux algorithmes de *k*-means avec topologie dans le plan. Les algorithmes proposés sont appliqués au voyageur de commerce, à la génération de maillage structuré et à la segmentation d'image suivant le concept de superpixel. L'approche est nommée *superpixel adaptive segmentation map* (SPASM). Dans la troisième partie, nous proposons un algorithme de recherche locale parallèle, appelé *distributed local search* (DLS). La solution du problème résulte des opérations locales sur les structures et les données réparties dans le plan, incluant des évaluations, des recherches de voisinage, et des mouvements structurés. L'algorithme est appliqué à des problèmes d'appariement de graphe tels que le stéréo-matching et le problème de flot optique.

Mots-clés : matrice cellulaire, l'appariement de graphes, *k*-means, recherche locale, algorithmiques parallèles, graphics processing unit (GPU)

SPIM

■ École doctorale SPIM - Université de Technologie Belfort-Montbéliard
F - 90010 Belfort Cedex ■ tél. +33 (0)3 84 58 31 39
■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

utbm
université de technologie Belfort-Montbéliard