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**

GPU Parallel Euclidean Minimum Spanning Tree and Massive 2-/3-opt Moves for Graph Minimization

---

Official Documentation

par
Wenbao QIAO

Le2i FRE2005, CNRS, Arts et Métiers,
Univ. Bourgogne Franche-Comté, France

Soutenue le xxxx:

Abstract

In this thesis, we propose two new branches of research interests to graph minimization problems, first one, parallel Euclidean minimum spanning tree/forest (EMST/EMSF) algorithms for input graph that only possesses vertexes as input; second one, multiple 2-opt / 3-opt moves that are found on the same global tour or in the same integral Euclidean space for traveling salesman problems (TSP). To respond these two research interests, firstly, we propose a newly proposed hierarchical EMST/EMSF algorithm that combines the cellular partition based nearest neighbor search with Borůvka's algorithm, which provides an alternative hierarchical data clustering solution to various optimization problems whose input graph only contains vertexes with uniform or bounded data distribution. Secondly, a judicious decision making methodology of offloading which part of the $k-$opt heuristic works in parallel on Graphics Processing Unit (GPU) while which part remains sequential, called "parallel local search but sequential selection", in order to simultaneously execute, without interference, massive 2-/3-opt moves that are globally found on the same TSP tour or the same Euclidean space for many edges as well as keeping high performance on GPU side. This methodology for multiple 2-/3-opt moves is judicious since intervention of a sequential $O(N)$ time complexity tour reversal operation is unavoidable when using ordered arrays as TSP tour data structure for high performance GPU parallel $k-$opt implementation that considers coalesced memory access and usage of limited on-chip shared memory; it is valuable since our newly proposed non-interacted 2-/3-exchanges set partition algorithm that takes $O(N)$ sequential time complexity, and a new TSP tour data structure, array of ordered coordinates-index, for GPU high performance local search implementation. All these proposed parallel solutions work on GPU with parallel technique of "decentralized control, data/task decomposition, GPU on-chip shared or off-chip global memory".

Résumé

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Driven by the insatiable demand for real-time and high performance solutions for Euclidean optimization problems or pattern recognition applications, designing parallel or distributed algorithms is an practical choice. Various parallel/distributed implementations of the same classical algorithm have been studied. These algorithms differ from various parallel technique which they follow with, while different parallel techniques evolve with the development of computer hardwares. For example, concurrent computing, parallel computing and distributed computing are three large branches to fit for different computer hardwares.

Recent years, Graphics Processing Units (GPU) has developed into a highly parallel, multithreaded, manycore processor with tremendous computation horsepower and very high memory bandwidth, which provides a useful hardware for implementing various parallel algorithms.

To explore this tremendous computation horsepower provided by GPU for parallelizing the same target algorithm or problem, traditional parallel methods should be modified, or even totally new parallel algorithms should be proposed, since different hardwares have their own restraints and merits for designing parallel algorithms.

Further more, various parallel algorithms working on same GPU also differ in concrete "parallelism" with which they follow. For example,

- Parallel algorithms working based on data decomposition differ with those working based on task decomposition;

- Parallel algorithms that assign different roles to different process differ with those assign each process same and equal roles. For example, the *master-slave* model where the master process plays a central role to manage the slave processes, but each parallel process in our proposed algorithms works independently and equally.
- Parallel algorithms differ in quantity of global memory occupied by all processes, or in quantity of independent local memory occupied for each process. For example, parallel algorithms working based on data duplication differ from these working on data decomposition.
- Parallel algorithms also differ in the basic data structure on which they works, like the topological grid data structure (2D array) possesses an implicit index relationship between every two grid nodes, like node v_1 with index (0,1) can be accessed directly by node v_2 with index (2, 1), while adjacency list data structure does not possesses that attribute.

Apart from the hardware restraints, it is attribute of the target classical sequential algorithm and concrete applications that decide which kind of parallelism is better. For example, some image processing tasks working on regular 2D array (grid) can directly profit from the implicit index relationship between every two grid nodes for accessing neighbor pixels from any one pixel independently, while it is hard to use 2D grid to represent irregular graphs; applications that require self-training procedures on an irregular graph can hardly be implemented on a self-organizing map constructed by regular topological 2D grids; classical Kruskal's algorithm for building minimum spanning tree works based on sorting all edges of a complete graph, which means preparation of these edges and parallel sorting operation are necessary for parallelism of Kruskal's algorithm; implementation of classical 2-opt and 3-opt local search heuristics involves sub-tour reversal operation, which requires communication between parallel k-opt moves and restrains most existing parallel 2-/3-opt implementations.

In this thesis, we aim to design parallel algorithms working on GPU CUDA programming [Nvi10] for Euclidean graph minimization problems, such as parallel Euclidean minimum spanning tree (EMST) for input graph that only has vertexes, and multiple 2-opt / 3-opt moves of parallel 2-/3-opt local search on global TSP tour or integral Euclidean space for large scales traveling salesman problems (TSP). These parallel algorithms follow decentralized control with which each GPU processor works independently, in parallel, and without depending on data duplication.

In the literature, parallel minimum spanning tree (MST) usually work on general graph $G = (V, E)$ with explicit edge list E. So far, we do not find an parallel algorithm directly building EMST that only possesses vertexes as input. In this thesis, we propose a GPU

implementation of parallel Euclidean MST (EMST), which is the first attempt in this domain.

Current researches on multiple 2-opt / 3-opt moves of parallel 2-/3-opt local search implementation commonly work based on disjoint partitions of the original TSP tour or partitions of the Euclidean space. In this thesis, we are interested in efficiently massive 2-/3-opt moves found along the same global tour or in the same integral Euclidean space with high performance GPU computing.

1.2 Objectives and Contribution

Main objectives of this thesis are to design massively parallel algorithms based on GPU CUDA programming to solve two classes of parallel graph minimization problems, namely:

- parallel hierarchical Euclidean minimum spanning tree;
- multiple 2-opt / 3-opt moves along the global TSP tour or in the integral Euclidean space.

Different versions of these two classes of parallel algorithms follow the parallel characteristics of *"decentralized control, data/task decomposition, GPU on-chip shared or off-chip global memory"*.

So far, we do not find much similar work on solving these two classes of parallel graph minimization problems. We are the pioneers to find and enter these two domains. Besides, the experimental results show that our proposed methodologies to solve these two problems are judicious and valuable. Main contributions are listed below:

- A judicious decision making methodology of offloading which part of the $k-$opt heuristic works in parallel on Graphics Processing Unit (GPU) while which part remains sequential, called "parallel local search but sequential selection", in order to simultaneously execute, without interference, massive 2-/3-opt moves that are globally found on the same TSP tour or the same Euclidean space for many edges as well as keeping high performance on GPU side.

  This methodology is judicious since intervention of a sequential $O(N)$ time complexity tour reversal operation is unavoidable for each 2-/3-opt move when using arrays as TSP tour data structure for high performance GPU parallel $k-$opt implementation that considers coalesced memory access and usage of limited on-chip shared memory.

The methodology is valuable because of our originally proposed sequential non-interacted 2-/3-exchange set partition algorithm taking linear time complexity, and a new TSP data structure that unveils how to use GPU on-chip shared memory to achieve the same goal as using doubly linked list and array of ordered coordinates for parallel $k-$opt implementation.

Just as the name implies, the methodology of "parallel local search but sequential selection" divides parallel k-opt into two parts: GPU implementation of parallel local search to find valid k-exchanges for many edges (or all edges) along the same global tour or in the same Euclidean space, while it is the sequential selection algorithm to filter multiple non-interacted k-opt moves produced by the preceding GPU local search. Highlight is that, data structure for simultaneously implementing multi 2-/3-opt moves can hardly bring high performance on GPU side with consideration of coalesced GPU memory access and utilization of GPU on-chip shared memory.

We originally propose a new intermediate data structure to work on GPU side, and unveil the methods to combine high performance GPU local search algorithms with our proposed efficient non-interacted 2-/3-opt set partition algorithm for implementing massive k-opt moves simultaneously. We implement three GPU parallel local search algorithms following this methodology.

Besides, we proposed a GPU distributed local search algorithm with disjoint tour divisions that dynamically moves along the tour. This parallel local search enables fixed quantity of 2-opt moves to be executed simultaneously on the same tour, and entirely works on GPU side without CPU intervention.

- An originally proposed hierarchical minimum spanning forest/tree (MSF/MST) data clustering algorithm that combines Elias' cellular partition nearest neighbor search (or Bentley's spiral search) with Borůvka's algorithm, to build MSF/MST in its Euclidean version where the input data is only a set of Euclidean points with uniform or bounded data distribution. To achieve that, it is unavoidable to solve following sub-problems working on irregular graphs. These sub-problems include
  - Basic data structure for distributed / parallel computing;
  - Parallel closest outgoing edge findings to a whole irregular graph;

  Based on contribution of this work, application to other Euclidean optimization problems, for example the well known traveling salesman problem or the hierarchical image clusterings could be envisaged with high efficiency and within divide and conquer scheme.

Parallel solutions to these two graph minimization problems further provide basic methods for constructings more complex combinatorial optimization algorithms, like divide-conquer, TABU search [Glo89] and variable neighborhood search (VNS) [MH97].

1.3 Plan of the thesis

In chapter 2, we briefly introduce background on these two classes of graph minimization algorithms, including both sequential and parallel related work to minimum spanning tree and 2-opt / 3-opt local search algorithms. As our proposed parallel EMST algorithm takes advantage of nearest neighbor search algorithm, classical algorithms for nearest neighbor search will be an independent section in this background chapter.

In chapter 3, we introduce the common data structure used in this thesis, which is for better understanding of our proposed algorithms.

In Chapter 4, we present our proposed Euclidean hierarchical minimum spanning forest (tree) that combines Elias' nearest neighbor search with Borůvka's algorithm to build hierarchical data clusterings from bottom-to-up and in a divide-conquer mode.

In Chapter 5, we present our judicious decision making methodology of offloading which part of the $k-$opt heuristic works in parallel on Graphics Processing Unit (GPU) while which part remains sequential, called "parallel local search but sequential selection". We also present our GPU distributed 2-opt local search with disjoint tour partitions to execute fixed quantity of 2-exchanges in parallel, which follows the basic ideas of parallel local search proposed by Verhoven et al. [VAS95].

In Chapter 6, we conclude this thesis and discuss some future works.

Chapter 2

Background

2.1 Introduction

Before presenting the principal work of this thesis, we want to introduce some basic concepts and related algorithms in this chapter, which are related to our main work and necessary for readers to better understand our contributions in the latter chapters.

Main contribution of this thesis lies in two classes of originally proposed parallel graph minimization algorithms working on GPU CUDA platform with parallel characteristics of "parallel computing, decentralized control and GPU shared/global memory", in order to deal with various Euclidean optimization problems. Therefore, in this background chapter, we first introduce classification of parallel algorithms according to their parallel characteristics.

Following the selected parallel characteristic, the first class of graph minimization algorithm is GPU implementation of Euclidean minimum spanning tree/forest (ESMT/EMSF) based on combining Borůvka's algorithm with Elias nearest neighbor search or Bentley's spiral search approaches. It provides solutions to applications like parallel closest outgoing edge finding to a graph, or parallel/distributed hierarchical EMST data clustering problems, whose input only contains vertexes with uniformly or bounded data distribution in the Euclidean space. So far, we do not find such combination for sequential or parallel EMST methods in the literature. In the third and fourth sections of this chapter, we will introduce the nearest neighbor search algorithms and related work for building EMST.

The second class of graph minimization algorithms is on parallel 2-opt, 3-opt local search algorithms for large-scale traveling salesman problem (TSP). We provide a new methodology to select and execute, without interference, multiple 2-/3-opt moves that

are found globally on the original whole tour or the whole Euclidean space. So far, this methodology could reach the fastest convergence to 2-opt local optima in the literature. After the related work about EMST, we introduce 2-opt, 3-opt and $k-$opt local search and its related work in the fifth section.

2.2 Classification of parallel algorithms

Different parallel algorithms should be classified in order to distinguish the difference between various parallel implementations that solve the same problems. They mainly fall into categories like concurrent computing, parallel computing, and distributed computing, while the parallel computing still have variants like data and task parallelism. In this section, we will briefly clarify different categories of parallel implementations, introduce parallel computing with CUDA programming on GPU.

Concurrent Computing

At the view of one computing processor (core), if many different tasks can be executed concurrently by one same processor, while in an interleaved fashion and only one statement is executed at any point in time, it is called concurrent computing [ADM82].

Parallel Computing

At the view of multiple processors (cores), if each processor works independently on separate task or data in parallel, and many statements are executed at any point in time, it is called parallel computing [QQ94].

- *Data Parallelism* refers to a parallel procedure that multiple processors simultaneously execute the same function across the elements of a dataset.
- *Task Parallelism* refers to a parallel procedure that multiple processors execute many different functions across same or different datasets.

Various implementations of data parallelism also differ from which level of data element one processor deals with, for example, an input dataset containing various independent graphs and vertexes, parallel algorithms working on independent graphs differ from these working on independent vertexes of these graph.

Distributed Computing

Comparing with concurrent and parallel computing, distributed computing usually indicates that the algorithm firstly divides a big problems into sub-problems that can be

solved by using concurrent or parallel computing, then collect or combine these sub-solutions to get final results [TTL05].

Memory Access

From the view of memory access, parallel programs look like concurrent ones when memory accesses are serialized. During processes of parallel computing, one processor may works on the results of another processor, these dependencies and communication among the many processors leads to conflict of memory access. Serialized memory access is necessary in this case.

Parallel Computing with CUDA programming on GPU

The NVIDIA GPU provides a highly parallel, multithreaded and many-core environment for a special kind of parallel computing architecture called SIMT (Single-Instruction, Multiple-Thread), which is akin to SIMD (Single Instruction, Multiple Data). Here in this section, to get a quickly short impression of CUDA programming, two common well-known characteristic of CUDA programming is presented below according to the CUDA manual. For further information about CUDA programming, please turn to CUDA manual [Nvi10].

Firstly, multiple threads simultaneously execute the same program, which is achieved by the SIMT architecture. When a CUDA program on the host CPU invokes a kernel function with appropriate kernel configuration according to the GPU hardware, each multiprocessor on this GPU gets one or more thread blocks to execute. Then the multiprocessor creates, manages, schedules, and executes threads in group of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. A warp executes one common instruction at a time, so full efficiency can be realized when all 32 threads of a warp agree on their execution path. [Nvi10].

Secondly, considering memory access, GPU provides hierarchical device memory spaces composed by three types of read-write memory spaces as shown in Fig.2.1. Each thread has its own local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. Two additional read-only memory spaces, constant and texture memory, are also provided by GPU [Nvi10]. Because shared memory is on-chip, it has much higher bandwidth and much lower latency than local or global memory[Nvi10].

More efficiency can be achieved by coalesced device memory access performed by consecutive threads in a half-warp, though CUDA programming on GPU allows arbitrarily

coalesced and scattered memory access from multiple threads [Nvi10]. Besides, using the on-chip shred memory when it is possible can also improve the efficiency. However, size of this on-chip shared memory is generally limited in most current GPU cards and is far less than global memory [Nvi10].

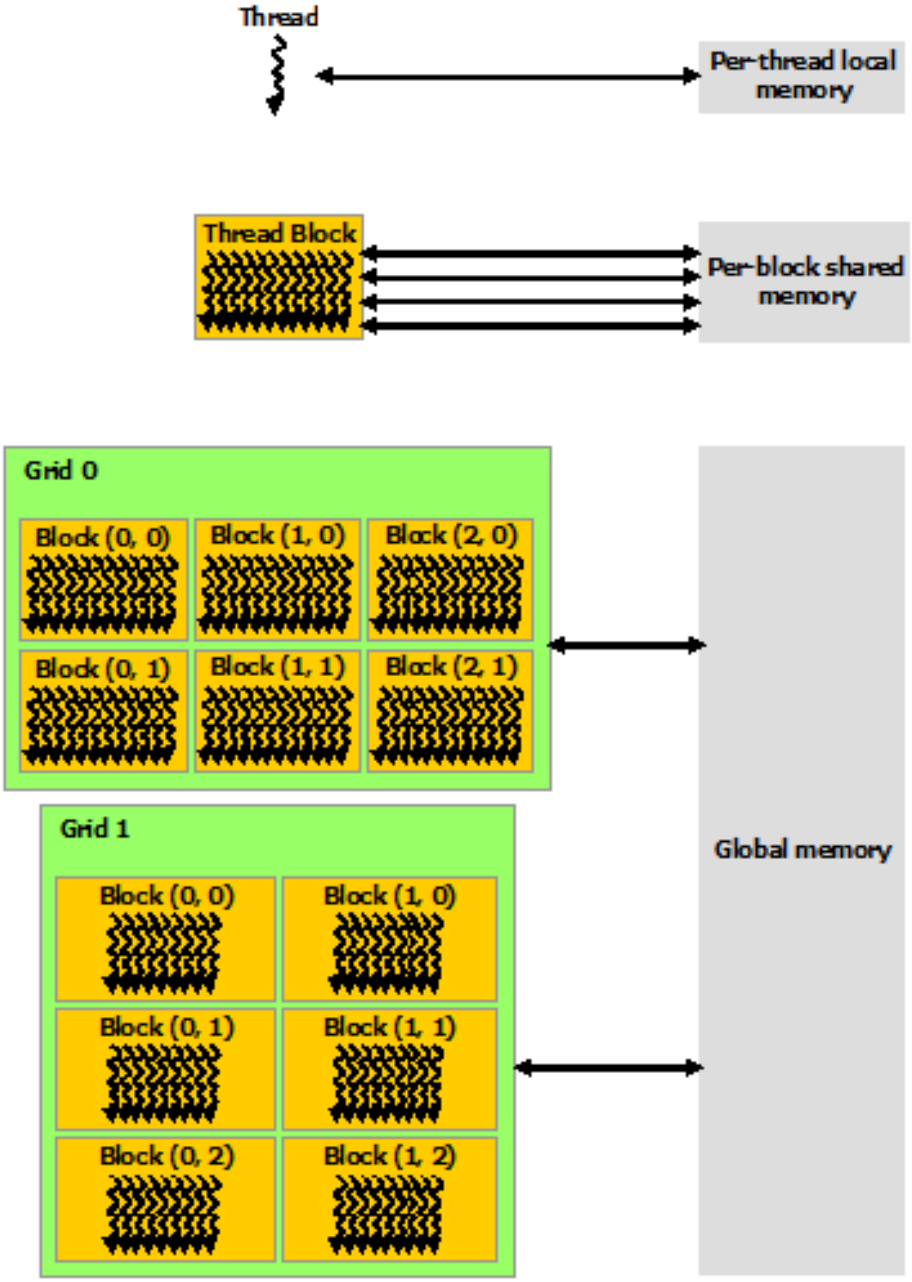


FIGURE 2.1: GPU device memory hierarchy [Nvi10].

2.3 Nearest neighbor search

Let $P = \{p_i\}_0^{N-1}$ be a set of N points in a k-dimensional space, and let q be a query point. A statement of the *Nearest Neighbour Problem* is [GGT00]:

Find the point p_c in P which is the minimum distance from q, i.e.

$$\|q - p_c\| \leq \|q - p_i\| \quad \forall p_i \in P \tag{2.1}$$

There are two assumptions in this standard definition of *Nearest Neighbour Problem* Eq.2.1:

- p_c is unique;
- distance metric is the Euclidean distance and denote it as $D_{ij} = \|p_i - \vec{p_j}\|$

Instead of sorting all possible edges connecting the query point q in order to find its nearest neighbor, two most common nearest neighbor methods have been widely applied under above two assumptions: $k-d$ *tree* [Ben75] and Elias method [Riv74, Cle79]. Bentley's local spiral search [BWY80] can be categorized into Elias' approaches. Though there are cases in which the first assumption does not hold, for example two or more points lie in the same Euclidean position or on a circle centering the query point, these cases only affect concrete applications and does not influence the nearest neighbors searching by using these well-known methods.

2.3.1 K-D Tree

In computer science, a $k-d$ tree, or k-dimensional tree [Ben75] [FBF77] is a special space-partitioning data structure for organizing some number of N points in a k-dimensional space. It is a special binary search tree with other constraints imposed on it. The root of the tree represents the entire space and the leaf nodes represent subspaces containing mutually exclusive small subsets. Every non-leaf node can be regarded as implicitly generating a splitting hyperplane that divides the space into two parts. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree [Ben75].

Sequentially building a $k-d$ tree takes $O(N\log N)$ time complexity and $O(K*N)$ space complexity. A single sequential nearest neighbor search takes close to $O(logN)$ time.

2.3.2 Elias' neighbor search on cellular partition

Elias' approaches [Riv74, Cle79] indicate a category of nearest neighbor searching algorithms that works on dividing the Euclidean space into a regular grid of congruent and non-overlapping sub-regions, cells, or bins, which is called *cellular partition* in this thesis. Each cell has its coordinates on the grid which can be accessed by its neighbor cells. Each cell contains a list of the points that fall within its boundaries. The partitioned area that relates data to cell is illustrated in Fig.2.2, where the data point and its cell has a mapping relationship. Given position of a point, the algorithm can access to the cell containing this point.

When a query point q comes in, the algorithm firstly searches the cell where the query point q is located, then passes to search these neighbor cells that are close to the starting cell [GGT00]. This process is repeated until the distance to any unexplored cell is greater than the distance to the candidate closest point found. In practice, the given spatial ordering of the cells is used to restrict the search: cells are accessed in a concentric order around the query point, and the search stops when the radius of search exceeds the distance to the nearest candidate point found [GGT00].

Time complexity of Elias method is driven mostly by the number of points in individual cells. Given an Euclidean space, actual cost of Elias method depends on the efficiency of the implementation of the bin data structure as well as its access function [GGT00].

Elias methods get their best time performance when dealing with uniform or bounded data distribution, while the worst case happens when all of the data points are located in a single bin. Given a uniformly or bounded input data distribution, a $\sqrt{n} \times \sqrt{n}$ size of bins on 2-D Euclidean space and $\sqrt{n} \times \sqrt{n} \times \sqrt{n}$ size of bins on 3-D Euclidean space will lead to the best performance of Elias method. This is due to the reason that number of points in each bin is bounded when dealing with the uniform or bounded data distribution. Given an arbitrary data distribution, adaptive division of the Euclidean space according to data distribution will also improve performance of Elias method. [GGT00]

Constructing Elias cells (cellular partition) takes $O(N)$ complexity and occupies $O(N)$ space complexity, while a single nearest neighbor search takes $O(1)$ complexity on uniformly or bounded data distribution [BWY80].

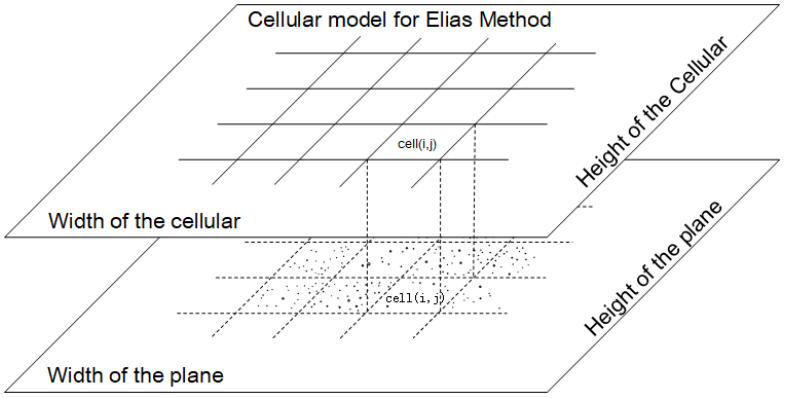


FIGURE 2.2: Cellular partition based on Elias' approaches. Lower layer: the 2D Euclidean area that contains all the input data; Upper layer: the cellular partition.

2.3.3 Bentley local spiral search

Bentley's local spiral search [BWY80] can be categorized into Elias' approaches explained in section 2.3.2, which works based on cellular partition of the euclidean space.

Considering the nearest neighbor searching in the plane, where both the original N points and the query point are chosen independently from a uniform distribution over the unit square, the idea of preprocessing step is to assign each point a small square (bin or cell) of area C/N, where C is constant, so that the expected number of points in each cell is C. This is easily done by creating an array of size $\sqrt{N/C} \times \sqrt{N/C}$ that holds pointers to the lists of points in each bin [BWY80].

When a query point q comes in, the spiral search algorithm searches the cell where point q is located. If that cell is empty, the algorithm starts searching neighbor cells surrounding that cell in a spiral-like pattern until a point is found. As shown in Fig.2.3, once one point is found, it is guaranteed that there is no need to search any cells that do not intersect the circle of radius equals to the distance to the first point found and centered at the query point [BWY80]. One stopping criterion for this spiral search is that " once the point in the nearest neighbor is found, only bins intersecting the circle must be searched" [BWY80]. This stopping criterion makes the number of cell accesses slightly larger than necessary but simplifies the specification of how the appropriate bins are to be found [BWY80].

With the premise that there are a constant number of points per cell on the average, the expected running time of a spiral search is bounded by a constant, independent of the value N [BWY80].

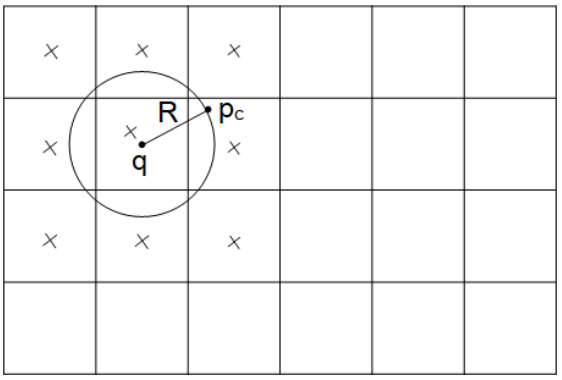


FIGURE 2.3: Bentley spiral search with cellular partition. R indicates the Euclidean radius centering current query point on the Plane. x indicates these cells being accessed to find the closest point p_c to the query point q.

Voronoi diagram The Voronoi Diagram of a given point set P captures many *closeness properties* for solving the nearest neighbor problems. For example, the *voronoi polygon* of the query point q is defined to be locus of all points that are closer to q than any other point in P; the dual of the Voronoi diagram, which is obtained by connecting all pairs of points that share an edge in their respective Voronoi polygones, is a supergraph of the minimum spanning tree of point set P. [BWY80]

The problem of constructing the Voronoi diagram of a planar point set is somewhat more delicate than nearest-neighbor searching. Bentley et al. [BWY80] have mentioned the fact that the nearest neighbor to a new point is that point whose Voronoi polygon contains the new point. This fact can be used to give a fast worst-case algorithm for nearest neighbor searching. For each point, Bentley et al. compute its Voronoi polygon by listing its edges together with the associated Voronoi neighbors in clockwise order.

With the premise of bivariate uniform data distribution, Bentley et al. proposed linear expected time to construct Voronoi diagram. The basic idea is to search all cells in a relatively small neighborhood of each point in a spiral-like fashion until at least one point is found in each of the eight octants shown in Fig.2.4, or give up having examined $O(logN)$ cells. [BWY80]

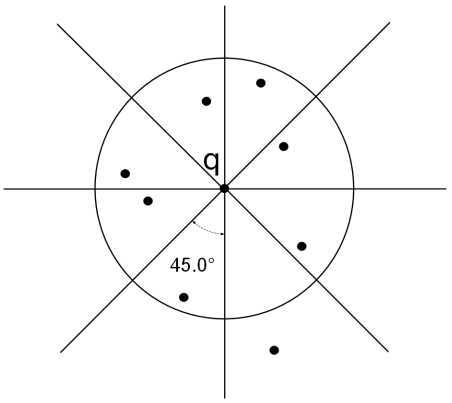
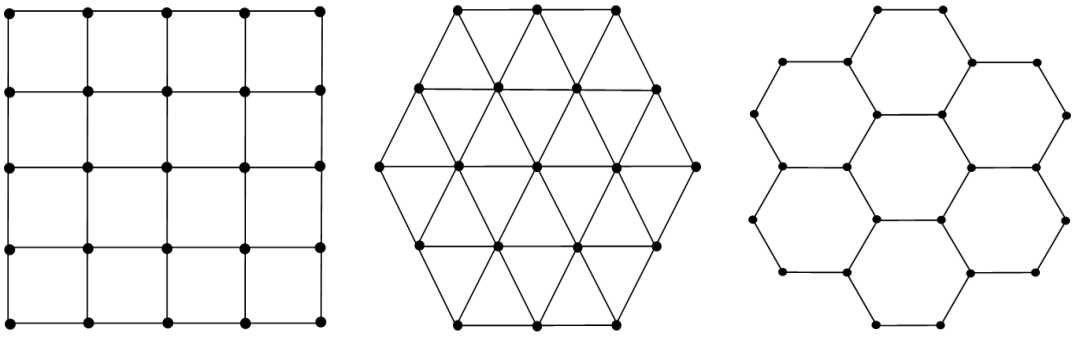


FIGURE 2.4: Bentley's method to construct Voronoi Diagram with linear expected time.

2.3.4 GPU cellular matrix model

Cellular matrix is defined as a set of functions to allow easy manipulation of gird coordinate systems, where the main tools are coordinate transfer functions provided in relation to some subdivision of the plane at different levels of recursive decomposition [Wan15]. According to this definition, the cellular matrix includes three components as it is claimed in Wang's thesis of [Wan15]. Firstly, basic hierarchical structure in the plane, which contains three levels working on the regular 2-Dimension grid. The first level indicates that starting points of the cellular decomposition of the plane are three possible tessellations [Sto97] of the plane with regular polygons of the same type as presented in Fig.2.5. Then the second level is a zoom-out regular grid obtained from the first level with cell radius r, same zoom-out makes the third level. Secondly, possibility for more recursive decomposition working based on these basic three levels by repeating that zoom-out function. Lastly, cellular matrix with different topologies.



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

FIGURE 2.5: Three possible tessellations of a plane acting as starting points of cellular matrix in Wang' works [Wan15].

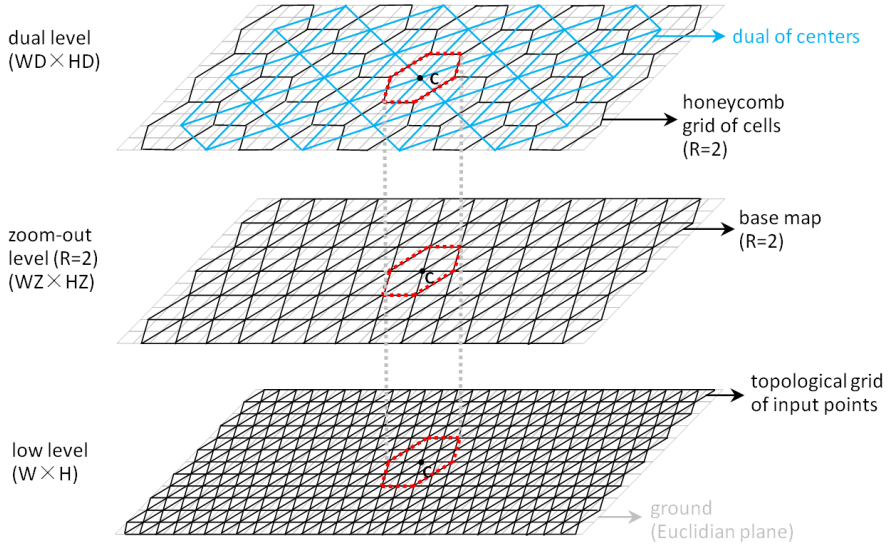


FIGURE 2.6: Cellular matrix model with hexagonal topology (3D view) [Wan15].

Every level of this cellular matrix possesses the same topological 2D grid extracted from the lower level. The cellular matrix includes three components as it is claimed in Wang's thesis of [Wan15]. First component, basic hierarchical structure in the plane, which contains three levels working on the regular 2-Dimension grid. The first level indicates that starting points of the cellular decomposition of the plane are three possible tessellations [Sto97] of the plane with regular polygons of the same type as presented in Fig.2.5. Then the second level is a zoom-out regular grid obtained from the first level with cell radius r, same zoom-out makes the third level. Second component, possibility for more recursive decomposition working based on these basic three levels by repeating that zoom-out function. Last component, cellular matrix with different topologies as shown in Fig.2.6 [Wan15].

In our work of this thesis, we extend this hierarchical topological 2D grids to topological 2D irregular graphs presented in chapter 3 and chapter 4, which is more flexible for preserving both regular and irregular hierarchical topologies.

2.4 Minimum spanning tree (MST)

Given a general complete weighted undirected graph $G = (V, E)$ with explicit edge list E, a minimum spanning tree (MST) means a subset of E that connects all vertices without any circles and with minimum total weight. A minimum spanning forest (MSF) consists of MSTs on each of the connected components of that graph G. In graph theory, a *connected component (or just component)* of an undirected graph G is a subgraph of G in which any two vertices are connected by a finite or infinite sequence of edges.

MST has long been researched as it can roughly estimate the intrinsic structure of a dataset [ZMMF15]. Due to this quality, MST has long been applied in image segmentation [AXC00, XU97], cluster analysis [XOX02, Zah71, ZMW10, ZMF11], classification [JTP+09], manifold learning [Yan04], density estimation [LKC+12], diversity estimation [LA11], TSP estimation [Chr76] and some applications of the variant problems of MST [CCGC07, Önc07, SS10].

Three classical MST algorithms like Borůvka's [Bor26], Kruskal's (1956) [Kru56] and Prim's (1957) [Pri57] algorithms establish the basis of MST implementations. However, it is difficult to apply these traditional MST algorithms to large dataset since their time complexity is quadratic. The best time complexity for a sequential general MST solution is proposed by Bernard Chazelle [Cha00], and it takes $O(E\alpha(E,V))$ where α is the functional inverse of Ackermann's function.

Borůvka's algorithm begins with each vertex of the graph G being a component, finds minimum weighted outgoing edge for each component and adds all such edges to the MSF in one iteration. Finding a minimum weighted outgoing edge for each subtree and merging these subtrees into a new larger component (subtree) of the new spanning forest constitute a Borůvka step. Original Borůvka's algorithm works sequentially and takes $O(n^2 \log n)$ complexity. This algorithm has nature attribute to perform in divide-conquer manner in massively parallel way.

Kruskal's algorithm works by firstly sorting all edges of the complete graph, and then inserting edges one by one until all points in P have been connected without circles. The time complexity of Kruskal's algorithm is $O(n^2 \log n)$.

Prim's algorithm is a greedy algorithm. It firstly selects a random vertex as a tree, and then repeatedly adds the shortest edge that connects a new vertex to the tree until all the vertices are included [ZMMF15]. The time complexity of Prim's algorithm is $O(n^2 \log n)$.

To accelerate MST implementation, various MST algorithms have been proposed based on those three classical algorithms. Two main branches exist in these newly proposed MST algorithms. The first one is parallelism of classical MST algorithms working on general graph $G = (V, E)$ with explicit edge list, which is explained in section 2.4.1. The second one is to take advantage of geometrical closest points information when dealing with MSTs in its Euclidean version (EMST), which is introduced in section 2.4.2.

2.4.1 Parallel MST

When working on general weighted undirected graph $G = (V, E)$ with explicitly predefined edge list E, many attempts exist to address the MST problem in parallel or distributed way based on different hardwares. They are all variations of the well-known algorithms of Prim's, Kruskal's or Borůvka's algorithms [CC96] [ADJ+98] [DG98] [HJ99] [BC04] [HVN09] [VHPN09] [WHG11] [RP15].

Chung et al.(1996)[CC96] implemented parallel MST algorithm based on the sequential Borůvka's algorithm focusing on reducing communication costs on asynchronous, distributed-memory machines. Given a connected, undirected graph G with n vertices and m weighted edges, their work pays more attention to the case that number of processors is much less than the size of the graph, and the graph is distributed among the processors.

Bader et al.(2004)[BC04] tried to marry the Prim and Borůvka's approaches base on symmetric multiprocessors, in which they synthesize previous parallel solutions for different graph operation, like point-jumping approaches proposed in [CC96], parallel sample sort operation in [HJ99], cache-friendly adjacency arrays in [PPP04], and provide their new flexible adjacency list representation to achieve acceleration for building MST. With the cache-friendly adjacency arrays, each enty of an index array of vertices points to a list of its incident edges. The compact-graph step first sorts the vertex array according to the supervertex label, then concurrently sorts each vertex's adjacency list using the supervertex of the other endpoint of the edge as the key. After sorting, the set of vertices with the same supervertex label are contiguous in the array, and can be merged efficiently [BC04] .

Vineet et al.(2009) [VHPN09] formulated parallel Borůvka's algorithm in a recursive framework exploiting parallel data mapping primitives base on GPU. In each iteration, they reconstruct the supervertex graph which is given as an input to the next level of recursion. They use the segmented scan [SHZO07] to find the minimum weighted outgoing edge from each vertex. They then merge vertices into supervertices using split and scan primitives. Another split/scan pair is used to remove duplicate edges and to assign supervertex numbers. The main contribution of their work is the recursive formulation of Borůvka's approach for undirected graphs as a series of general primitive operations [VHPN09].

Harish et al.(2009) [HVN09] proposed a parallel implementation of Borůvka's algorithm on GPU with well-designed adjacency list and various well-designed GPU kernel launches [VHPN09].

Wang et al.(2011) [WHG11] proposed parallel Prim's algorithm on GPU by using the developed GPU-based min-reduction data parallel primitive in the key step of Prim's algorithm.s

However, when using these parallel MST algorithms, despite the need to prepare the $N \times N$ edges of a complete graph, it also needs necessary sorting jobs when using various adjacency list and it is also a challenging problem to create an efficient data structure for parallel computation [HVN09].

2.4.2 Euclidean MST (EMST)

Given a set S of N points in Euclidean $d-$dimensional space, a Euclidean minimum spanning tree (EMST) is a spanning tree of S whose edges have a minimum total length among all spanning tree of S, where the length of an edge is the Euclidean distance between its vertices [AESW91]. EMST is special since geometrically closest points intrinsically correspond to requirement of MST. Researchers have provided efficient sequential EMST algorithms taking advantage of that geometrical information by mainly improving efficiency of accessing correct edges for MST.

Since MST is a subset of edges in the dual of the Voronoi diagram, Shamos et al.(1975) [SH75] described the Voronoi diagram as a general structure for searching the plane and gives a worst case $O(n\log n)$ algorithm to find EMST of 2D plane with a Euclidean distance measure.

Lingas(1994) [Lin94] provided a linear-time algorithm for constructing a relative neighborhood sparse graph from the Delaunay triangulation. This sparse graph can be used to generate EMST.

Bentley et al.(1975)[BF75] proposed a variant of Prim's algorithm, which employs the fast nearest neighbor algorithm of $k-d$ tree [FBF77]. The algorithm's performance takes an estimated $O(n\log n)$ time depending on configuration of the points in coordinate space [BF75].

Rajasekaran (2004)[Raj04] presented a variant of Prim's algorithm with expected linear time algorithm, the key point of his algorithm is to use partition of Euclidean space to find neighborhood closest points, for example, the algorithm partitions 2D Euclidean area into a $\sqrt{n} \times \sqrt{n}$ square grid with n cells possessing uniformly distributed data.

March et al.(2010) [MRG10] used a dual-tree method on a $k-d$ tree to find the nearest neighbor pair for each component under Borůvka's framework. They get a significant acceleration for building EMST in three or four dimension space [MRG10].

Delaunay Triangulation and EMST. Given a discrete point set P in a plane, a Delaunay triangulation is a triangulation such that no point in P is inside the circumcircle of any triangle. Two points p_i and p_j are connected by an edge in the Delaunay triangulation if and only if there is an empty circle passing through p_i and p_j. Delaunay triangulation defines a sparse graph from the complete graph, where the closest pair of points in P are neighbors in the Delaunay triangulation.

It has been proved that the minimum spanning tree of a point set P is a subgraph of the Delaunay triangulation. To build EMST from Delaunay triangulation, the algorithm should firstly build Delaunay triangulation of the point set, which takes $O(n \log n)$ time; and then compute the MST by Kruskal's algorithm. This leads to a total time complexity of $O(n \log n)$ for building EMST. However, there is no Delaunay triangulation for a set of points on the same line.

So far, we do not find an efficient parallel/distributed algorithm that directly addresses the EMST without predefining the $N \times N$ edge list that takes $O(N^2)$ time complexity before starting the algorithms.

2.5 Local Search for Euclidean traveling salesman problem (TSP)

The traveling salesman problem (TSP) [Lap92, JM97, ABCC11] is one of the well-known combinatorial optimization problems that have been classified as NP-hard [GJ79]. One solution to a TSP instance is one permutation of input N cities, while different possible permutations form various solutions. According to the permutation, the salesman travels each city once and returns to the starting city of the permutation. Symmetric TSP forms undirected graph where distance between every two nodes is the same. Asymmetric TSP forms directed graph where paths may not exist or the distances in opposite direction might be different between two nodes.

Due to the quantity of possible solutions augments significantly along with the size of input cities, finding an optimal solution with brute sequential algorithms requiring factorial execution time for large scale instances. A number of TSP heuristics have been developed to approximate the optimal tour, various parallelism of these heuristics [VAS95, Kar77, RS12, RS13, Luo11, QC17b, QC17a] have also been studied.

In this section, we firstly present the common TSP formulation used in this thesis to better explain our proposed algorithms; then, we introduce the well-known 2-opt 3-opt local search algorithms, discuss their principles and variants; their parallel implementation will be introduced at last.

2.5.1 TSP formulation

Given a symmetric Euclidean TSP instance, let V be a set of N cities and $d(u, v) \in \Re$ a distance for each pair of cities $u, v \in V$. Let S be a set of all possible permutation solutions in complete undirected graph $(V, V \times V)$ and $s \in S$ a solution. Let $\pi : \{0, ..., k, k+1, ..., N-1\} \rightarrow V$ be a bijection. A TSP tour solution $s = (\pi_0, \pi_1, \pi_2, ..., \pi_k, \pi_{(k+1)}, ..., \pi_{(N-1)})$ indicates that the input N cities have been permuted with an increasing permutation order from 0 to $(N-1)$ along current tour direction. Edge e_k indicates the salesman travels from city π_k to π_{k+1} in order to avoid traveling one city twice, $(k+1) mod N = 0$ in case of $k = (N-1)$. The goal is to find a permutation solution $s \in S$ that minimize the cost function in Equation.2.1.

$$f(s) = \sum_{k=0}^{N-1} d(e_k) \tag{2.1}$$

2.5.2 Iterative 2-/3-opt

One of the most commonly used heuristic methods to approach TSP is repeating a series of local search algorithm called 2-opt [Cro58], 3-opt and $k-$opt ($k >= 2$), which will often result in a tour with length less than 5% (2-opt) or 3% (3-opt) above the Held-Karp bound [JM97, LMS03]. This repetitive procedure is called Iterative Local Search (ILS). Generic framework of ILS is shown in Algorithm.2.5.1. However, implementing a $k-$opt algorithm involves reversing permutation order of cities in some segments of the original tour, which should be carefully treated in order to keep the tour valid.

Algorithm 2.5.1 Generic Iterative Local Search framework.

```
1: procedure ITERATIVE LOCAL SEARCH
2:     s_0 ← InitialSolution
3:     while termination condition not met do
4:         s'_i ← Refresh Permutation Order (s_i) when it is necessary;
5:         s_(i+1)' ← LocalSearch (s_i);
6:         s*_(i+1) ← Acceptance Criterion (s'_i, s_(i+1)');
7:         s_(i+1) ← Execute the selected optimization (s*_(i+1)).
8:     end while
9: end procedure
```

2-opt local search:

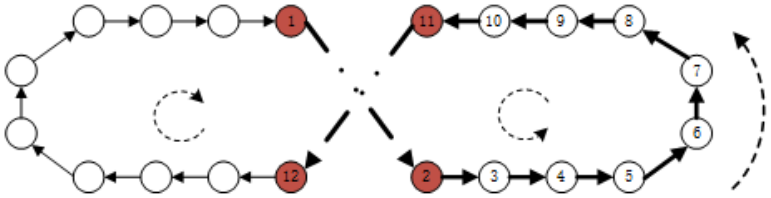
Consider two neighbor TSP tour solutions $s_i, s_{i+1} \in S$, if $s_{(i+1)}$ would be shorter than s_i by removing two edges and inserting two edges, this transition is defined as **a 2-opt exchange (or one move)** $\mathcal{N}_2$ [Lin65].

Searching one 2-opt move should consider related cities' tour order, since there is only one way to reconnect the two divided sub-tours in order to keep the tour valid after 2-exchange. Given two oriented edges (e_k, e_p) and four related cities $i = \pi_k, i+1 = \pi_{k+1}, j = \pi_p, j+1 = \pi_{p+1}$ along the same tour direction, in order to find a correct 2-opt exchange, the algorithm should check the following condition in Equation.2.2 to remain the tour valid, which is called one **2-opt check (or evaluation)**.

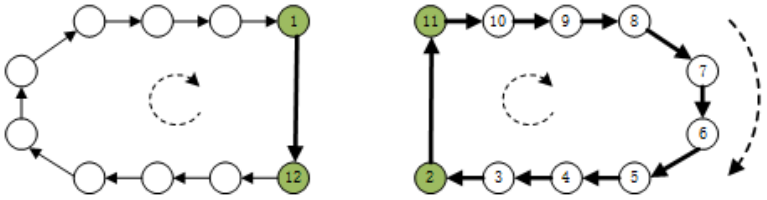
$$d(i, i+1) + d(j, j+1) > d(i, j) + d(i+1, j+1) \tag{2.2}$$

Once one 2-opt exchange $\mathscr{N}_2^{kp} \rightarrow \{e_k, e_p\}$ is found, removing two edges from the original tour produces two **sub-tours**, inserting two new edges will reverse original permutation order of cities in one of the two sub-tours. For example, after executing a correct 2-exchange $\mathscr{N}_2^{kp} \rightarrow \{e_k, e_p\}$, permutation order of cities in the original sub-tour $\pi_{(k+1)} \rightarrow \pi_p$ of s_i is totally inversed.

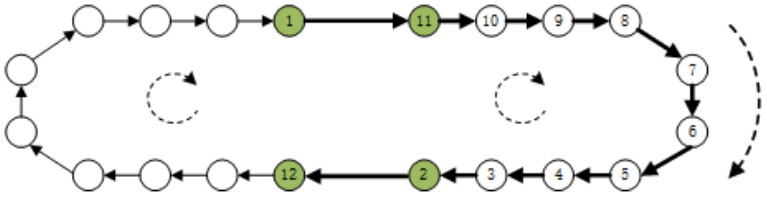
One example to present above detailed processes is shown in Fig.2.7. Consider current tour orientation that the salesman travels from city π_k to π_{k+1}, after removing two edges $e_1 = (\pi_1, \pi_2)$ and $e_{11} = (\pi_{11}, \pi_{12})$ in Fig.2.7(a), only inserting two new edges (π_1, π_{11}) (π_2, π_{12}) in Fig.2.7(c) that will remain the tour valid, while the previous sub-tour's permutation order from city $\pi_2 \rightarrow \pi_{11}$ in Fig.2.7(a) is totally inversed.



(a) Current tour solution s_i.



(b) Wrong 2-exchange would cut tour solution s_i.



(c) One correct 2-opt move leads to valid tour $s_{(i+1)}$ but reverses sub-tour of s_i.

FIGURE 2.7: Tour order plays an important role in iterative 2-opt implementations. ① represents one city's tour order.

3-opt local search:

The 3-opt works in a similar fashion with 2-opt. Consider two neighbor TSP tour solutions $s_i, s_{i+1} \in S$, if $s_{(i+1)}$ would be shorter than s_i by removing three edges and inserting three edges, this transition is defined as **a 3-opt exchange** $\mathcal{N}_3^{kpq}$.

Removing three edges from the original tour produces three sub-tours, namely sub-tours from city $\pi_{k+1} \rightarrow \pi_p$, $\pi_{p+1} \rightarrow \pi_q$ and $\pi_{q+1} \rightarrow \pi_k$. After executing a correct 3-exchange, one or two sub-tours' original permutation order have been inversed. If a tour is 3-optimal it is also 2-optimal [Hel00].

Sequential Acceleration of Iterative k-opt

According to the above analysis, the k-opt algorithm actually includes two sub-steps: search and execute k-opt. The **classical sequential 2-opt search** that aims to find the best 2-opt move for the integral tour often indicates that every pair of edges have been checked once, which leads to complete $\frac{N*(N-1)}{2}$ 2-opt checks needed to be done in one iteration (run) of Alg.2.5.1. Similarly, a complete 3-opt search leads to totally $\frac{N*(N-1)*(N-2)}{6}$ checks in one 3-opt run in order to find the best 3-opt move for the integral

tour in one iteration of Alg.2.5.1. This classical k-opt implementation is obvious time-consuming when treating an initial brute large-scale TSP solution that needs to iterate millions of times 2-opt or 3-opt moves before reaching convergence to local optima. Besides, implementing a $k-$opt algorithm involves reversing permutation order of cities in some segments of the original tour, which should be carefully treated in order to keep the tour valid.

Various sequential speeding up strategies exist to improve efficiency of the classical sequential 2-/3-opt implementation. For example, instead of executing complete $\frac{N*(N-1)}{2}$ 2-opt checks to find the best 2-opt move for the integral tour, the first strategy executes $(N-1)$ 2-opt checks along the global tour to find the best 2-opt move for one edge, which is the well-known "best improvement for one edge". The second strategy further decreases local searching ranges for one edge's optimization, for example, manually keep a fix list of each edge's m nearest neighbor, or the well-known "first improvement for one edge" strategy that prunes the $k-$opt search if the first optimization for one edge is found. After one edge is optimized, these algorithms begin next edge's optimization repeating the same procedure until termination condition is met.

The third acceleration strategy can be achieved by changing TSP data structures. TSP data structure also influences performance of iterative $k-$opt implementation, since $k-$opt involves reversing segments of current tour where the k-exchange happens. For example, in order to re-make an integral TSP tour, it is necessary to take $O(N)$ time complexity after each k-opt move when using an array of ordered coordinates TSP tour data structure and is bad for large instances with more than 10^3 cities [Nil03]. When working on doubly linked list data structure where each node has two links pointing to its two neighbor cities, each 2-/3-opt exchange only involves changing links of related cities and does not take $O(N)$ time to re-make an integral tour. It also exists two-level tree [FJMO95] that will provide advantage of $\sqrt{n}$ complexity per reversal operation, and splay trees takes $O(log_2(n))$ time complexity in worst cases both for moves and lookups. It is suggested to use two-level tree for problem size bigger than 10^3 but up to 10^6 cities, and use splay trees for problems with more than 10^6 cities [Nil03].

In general, though it exists these sequentially speeding up strategies, the $k-$opt local search algorithm still takes polynomial time complexity for large scale TSP instances. Another option is parallel $k-$opt implementation.

2.5.3 Parallel 2-/3-opt local search

Parallelism of 2-opt implementations have been studied since a long time ago [Kar77, VAS95, Luo11, RS12, RS13, QC17b]. Related works either execute multiple 2-opt moves

in separate disjoint partitions of the original tour [VAS95] or partitions of the Euclidean space [Kar77], or execute one 2-opt move after a whole procedure of parallel evaluation of classical 2-opt local search [RS12][RS13], or take quadratic time complexity to select multiple non-interacted 2-opt moves [ROB+18].

Johnson [JM97] [JM07] discussed parallel schemes like "geometric partitioning and tour-based partitioning". One geometric partitioning scheme proposed by Karp [Kar77] is based on a recursive subdivision of the overall input space containing the cities into rectangles [JM97].

Verhoeven et all [VAS95] distinguished parallel 2-opt algorithms between data and function parallelism. In an algorithm with data parallelism, data is distributed over a number of processes, and each process executes a local search algorithm; in an algorithm with function parallelism one step of the sequential local search algorithm is executed in parallel [VAS95]. In the paper [VAS95], they concentrate on tailored data parallelism to partition a solution into a number of disjoint partial solutions, subsequently, the neighborhoods of all partial solutions are searched simultaneously.

Parallel 2-opt implementations based on task parallelism like Rocki et al.[RS12, RS13] distribute totally $\frac{N*(N-1)}{2}$ 2-opt checks to $O(N)$ GPU threads to select the best 2-opt move in current tour, while they need CPU-aided tour reversal operation after each 2-exchange to prepare the TSP array of ordered coordinates for next iteration of high performance parallel $\frac{N*(N-1)}{2}$ 2-opt checks. They achieve a high performance GPU local search implementation because each thread responds to fixed quantity of 2-opt checks on GPU on-chip shared memory with coalesced memory access.

Our previous work in [QC17a] had mentioned a "parallel local search but sequential selection" methodology and the $O(N)$ time complexity non-interacted 2-opt set partition algorithm, in order to execute multiple 2-opt moves found in the same Euclidean space, while much less details were explained on how to keep high performance GPU local search. Rios et al.[ROB+18] propose a distributed variable neighborhood descent metaheuristic on multi-GPU system, while their method verifies conflict between a candidate move and the remaining moves [ROB+18], each verification goes over the remaining moves and takes quadratic time complexity to judge conflict between multiple moves.

In general, two main researching branches on multiple 2-/3-opt moves can be categorized: one is parallel local search on disjoint partitions of the same tour [VAS95] or Euclidean space [Kar77], the other one is our proposed massive 2-/3-opt moves that are globally found on the same tour or the same Euclidean space [QC17a].

2.6 Conclusion

We have reviewed related work on parallel implementations of two classes of graph minimization algorithms, namely Euclidean minimum spanning tree (ESMT) and iterative 2-opt, 3-opt local search algorithms.

Related researches on parallel MST usually work on general graph $G = (V, E)$ with explicit edge list E. When dealing with EMST with these general parallel MST algorithms, the implementations need to prepare the edge list, necessary sorting jobs when using various adjacency list and it is a challenging problem to create an efficient data structure for parallel computation. So far, we do not find an parallel algorithm directly approaching EMST in literature. In this thesis, we propose a parallel implementation of EMST by combining Borůvka's algorithm and Elias' nearest neighbor search algorithms in chapter 4, which is the first attempt in this domain.

Parallel implementation of iterative 2-opt, 3-opt local search have been researched from a long time ago, while related parallel implementations on multiple 2-/3-opt moves commonly work based on disjoint partitions of the original tour or Euclidean space. In this thesis, we are interested in efficient multiple 2-/3-opt moves found along the same global tour or in the same integral Euclidean space with high performance GPU computing, explained in chapter 5.

Chapter 3

Common data structure

3.1 Introduction

Objectives of this thesis are to design massively parallel algorithms based on GPU CUDA programming. These parallel algorithms follow parallel techniques of *"decentralized control, data/task decomposition, GPU on-chip shared or off-chip global memory"*. To achieve this, besides the delicate design of parallel algorithms, the basic data structure should also be clarified to provide basic support for implementing that parallel characteristics.

In this chapter, we present two basic data structures used in later chapters for parallel/distributed computing.

3.2 Doubly linked vertex list (DLVL)

Considering massively parallelism with decentralized control and data decomposition, which require separate graph or vertex works independently and in parallel, graph representation plays an important role.

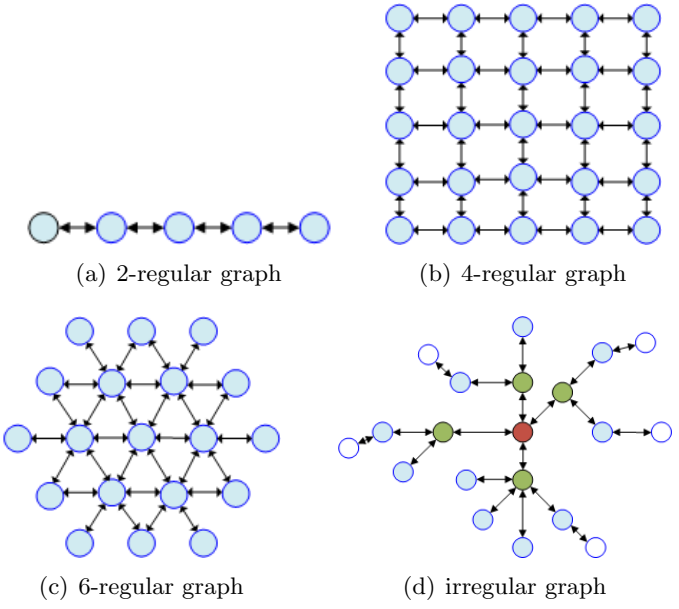
Graph representation should satisfy two basic requirements for these parallel/distributed algorithms proposed in later chapters: firstly, allowing the algorithms to access the whole current graph from any one of its vertex for broadcasting when necessary; secondly, the whole graph occupies linear memory on GPU global memory.

Various graph representation can satisfy those two requirements. For example, give N nodes on 2D hexagonal or rectangular grid, each grid node can work independently and in parallel with the predefined topologies to access neighbor grid vertices, and the

grid also takes $O(N)$ space. An *adjacency list* requiring $O(V + E)$ space [HVN09] for representation of a finite graph also satisfy those requirement due to the reason that it associates each vertex in the graph with the collection of its neighboring vertices or edges. Many variations of adjacency list exist in literature [HVN09, VHPN09, BC04], differing in the details of how they associate vertices and collections, in what kinds of objects are used to represent these vertices and edges.

We choose to apply a standard distributed graph representation with independent adjacency list assigned to each vertex of the graph, namely an adjacency list where each vertex only possesses a collection of its adjacency neighboring vertices.

It naturally follows that the graph is doubly linked since each node of a given edge has a link in the node's adjacency list towards to the connected node. We call it as doubly linked vertex list (DLVL) in order to distinguish DLVL from doubly linked list (DLL) or doubly connected edge list (DCEL). In DLL, every node has only two links that are references to the previous and to the next node in the sequence of nodes. A DCEL contains a record for each edge, vertex and face of a subdivision of the input space, while we mainly emphasize vertexes in a DLVL.



(a) 2-regular graph (b) 4-regular graph

(c) 6-regular graph (d) irregular graph

FIGURE 3.1: Graphs represented by doubly linked vertex list. (a) doubly linked list. (b, c, d) Two dimensional doubly linked vertex list to present regular and irregular graph.

In graph theory, a regular graph indicates that each vertex has the same number of neighbors (degrees), otherwise the graph is called an irregular graph. Comparing with 2D grid that can only represent regular graphs, like the hexagonal grid represents 6-regular graph, the rectangular grid represents 4-regular graph in Fig.2.5, DLVL is much more flexible to represent both 2D or 3D, regular or irregular graphs as shown in Fig.3.1. For

example, Fig.3.1 (a) is doubly linked list that can be used to represent TSP tour, Fig.3.1 (b, c) shows two dimensional regular graphs, while Fig.3.1 (d) shows irregular graph represented by DLVL. This is due to the reason that total number of links possessed by one node can be changed easily, this total number also indicates the node's degree in the graph. We list the advantages of using DLVL below:

Advantage of Using DLVL. Though these graphs represented by regular 2D grid (2D array) also allow the parallel/distributed algorithms to access the whole current graph from any one of its vertex for broadcasting when necessary, for example these work in [WZC13, WMC+15, WMC17], DLVL (doubly linked vertex list) has its outstanding advantages over regular grid for parallel/distributed algorithms working based on data decomposition or distributed divide-conquer framework, for example:

- Multiple independent small DLVLs can be merged easily into large DLVLs in a divide-conquer mode;
- Graph presented by DLVL can be dynamically grown or decreased according to concrete applications;
- Graph presented by DLVL can be predefined as topological map that maintains regular or irregular topologies between neighboring nodes;
- Topological relationship is presented indirectly by links of nodes, not directly by nodes;

Considering software implementation of DLVL, it is implemented by using arrays with same indices in this thesis. One example of using arrays to represent doubly linked list (2-regular graph in Fig.3.1) is shown in Fig.3.2.

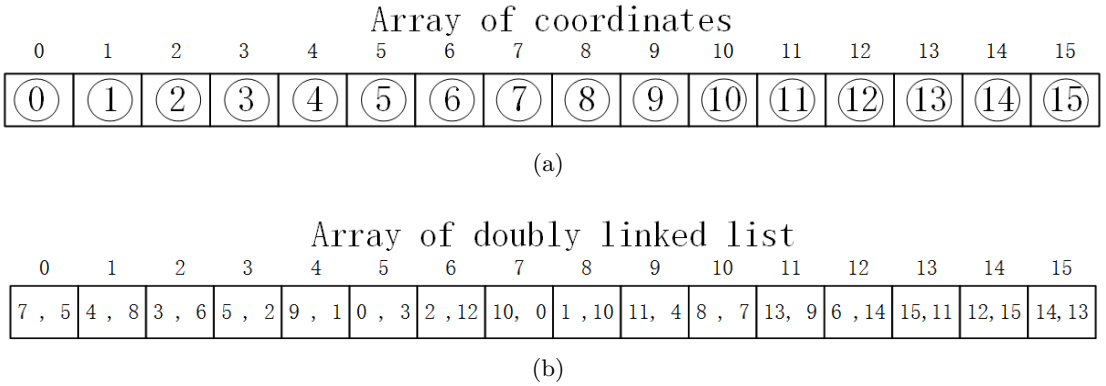


FIGURE 3.2: Doubly linked list implemented by arrays.

3.3 Bounded Cellular Partition

One central operation in this thesis is local spiral search [BWY80] based on cellular partition. Many local spiral searches are executed in parallel by the many nodes. Following the definition of Elias' approaches explained in section 2.3.2, the Euclidean space containing the input N points is partitioned into $\sqrt{n} \times \sqrt{n}$ congruent and non-overlapping cells. One example of 2-D partition is shown in Fig.3.3. Each cell has a vector storing the identifiers of input Euclidean points it covers. Each point has a projection to the cell where this point locates.

With this kind of cellular partition, each cell only contains average one point in case of uniform input data distribution, each cell contains constant quantity of points in case of bounded data distribution. Considering GPU implementation, each cell has a bounded buffer listing the identifiers of Euclidean points locating in that cell.

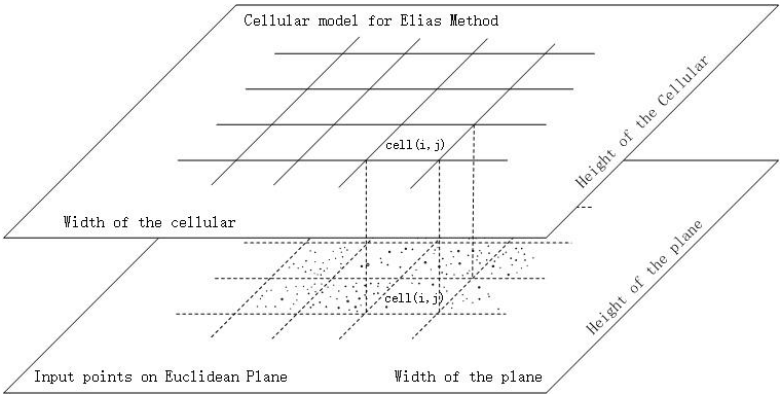


FIGURE 3.3: Cellular partition based on Elias' method. Lower layer: the 2D Euclidean area that contains all the input data; Upper layer: the cellular partition.

3.4 Conclusion

In this chapter, we introduce two common data structures used in this thesis, namely doubly linked vertex list (DLVL) for distributed graph representation, and the cellular partition for GPU parallel Elias' nearest neighbor search algorithms.

Chapter 4

EMST-Elias methodology

4.1 Introduction

In this chapter, we will introduce the first original contribution of this thesis, the ***EMST-Elias*** methodology to build hierarchical minimum spanning tree or forest of large N points set in Euclidean space (EMST/EMSF) for applications whose input data is uniformly or boundedly distributed in the Euclidean space. The methodology follows principles of the divide-and-conquer Borůvka algorithm working from bottom to up, while adapting Elias' nearest neighbor approaches to compute the minimum outgoing edges of each independent sub-tree, and will be a candidate choice for parallel applications with data decomposition and decentralized control. Though the parallel MSF/MST has been studied from a long time ago, we do not find similar parallel EMST methods in literature, and name this kind of combination using Borůvka's framework and Elias' nearest neighbor search as ***EMST-Elias*** method.

A minimum spanning tree (MST) roughly estimates the intrinsic structure of a dataset, and a minimum spanning forest (MSF) consisting of various small independent MSTs naturally divides the input N data into different categories or clusterings. A hierarchical MSF containing hierarchical data clustering can be widely used for optimization algorithms within divide-conquer framework. Solutions to build hierarchical minimum spanning tree or forest (MST/MSF) provide basic component to many other algorithms, like bichromatic closest pair problems [AESW91], nearest insertion [Nil03] or Chirstofid's heuristics [Chr76]. MST/MSF has long been applied in image segmentation, cluster analysis, classification, manifold learning, density estimation, diversity estimation, TSP estimation and some applications of the variant problems of MST [ZMMF15].

Two kinds of such clustering strategies exist in literature. One is obtained by deleting all edges longer than a specified cutoff in the final MST, which follows a top-to-down

strategy by varying the scale of the cutoff, like these work in [WWW09]; the other is to build MST by following Borůvka's algorithm that maintains a minimum spanning forest from bottom-to-up, like the work done by March et al. [MRG10] that combines a modified sequential kd-tree-based [FBF77] nearest neighbor search algorithm with Borůvka's algorithm.

Given a complete general graph $G = (V, E)$ with N vertices and predefined $N \times N$ edges, the traditional sequential MST algorithms like Prim's (1957) [Pri57], Kruskal's (1956) [Kru56] or Borůvka's (1926) algorithms [Bor26] (see [NMN01] for translation) takes $O(N^2)$ time complexity. The classical *Borůvka's Algorithm* [Bor26] has a natural attribute to maintain hierarchical minimum spanning forest (MSF) from bottom-to-up in divide-conquer mode. It works by iteratively finding a minimum outgoing edge for each subtree (component) and merging these subtrees into a new larger one.

Many attempts exist to address the MST problem in parallel or distributed way when the input is a general graph with explicit edge list [CC96] [ADJ+98] [DG98] [HJ99] [BC04] [HVN09] [VHPN09] [WHG11] [RP15]. They are all variations of the well-known algorithms of Prim's, Kruskal's or Borůvka's algorithms.

However, considering MST in its Euclidean version (EMST) when the input data is a set of points in the Euclidean space $\Re^d$, with no edge list as input, and where weights between any two nodes is the Euclidean distance between them, the graph is implicitly complete and very few parallel approaches directly address EMST except for some efficient sequential algorithms like these works using Delaunay trangulation [Lin94], using Voronoi diagram [SH75], using k-d tree and dual tree [MRG10].

4.2 Problems of parallel EMST

So far, we do not find an efficient parallel/distributed algorithm that directly addresses the EMST without predefining the $N \times N$ edge list that takes $O(N^2)$ time complexity before starting the algorithms.

Since no edge list is given as an input, and the beginning graph $G = (V, \emptyset)$, $(v_0, v_1, ...v_n \in V)$ is implicitly complete, instead of preparing the complete $N \times N$ edge list, we choose to directly find the closest neighbor point to each input point since geometrically closest neighbors naturally satisfy the requirement of MST. And it is a natural choice to follow the classical *Borůvka's Algorithm* [Bor26]. *Borůvka's Algorithm* works by iteratively finding a minimum outgoing edge for each subtree (component) and merging these subtrees into a new larger one.

Without predefining edge list leads to one key step in building EMST, namely finding the closest **outgoing edge** to one entire sub-tree component in order to follow Borůvka's framework. This key point can be turned into following **two sub-steps**.

- Find the graph's closest **outgoing point** to each vertex;
- Find the minimum **outgoing edge** within one component by using graph search.

The first sub-step can be solved by using classical **nearest neighbor search** algorithms like k-d tree [Ben75] or Elias method [Riv74, Cle79] with searching filters, which has been explained in section 2.3. These algorithms are original used to find the closest point to one vertex. K-d tree is often preferred because its $O(logN)$ time complexity for searching closest point, and the recent state-of-art sequential EMST algorithm proposed by March et al. [MRG10] uses a dual-tree on a kd-tree and a dual-tree on a cover-tree for searching closest point and claims time complexity as $O(NlogN\alpha(N))$, where α is the inverse of the Ackermann function. But during parallelism of the hierarchical k-d tree data structure, it is complex to both consider the requirement of memory occupation and decentralized control with which each point's nearest neighbor search works independently.

Rather than dealing with parallelism of the hierarchical k-d tree data structure, we turn to **Elias'** closest point finding approaches [Riv74, Cle79], like **Bentley's** spiral search [BWY80] and its variants, that we think to be natural candidate approaches to implement massively parallel closest point findings that follows decentralized control, data decomposition and is independent to input size. Balanced k-d tree has a sequential construction cost in O(Nlog$N$) operations, Elias' approaches have sequential O(N) construction time/space cost and $O(1)$ time complexity for one closest point finding on uniformly or bounded data distribution [BWY80] in the Euclidean space. Here, we expect to exploit such geometric properties into the GPU parallel platform and their improvement adapted to EMST.

The second sub-step can be achieved by graph search algorithms like breadth-first search (BFS). Classical Sequential BFS operates on a $frontier$ data structure (usually queues or vectors) that contains a set of vertices to be visited next, and keeps this $frontier$ data as local variables. Since CUDA programming on GPU allows arbitrarily coalesced and scattered memory access from multiple threads, each kernel launch can run one independently sequential BFS to collect the minimum outgoing edge within one component under the assumption that GPU can provide sufficient local memory for each independent BFS implementation. This BFS implementation can be found in the

state-of-art work done by Zhou et al.[ZHWG08] for constructing parallel k-d tree, while there are other specific researches on parallel BFS .

In the following sections, we will firstly present outline of the proposed parallel EMST-Elias methodology. Secondly, we present the data structure used to implement the proposed parallel EMSF/EMST algorithm. Thirdly, we introduce this originally proposed approaches on building EMSF/EMST both serially and in parallel. We discuss both theoretical and practical time complexity in the fourth section and present our experimental results in the fifth section. In the last section, we present its application to compose optimization algorithms, like bichromatic closest pair problem, nearest insertion, Christofid's heuristics.

4.3 Parallel EMST-Elias methodology

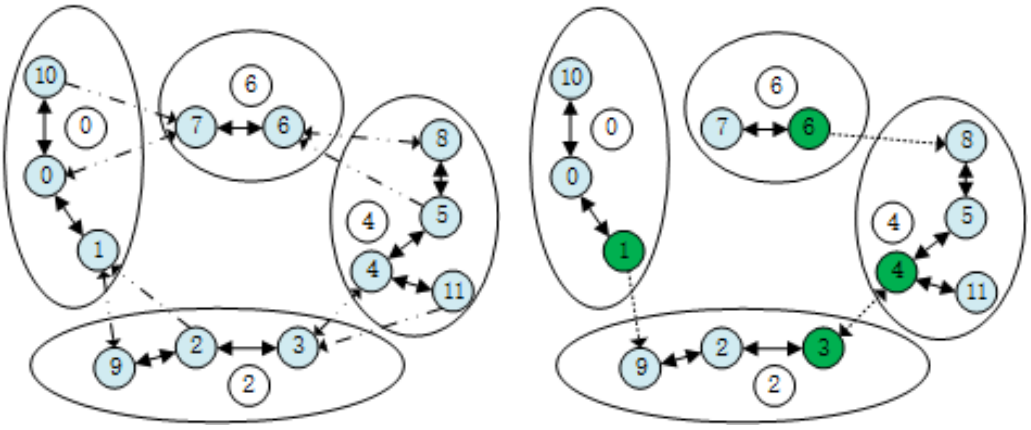
4.3.1 Outline

One iteration of the proposed EMST-Elias algorithm consists in the execution of four steps illustrated in Fig.4.1 by following Borůvka's framework. In its parallel version, each one step can be executed as Kernels in GPU platform with CUDA programming [Nvi10] following data decomposition and decentralized control. These four steps consist in augmenting a spanning forest by finding minimum outgoing edges until the overall spanning tree is reached. Besides the proposed methodology, the distributed graph representation, DLVL (doubly linked vertex list explained in section 3.2), provides essential data structure for these four steps.

4.3.2 Algorithm initialization

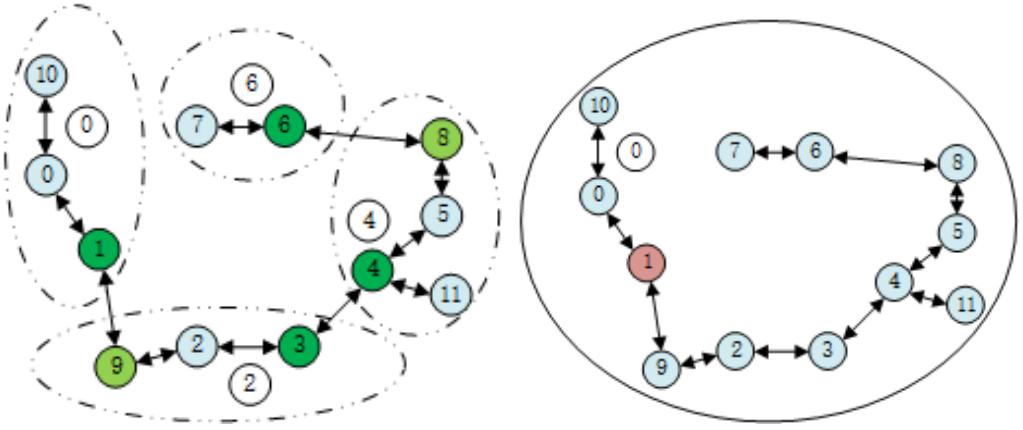
The initialization step mainly includes the initialization of the DLVL explained in section 3.2 as the initial spanning forest and the construction of bounded cellular partition illustrated in Fig.3.3 for implementing Elias' nearest neighbor search algorithms.

Distributed graph representation In order to satisfy the proposed parallel procedures outlined in section 4.3.1, we use a simple variation of adjacent list data structure to represent distributed EMST graphs (or sub-trees) in the spanning forest, namely the doubly linked vertex list (DLVL) explained in section 3.2. It follows that the graph is doubly linked, as shown in Fig.4.1 since each node of a given edge has a link to its connected node. This doubly linked property provides key support to the proposed Euclidean Minimum Spanning Forest (Tree) algorithm, since the algorithm needs to access



(a) *Find Minimum 1*: Find component's closest outgoing point to each vertex. Number in white circles represent each component' root identifier.

(b) *Find Minimum 2*: Find minimum outgoing edge (dark blue nodes) for each component.



(c) *Connect Graph*: Add link to each winner node's (dark blue nodes) adjacency list and to its correspondent node (green nodes).

(d) *Compact Graph*: Build new component structure by updating each component's root identifier.

FIGURE 4.1: The four main steps of the proposed parallel EMST implementation by following Borůvka's framework.

one whole independent sub-tree from any one of its vertex, and for easy connection of any two adjacent sub-trees with the natural advantage of using DLVL explained in section 3.2.

At the initialization step of EMST-Elias method, each node of the DLVL contains a bounded size buffer (explained in section 3.2) for memorizing its adjacent neighbor nodes because EMST has bounded degree with the assumption that every point's position is unique in the Euclidean space. Maximum degree of a EMST is 8 in 2D space or 14 for 3D space according to the uniqueness property [RS94]. Each vertex is treated as one component of the initial spanning forest and has its empty bounded buffer for memorizing its MST neighbors.

Bounded Cellular Partition for the Closest Point Finding. Following the definition of Elias' nearest neighbor search approaches, the Euclidean space containing the input N points is partitioned into $\sqrt{n} \times \sqrt{n}$ congruent and non-overlapping cells illustrated in Fig.3.3 and explained in section 3.3.

Besides, each vertex contains a special link to itself as its root in order to identify the component to which this vertex belongs. It is managed according to disjoint set data structure.

4.3.3 Find component's closest outgoing point to each vertex

With the analysis of parallel EMST problematic explained in section 4.2, the first step of the proposed EMST-Elias' method shown in Fig.4.1 needs to find the graph's closest outgoing point to each vertex, namely *Find Minimum 1*. At each execution of this step, each sub-tree of current minimum spanning forest is treated as an independent graph. Given a graph $G = (V, E)$ (component) together with other vertexes that belong to other different graphs locating in the Euclidean spaces, finding the graph's outgoing point to each vertex $v_i \in V$ can be turned into nearest neighbor problems, which is done by searching each vertex' closest outgoing point with Bentley's spiral search by adding a filter.

One classical Bentley's spiral search is shown in Fig.4.2. Starting from the cell containing the query point q and then accesses to neighbor cells in a spiral manner centering the starting cell, until all neighbor cells intersecting a circle have been searched, the circle is defined as "a circle of radius equals to the distance to the first point found and centered at the query point" [BWY80].

However, as shown in Fig.4.3, small EMSTs in the hierarchical EMSF grow with different iterations, while the cellular partition of the input Euclidean space does not change. This leads to a phenomenon that searching one point's closest outgoing point using pure Bentley's spiral search will take $O(N/2)$ complexity in the worst case, even though this worst case rarely appears. But this case could be improved by canceling unnecessary cell's access in later iterations, for example adding uniqueness property for each query point during Benley's spiral search processes.

4.3.4 Find minimum outgoing edge within each component

After the work of finding the graph $G = (V, E)$'s closest outgoing point to each vertex $v_i \in V$, it is the graph search algorithm to traverse the graph in order to collect the minimum outgoing edge for this graph, namely *Find Minimum 2* in Fig.4.1.

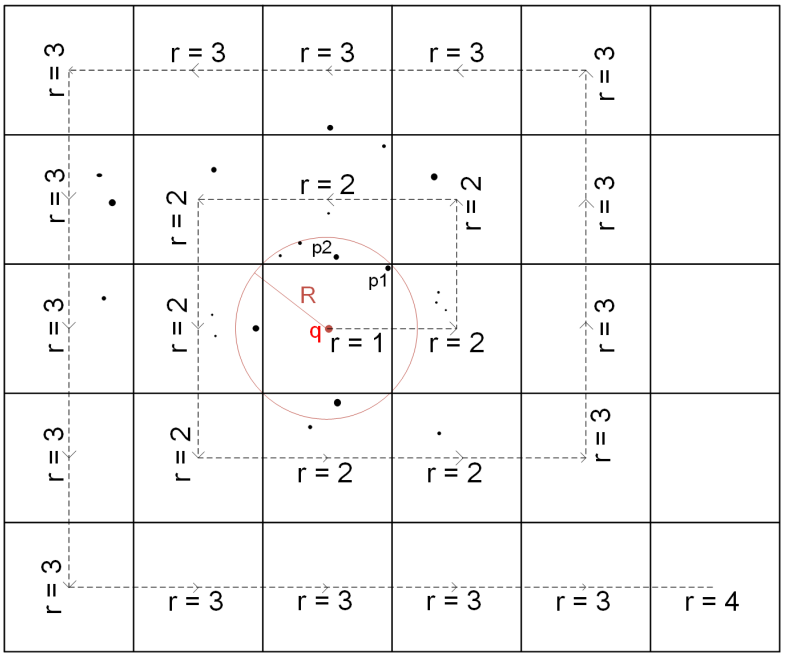


FIGURE 4.2: The way to access neighbor cells in Elias' approaches. Black round points are Euclidean points. r indicates the searching radius on cellular partition centering the starting cell containing the query point q. R indicates the Euclidean radius centering current q on the Plane.

With the assumption that GPU can provide sufficient device memory for each kernel launch, the classical sequential BFS algorithm can be implemented on GPU, since CUDA programming allows arbitrary coalesced and scattered memory access to both the GPU off-chip global and on-chip shared memory. While it does not exclude other parallel BFS implementations to collect the minimum outgoing edge within an independent component.

4.3.5 Avoid circles with parallel choosing rules

During the two *Find Minimum* steps for finding the minimum outgoing edges to different components, in order to avoid circles in cases where equal minimum outgoing edges exist, each independent process should select the same connections between components. Considering input data arbitrarily distributed on 2D Euclidean space, for example, two or more points locate in the same Euclidean position or on the same circle centering the query point, graph circles may appear at following two cases.

- First case, a component (or vertex) has more than one outgoing edges with equal minimum weight towards different components, as shown in Fig.4.4 (a);
- Second case, a component (or vertex) has more than one outgoing edges with equal minimum weight towards one same component, as shown in Fig.4.4 (b, c).

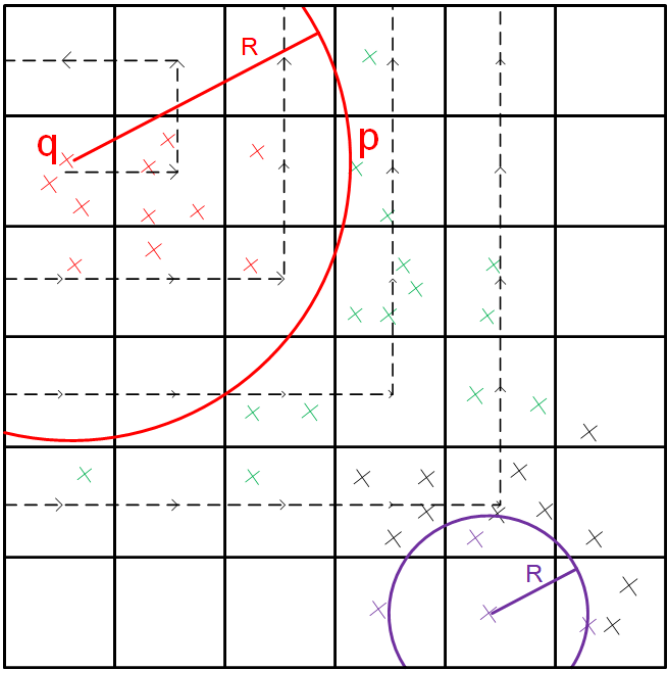
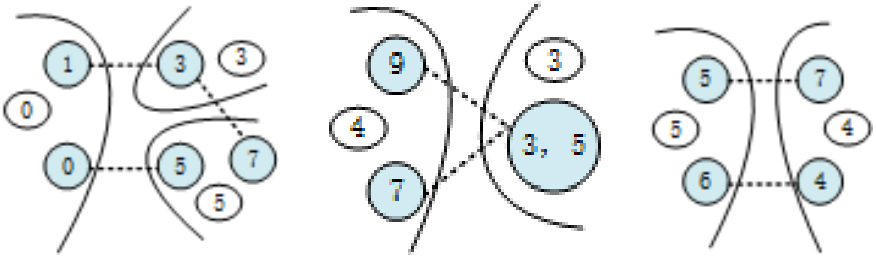


FIGURE 4.3: Find the closest outgoing point p for a query point q with pure Bentley's Spiral Search.



(a) (b) (c)

FIGURE 4.4: Cases where circles may appear, which should be avoided by strict parallel choosing rules. According to the rules, multiple components find the same outgoing edge in parallel during the two *Find Minimum* steps. For example, (a) only connection (v_1, v_3) and (v_0, v_5) are selected; (b) only (v_3, v_7) is selected when vertex v_3, v_5 lie on the same Euclidean position; (c) only (v_4, v_6) is selected. Round blue circles mark the input vertexes v_i, their component roots are indicated in the ellipse white circles, dash lines indicate outgoing edges that have equal weight.

To avoid circles during parallel computation, separate choosing rules should be respected at the two independent [*Find Minimum*] steps to select edges in the same minimum or maximum direction. Taking minimum as an example, the former [*Find minimum 1 for each vertex*] should respect that outgoing vertex v_j possessing the least identifier j has the first priority, as shown in Fig.4.4(a,b). The step [*Find minimum 2 for each component*] should respect that the edge possessing the uniform minimum identifiers of its two points $(i, j), i < j$ is selected, as shown in Fig.4.4(c) where connection (v_4, v_6) is selected by both the two components in parallel and independently. Here i, j are

1-Dimensional identifiers of the input points.

4.3.6 Connect Graph

After the preceding *Find Minimum 2* step, each component finds its closest outgoing vertex v_j in parallel and has its winner vertex w_{v_i} pointing to v_j. Here, this step builds graph edges by adding links to these two corresponding vertices' adjacency list in parallel, while at the same time avoiding repeated links in each node's adjacency list. Three kinds of relationship exist between w_{v_i} and v_j, which are listed below, but only the last case leads to repeat links.

- Firstly, v_j is not a winner vertex;
- Secondly, v_j is also a winner vertex but its corresponding node is not w_{v_i};
- Thirdly, v_j is also a winner vertex and its corresponding node is w_{v_i}.

For both sequential and parallel implementation of this step, the winner node w_{v_i} of a component adds its corresponding vertex v_j to its own adjacency list, while the node v_j adds w_{v_i} under the above three conditions. Parallel implementation of this step uses atomic operations to update these nodes' adjacency list, since two or more components may find the same vertex as their closest outgoing nodes. After this *Connect Graph* step, small components have already been connected into the spanning tree and become new bigger components.

4.3.7 Compact Graph

All previous steps work on the DLVL graph data structure explained in section 3.2, while the fourth step called *compact graph* executes the components' union operation on disjoint set data structure explained in section **??**. The role of this step is to update these disjoint components' root representation that will allow nodes to distinguish other component nodes during the *Find Minimum* steps.

The sequential version follows the classical union-find and path compression operation for this step, while parallel union-find operation has to construct a root merging graph in order to find one uniform new root among the connected sub-trees and then do the union operation. The path compression step can be operated in parallel by associating one thread to one input point.

4.4 Complexity Analysis

About memory occupation, the different EMST steps shown in Fig.4.1 take various memory occupation. Though the steps *Find Minimum 1*, *Connect Graph* and *Compact Graph* occupy $O(N)$ global memory using the DLVL graph representation and disjoint set data structure, local memory occupation by each kernel execution at step *Find Minimum 2* varies with different parallel graph search algorithms on GPU.

About theoretical time complexity, except for the *Find Minimum 1* step, other three steps shown in Fig.4.1 take maximum $O(N)$ sequential time complexity. Time complexity of *Find Minimum 1* depends on the applied local spiral search strategies explained in section 4.3.3. Whereas, no matter which sequential spiral search strategy is applied in practice, time of parallel version is amortized to the many parallel CUDA threads.

4.5 Experiments

We provide three test versions of the proposed EMST algorithm and also implement sequential Prim's algorithm and Delaunay triangulation EMST algorithm using source codes from the Internet [1].

The first version is referred as **EMST-Elias-sequential**. It runs all EMST steps show in Fig.4.1 sequentially, for example the pure local spiral search step explained in section 4.3.3, sequential breadth-first search, sequential union-find algorithm on disjoint set data structure.

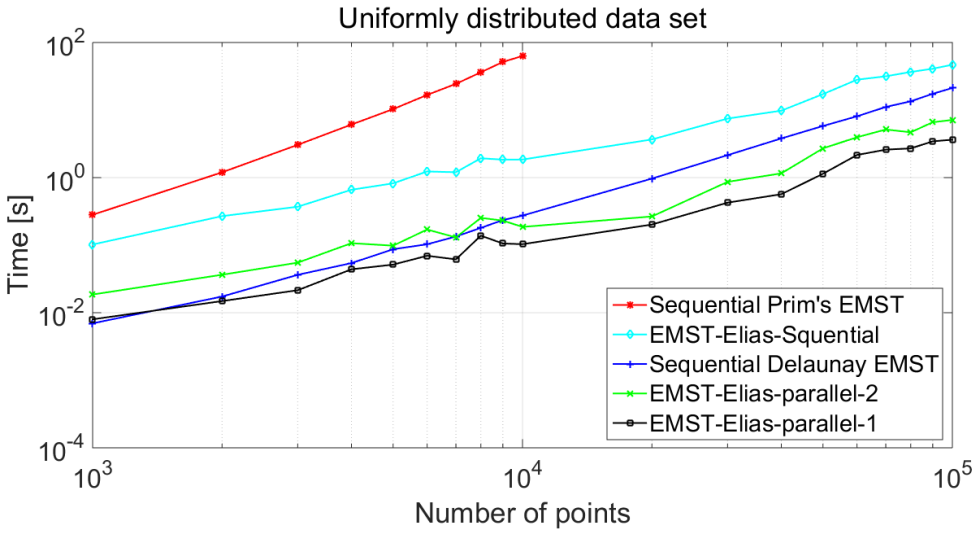
The second version is referred as **EMST-Elias-1**. It only runs the *Find Minimum 1* step in parallel, while other steps are executed sequentially, like breadth-first search and union-find algorithm on disjoint set data structure.

The third version is referred as **EMST-Elias-2**. It applies the parallel spiral search and breadth-first search to make all the proposed EMST steps being executed as kernels on GPU sides.

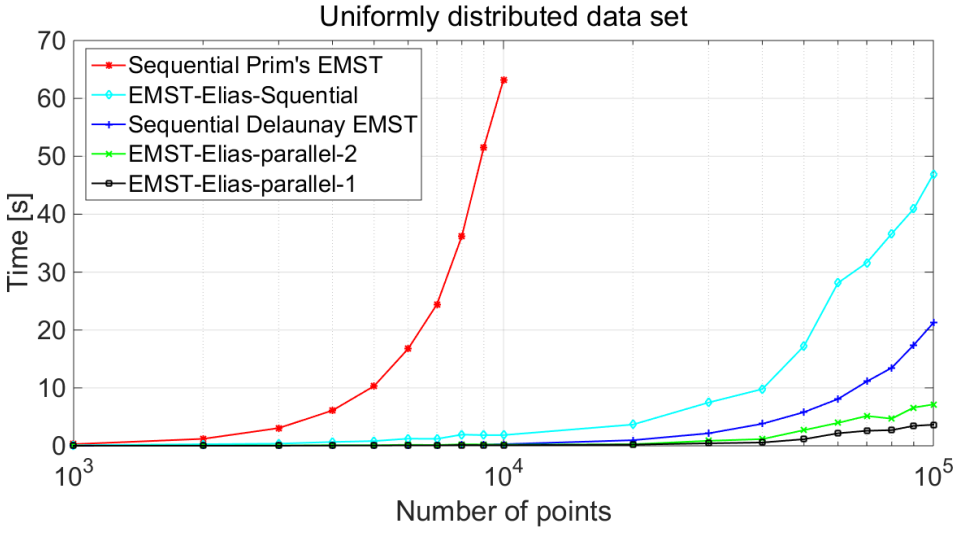
We test the proposed method on uniformly distributed data sets with 20 instances, and 24 National TSPs from TSPLIB [Rei91]. For each test instance, average running time over 10 runs is reported. Running time includes necessary time needed for data transmission between CPU and GPU side. Statistic results are shown in Fig.4.5, Fig.4.6. Visual results are shown in Fig.4.7 and Fig.4.8, we show the way in which this hierarchical Euclidean minimum spanning forest grows in Fig.4.7.

[1] Solving Euclidean Minimum Spanning Tree, source code from GitHub, https://github.com/hqythu/EMST

These tests are implemented with C++ and CUDA Toolkit v8.0 [Nvi10]. Code is compiled on laptop with CPU Intel(R) Core(TM) i7-4710HQ, 2.5 GHz running Windows and GPU card GeForce GTX 850M.



(a)

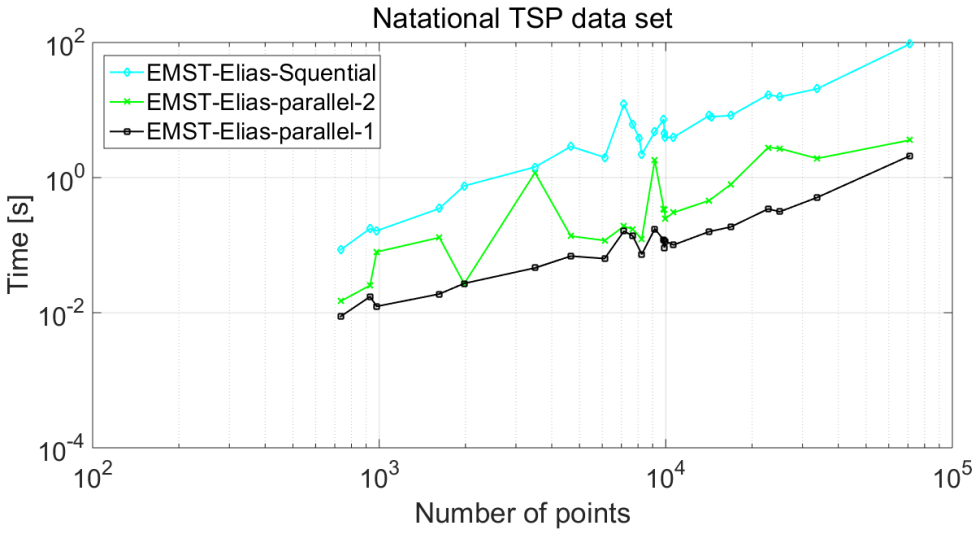


(b)

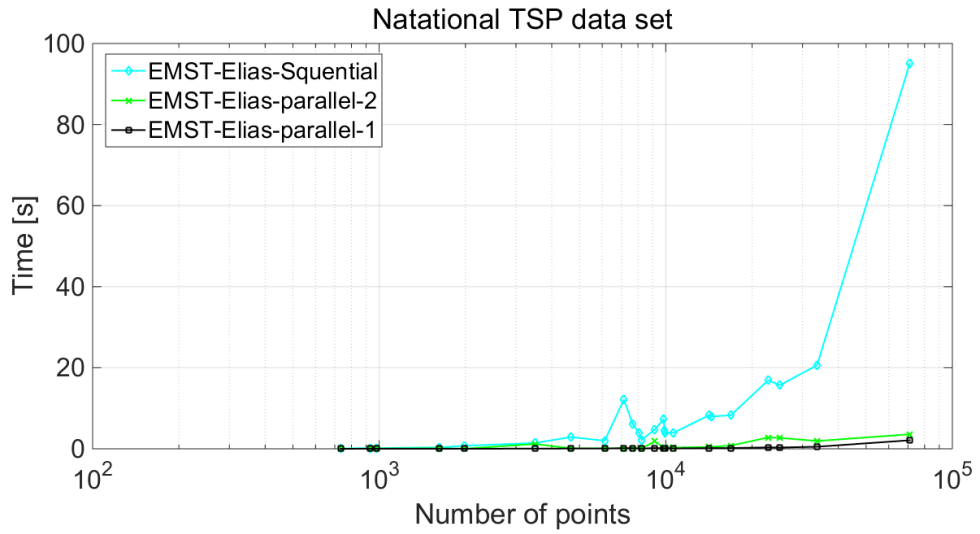
FIGURE 4.5: Average running time of the five EMST algorithms working on uniformly distributed data set. (a): exponential coordinate. (b): general coordinates.

4.6 Conclusion

In this chapter, we present both sequential and data parallel approaches for building hierarchical EMSF/ESMT. The hierarchical minimum spanning forest naturally forms independent clusterings from bottom-to-up and within divide-conquer scheme, which



(a)



(b)

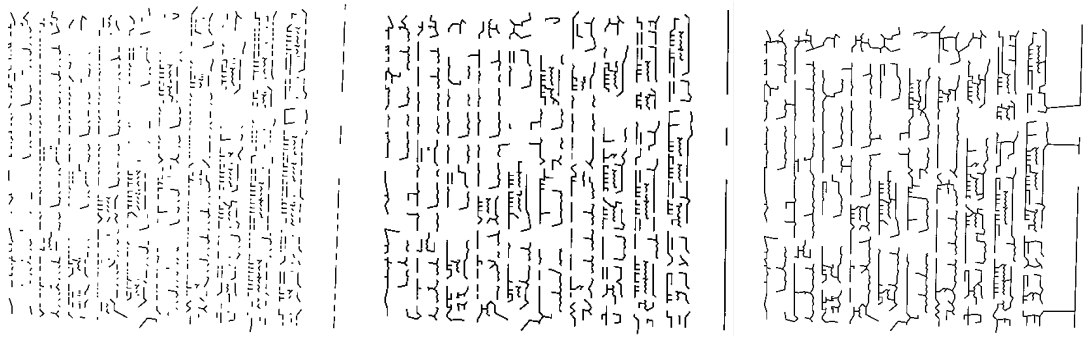
FIGURE 4.6: Average running time of the proposed EMST algorithms working on national TSP data set. (a): exponential coordinate. (b): general coordinates.

can be applied in TSP estimation, image segmentation, cluster analysis, due to the reason that it can roughly estimate the intrinsic structure of a dataset.

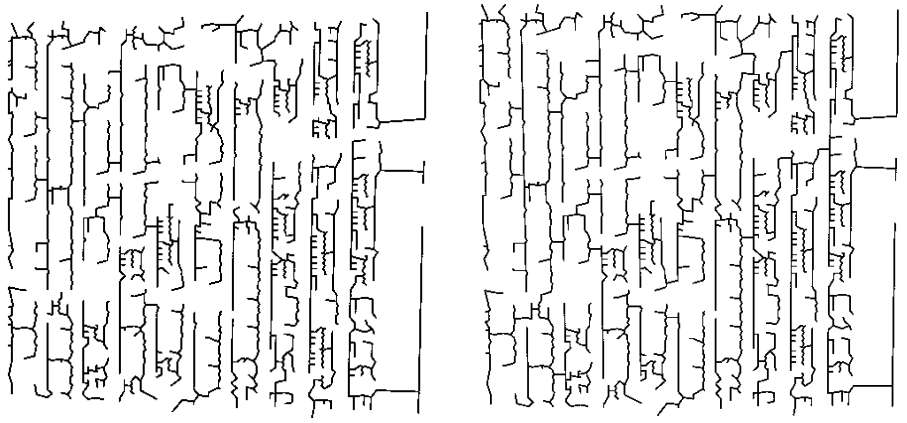
Characteristic of this EMSF/EMST algorithm lies in that it combines the cellular partition-based local spiral search with the Borůvka's algorithm to compute the shortest outgoing edges of each independent sub-tree. Both of them have natural attributes to be parallelized with technique of data decomposition and decentralized control. The experiments are encouraging in that sense when comparing with sequential Delaunay EMST algorithms for instances with more than 10^5 points in the plane.

These proposed procedures to implement this parallel EMSF/EMST algorithm provide

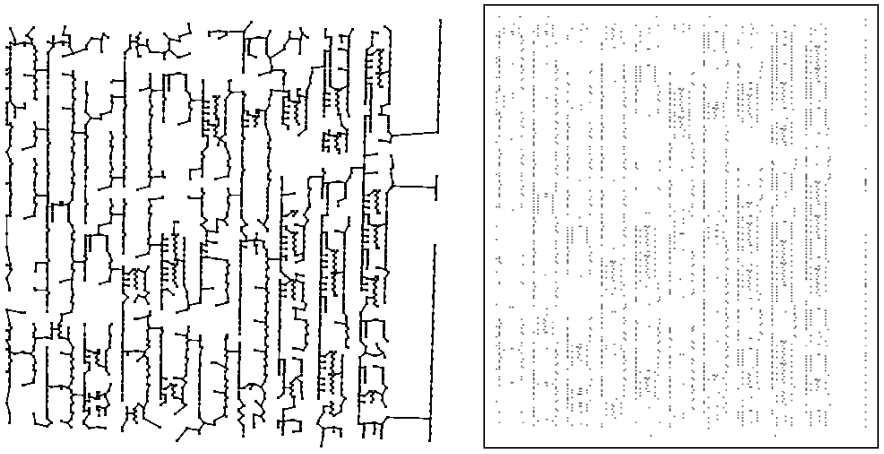
basic alternative solutions for parallelism of many other algorithms, like bichromatic closest pair problem, the nearest insertion and Christofid's TSP heuristics.



(a) 1st iteration (b) 2nd iteration (c) 3rd iteration

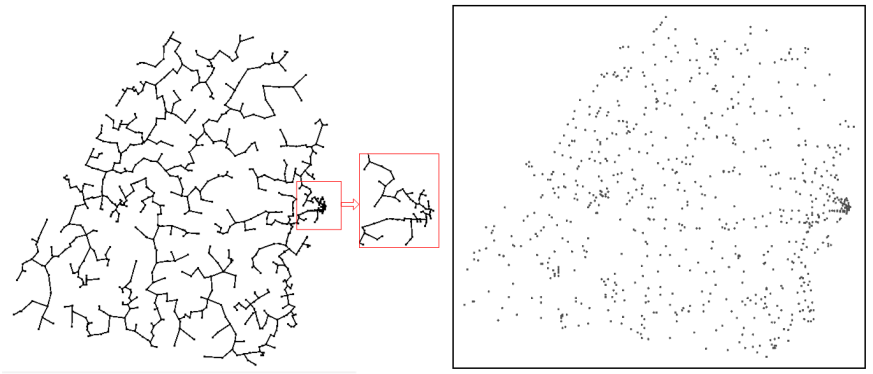


(d) 4th iteration (e) 5th iteration

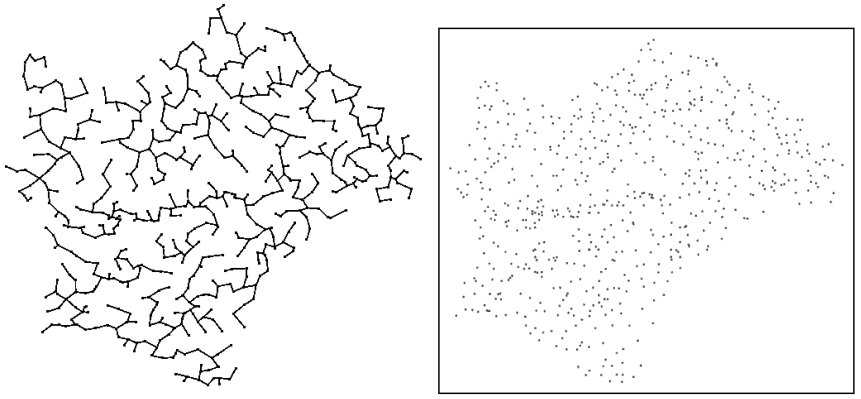


(f) final EMST (g) input points

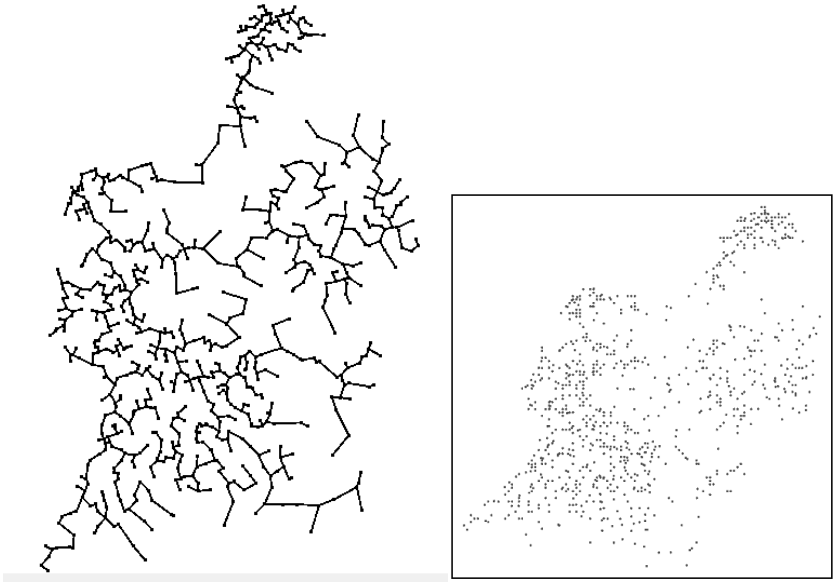
FIGURE 4.7: Test of the proposed hierarchical EMSF method working on rbw2481.tsp provided by TSPLIB [Rei91]. Images (a - e) shows different levels of EMSF built during previous five iterations. (f) shows the final EMST built at final iteration. (g) shows the input points.



(a) uy734.mst (b) uy734.tsp

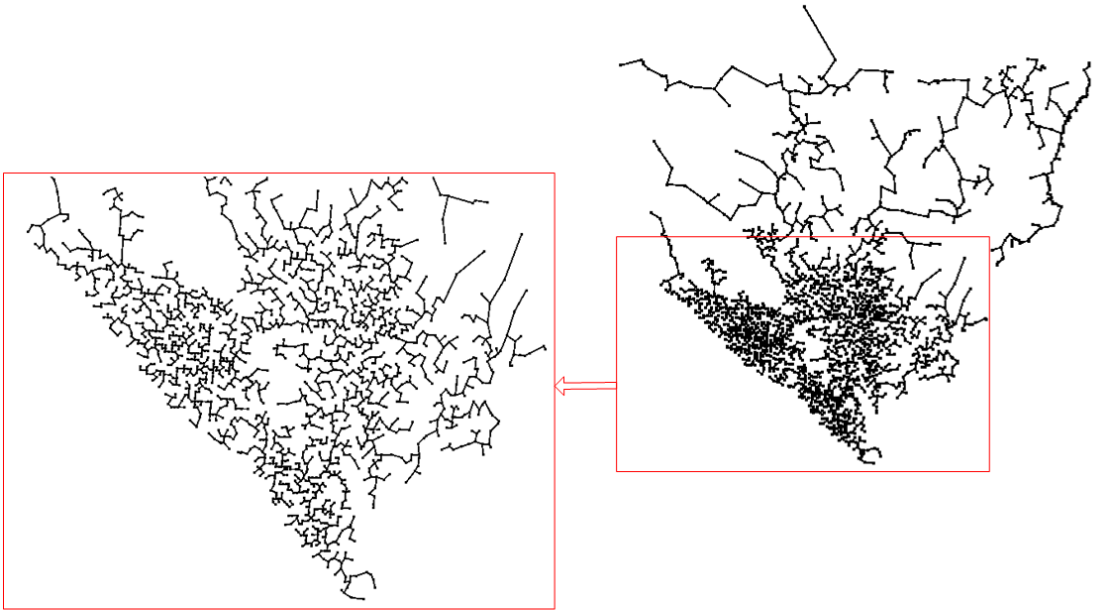


(c) lu980.mst (d) lu980.tsp

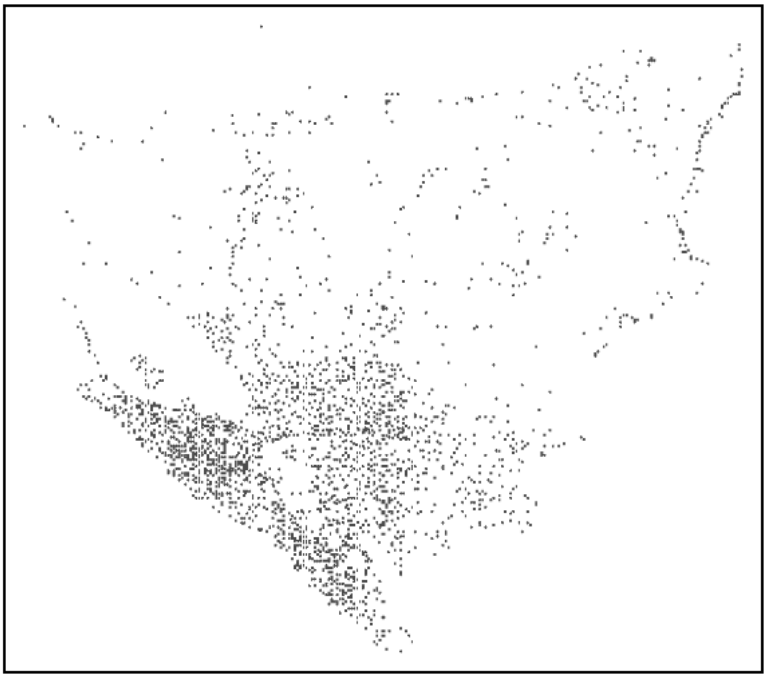


(e) rw1621.mst (f) rw1621.tsp

FIGURE 4.8: Test of the proposed EMST method working on TSP instances, uy734.tsp, lu980.tsp and rw1621.tsp. *xxx*.mst files are EMST results. *xxx*.tsp files are input TSP benchmarks.



(a) nu3496.mst



(b) nu3496.tsp

FIGURE 4.9: Test of the proposed EMST method working on TSP instances, nu3496.tsp. *xxx*.mst files are EMST results. *xxx*.tsp files are input TSP benchmarks.

Chapter 5

Multi moves / parallel evaluations of 2-/3-opt local search to TSP

5.1 Introduction

In this chapter, we will introduce our second original contribution in this thesis, a judicious decision making methodology of offloading which part of the $k-$opt heuristic works in parallel on Graphics Processing Unit (GPU) while which part remains sequential, called ***"parallel local search but sequential selection"***, in order to simultaneously execute, without interference, massive 2-/3-opt moves that are globally found on the same TSP tour or the same integral Euclidean space for many edges as well as keeping high performance on GPU side.

We prove this methodology is judicious and valuable because of our originally proposed sequential non-interacted 2-/3-exchange set partition algorithm taking linear time complexity and a new TSP tour representation, array of ordered coordinates-index, in order unveil how to use GPU on-chip shared memory to achieve the same goal as using doubly linked list and array of ordered coordinates for parallel $k-$opt implementation. We test this methodology on 22 national TSP instances with up to 71009 cities and with brute initial tour solution. Average maximum 997 non-interacted 2-exchanges are found and executed on the same tour of ch71009 TSP instance in one iteration. Experimental comparisons with both classical sequential and a state-of-art parallel 2-opt implementation show that our proposed methodology gets huge acceleration while at cost of little TSP quality degradation.

5.2 Problems of multi 2-/3-opt moves on the same TSP tour

Two main researching branches on multiple 2-/3-opt moves can be categorized in literature: one is parallel local search on disjoint partitions of the same TSP tour [VAS95] or divisions of the Euclidean space [Kar77] (also named as distributed local search [WMC17]), the other one is our proposed massive 2-/3-opt moves that are found on the same global tour or in the same integral Euclidean space [QC17a].

As explained in section 2.5, all current sequential acceleration strategies of 2-/3-opt take polynomial time complexity since they need one loop (2-opt) or two loops (3-opt) for one edge's sequential 2-/3-opt evaluation in order to find one 2-/3-opt move for this edge, and another loop for all edges' sequential evaluation. Parallelism of this 2-/3-opt local search is a natural choice.

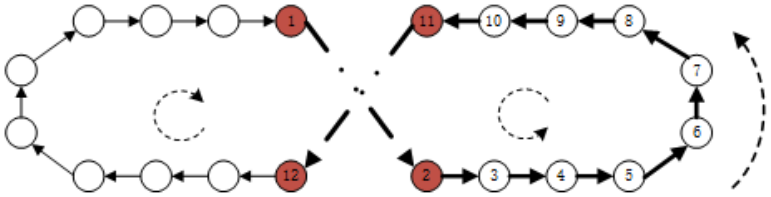
Implementation of 2-opt, 3-opt or k-opt local search algorithms includes three sub-steps: search (evaluate) $k-$opt, execute (move) $k-$opt, and sub-tour reversal. We separately consider the procedure of *evaluation* and the procedure of *move* due to the natural attribute of $k-$opt that, in order to keep the tour valid, implementation of $k-$opt involves reversing original tour order of segment or segments of the original TSP tour solution.

Three unavoidable factors should be considered for designing any parallelism of 2-/3-opt local search under the restriction of the nature attribute of $k-$opt local search. Namely:

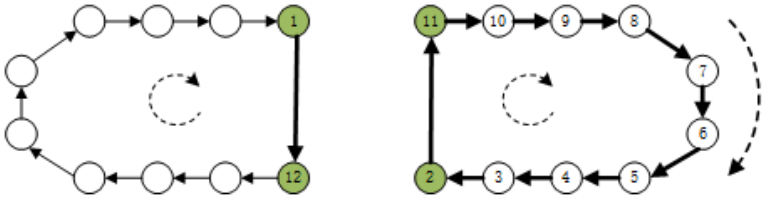
- One correct 2-/3-opt evaluation has to consider original tour order of cities on related tour edges, as shown in Fig.5.1;
- Multiple correct 2-/3-opt moves on the same TSP solution may interact with each other and cut the original TSP tour, as shown in Fig.5.2;
- TSP tour data structure.

Due to previous two factors, related researches on parallel 2-/3-opt implementation can be divided into two categories: one is parallel evaluation for findings one edge's 2-/3-opt move [RS12] [RS13], the other one is multiple 2-/3-opt moves that are found in parallel on separate disjoint partitions of the same tour or Euclidean space [VAS95] [Kar77]. We did not find a fast 2-opt implementation that simultaneously executes massive 2-opt moves that are globally found on the same tour or in the integral Euclidean space for many edges before our published work in [QC17a]. Recent state-of-art work proposed by Rios [ROB+18] mentioned the problem of massive 2-opt moves, while their proposition takes quadratic time complexity to verify non-interacted 2-opt moves.

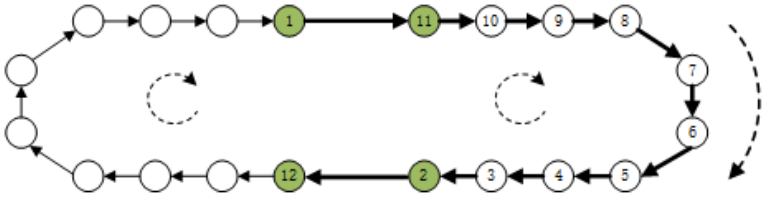
TSP tour data structure should also be an important factor since it influences the efficiency of tour reversal operation, multiple 2-/3-opt moves and parallel implementation on GPU CUDA architecture. For example, in order to re-make an integral TSP tour, it is necessary to take $O(N)$ time complexity after each k-opt move when using an array of ordered coordinates TSP tour data structure and is bad for brute initial TSP solutions that might need millions of selected $k-$opt exchanges before getting to local minimum. However, when working on doubly linked list where each node has two links pointing to its two neighbor cities, this reversal operation only involves changing links of related cities and does not take $O(N)$ time to re-make an integral tour. When considering parallel computing on GPU with coalesced memory access and usage of the limited size of on-chip shared memory, array of ordered coordinates is often preferred than doubly linked list.



(a) Current tour solution s_i.



(b) Wrong 2-exchange would cut tour solution s_i.



(c) One correct 2-opt move leads to valid tour $s_{(i+1)}$ by reversing sub-tour of s_i.

FIGURE 5.1: Tour order plays an important role in iterative 2-opt implementations. ① represents one city's tour order.

In the following sections, we will firstly compare two classical TSP tour data structures and present a newly proposed one for GPU parallel $k-$opt implementation, they are used for later 2-/3-opt implementations. Secondly, we tested a GPU implementation of distributed 2-opt local search that follows the work proposed by Verhoeven et al.[VAS95]. Thirdly, we detailedly introduce our proposed methodology of "parallel local search but

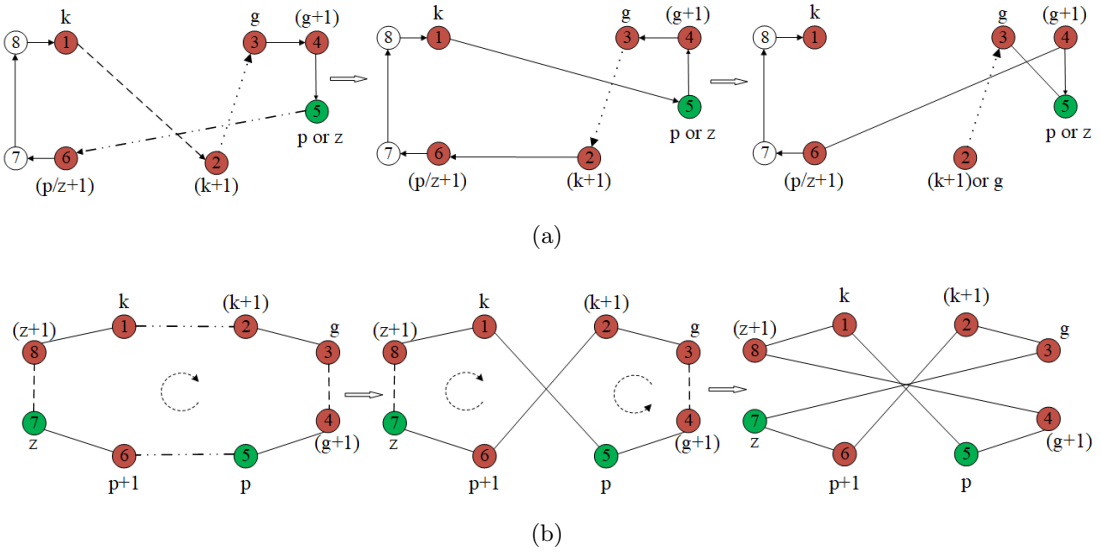


FIGURE 5.2: Cases of multiple 2-exchanges interacting with each other: in (a) 2-exchange $\mathcal{N}_2^{1,5} \rightarrow (e_1, e_5)$ interacts with $\mathcal{N}_2^{2,5} \rightarrow (e_2, e_5)$; in (b) 2-exchange $\mathcal{N}_2^{1,5} \rightarrow (e_1, e_5)$ interacts with $\mathcal{N}_2^{3,7} \rightarrow (e_3, e_7)$. Execution of the two 2-opt moves will cut the original integral tour. ① represents one city's tour order.

sequential selection" that aims to simultaneously select and execute, without interference, massive 2-/3-opt moves that are globally found on the same TSP tour for many edges as well as keeping high performance on GPU side. At last, we give an experimental comparison between various 2-opt implementations.

5.3 TSP Tour Data Structure for GPU Parallel $k-$opt

TSP tour data structure for GPU parallel implementation of iterative $k-$opt algorithms should concern two main aspects: tour reversal operation and coalesced memory access. Two basic data structures, array of ordered coordinates and doubly linked list (DLL), have their own drawbacks and advantages on satisfying these two aspects.

In the following sections, we firstly discuss both advantage and drawbacks of the two basic data structure for GPU parallel $k-$opt implementations. Secondly, we introduce an originally proposed array of ordered coordinate-index that combines advantages of DLL and array of ordered coordinates for GPU parallel $k-$opt.

5.3.1 Array of ordered coordinates *vs* doubly linked list

The array of ordered coordinates (shown in Fig.5.4) preferred by traditional GPU implementation, and the doubly linked list implemented by DLVL (doubly linked vertex

list) shown in Fig.5.5 have their respective advantages and drawbacks for GPU implementation of $k-$opt algorithms.

Advantages of using Array of Ordered Coordinates. Array of ordered coordinates where the order of cities in the array also represents current tour's permutation order at any time [RS13][MW03] is widely used for traditional parallel algorithms on GPU. One reason lies in that multiple threads' memory access to the array can be coalesced, which is an acceleration factor working with the GPU SIMT (single-instruction, multiple thread) structure (explained in section 2.2).

Another reason of using array of ordered coordinates for GPU parallel computing is that the array can be split into segments to be partially reloaded on GPU shared memory when the input TSP size is too large to be entirely memorized by GPU on-chip shared memory [RS13].

Drawback of using Array of ordered coordinate. Considering the necessary reversal operation during $k-$opt implementation, see explanation in section 2.5.2, massive exchanges on TSP tour represented by array of ordered coordinate are not efficient. This is due to the reason that too much cities will be involved in relocation for each exchange when using this data structure, as one case shown in Fig.5.4 for single 2-exchange $\mathscr{N}_2^{2,10}$ shown in Fig.5.3. This kind of relocation takes $O(N)$ sequential time complexity that can not be ignored when $k-$opt implementations perform in an iterative way (Alg.2.5.1) for instances with more than 10^3 cities and with brute initial solution [Nil03], since it might need millions of selected $k-$opt exchanges before getting to local minimum. Besides, changing one node in the array of ordered coordinates will influence two oriented edges, which is not suitable for executing massive 2-/3-opt exchanges on the same tour.

Advantage of Using Doubly Linked List. Doubly linked list (DLL) implemented by DLVL (doubly linked vertex list) shown in Fig.5.5, where each city has two links pointing to its two neighbor cities of current TSP tour, does not take that much time for doing this reversal job because an exchange on DLL only needs to change links of related cities. We do not specify which one of these two neighbor cities is "previous" or "next" tour city in our implementation, since this kind of role changes during the $k-$opt implementation.

Using DLVL (doubly linked vertex list) shown in Fig.5.5 as TSP tour data structure also provides a key property for executing massive 2-opt and 3-opt moves without interference on the same tour: changing one node in the array of ordered coordinate will influence two oriented edges in current tour solution, while changing one of the two links possessed by a node in DLL only influences one oriented tour edge.

These advantages could be explained in Fig.5.6. Fig.5.6 shows the way in which the same 2-exchange $\mathcal{N}_2^{2,10}$ shown in Fig.5.3 happens on doubly linked list.

Drawback of using Doubly linked List. However, memory access pattern is scattered as shown in Fig.5.5 when using this DLL tour representation. And large-scale TSP tour represented by DLL can not work on GPU limited size of on-chip shared memory when the size of TSP instance exceeds the size of on-chip shared memory, because the array of doubly linked route shown in Fig.5.5 should necessarily have global access to the array of coordinates that stores all input cities.

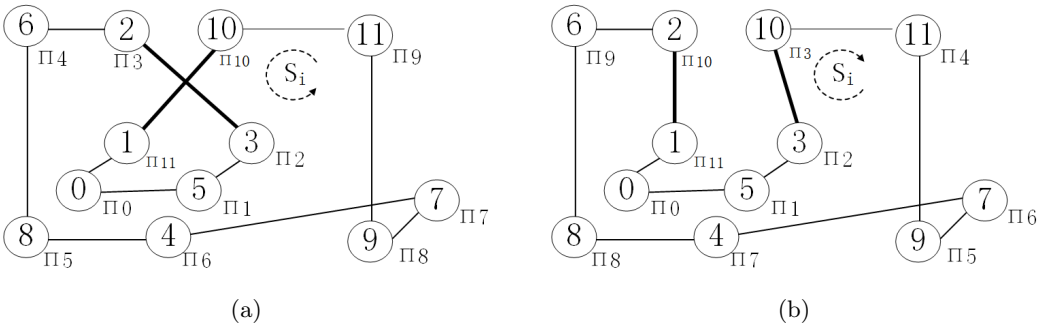


FIGURE 5.3: (a)Tour solution s_i possessing two candidate 2-exchanges; (b)New tour after executing the 2-exchange $\mathcal{N}_2^{2,10}$ on tour s_i. ① represents one city' input indice in the array of coordinate, π_1 indicate the city's tour order.

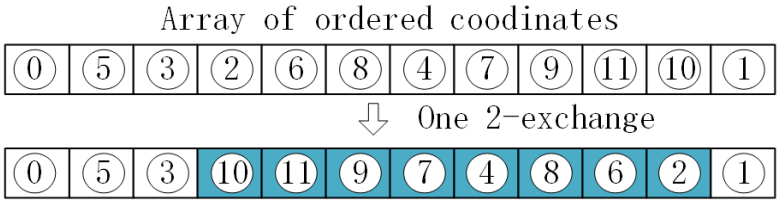


FIGURE 5.4: The 2-exchange $\mathcal{N}_2^{2,10}$ in Fig.5.3(a) happened on TSP tour represented by array of ordered coordinates.

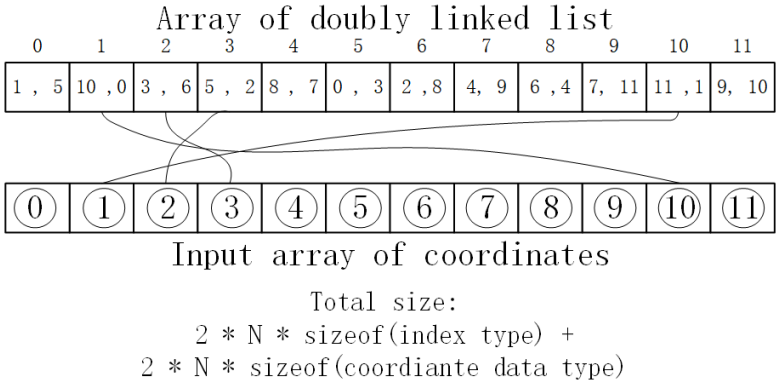


FIGURE 5.5: TSP tour represented by doubly linked list that is explained in section 3.2.

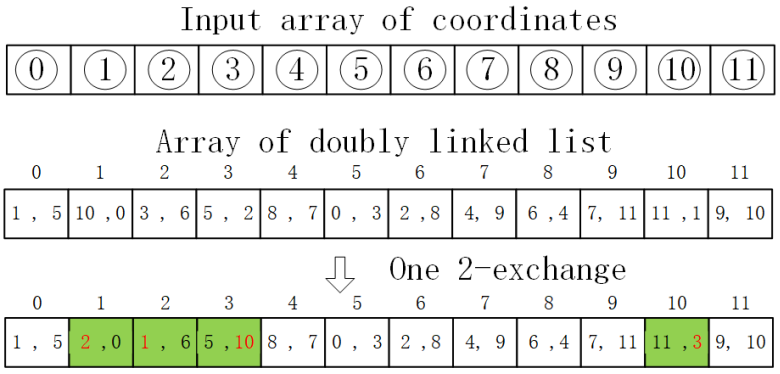


FIGURE 5.6: The 2-exchange $\mathcal{N}_2^{2,10}$ in Fig.5.3(a) happened on TSP tour represented by doubly linked list.

5.3.2 Array of ordered coordinates-index

We have compared both advantages and drawbacks of using doubly linked list and Array of ordered coordinates as TSP tour data structure for GPU parallel $k-$opt implementation in section 5.3.1.

A question appears to be can we find another TSP tour data structure that can combine advantages of these two data structure? The answer is yes, we propose an array of ordered coordinates-index to represent TSP tour.

Each element of the **Array of Ordered Coordinates-Index** consists of one city's Euclidean coordinates (usually two float values) and an index of this city. This index is identifier of a city on the array of doubly linked list, which is same with the index of input array of coordinate and is used for later registering candidate k-exchanges' information during the parallel $k-$opt local search. This array is also an ordered array where order of a city in the array also represents its permutation tour order. Fig.5.7 is an example to represent the tour solution shown in Fig.5.3(a).

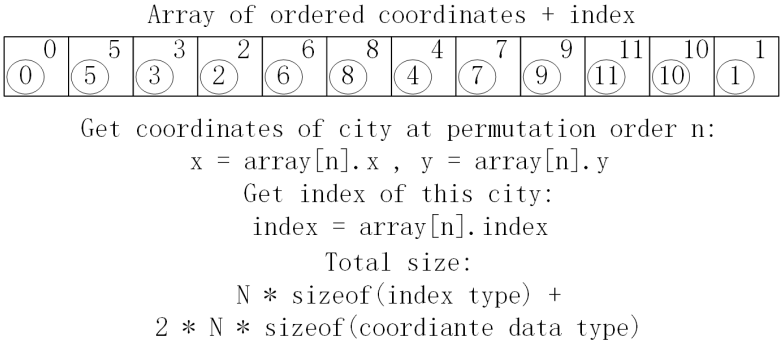


FIGURE 5.7: Array of ordered coordinates-index for GPU parallel $k-$opt implementations that combines the advantages of doubly linked list and array of ordered coordinates.

Advantages of Array of Ordered Coordinates-Index. Array of ordered coordinates-index combines advantages of doubly lined list with array of ordered coordinates. For GPU parallel implementation, this new TSP data structure allows coalesced memory access to GPU device memories, and further enable subdivisions of large-scale TSP tour can be reloaded on GPU limited shared memory as same as those traditional GPU $k-$opt implementations [RS13], and at the same time it provides an interface for algorithms to take advantage of doubly linked list, explained in section 5.5.

5.4 Distributed 2-opt local search

According to the analysis in section 5.2 about the problematic of multiple 2-/3-opt moves on the same TSP tour, with the consideration of tour reversal operation, a straightforward parallelism of 2-opt local search is to divide current TSP solution between processor, each one operating a local search with its own part of data (namely, disjoint partial tour). This kind of parallel 2-opt local search is similar to the work proposed by Verhoeven et al.[VAS95].

However, different local search strategies can still be adopted to independently optimize these distributed disjoint partial tours, for example, all edges of one partial solution are optimized serially by one processor, or only one edge of one partial solution is optimized by this processor.

Comparing with the strategy to serially optimize all edges of one partial solution by one processor, which will lead to the fact that a certain edges of this partial tour are optimized along a very short length (path) of local neighbors, we adopt another strategy of optimizing only one edge of one partial solution while the tour partition is dynamically moved along current tour, this will ensure that the length (N/M) of local neighbors for each edge's optimization.

We call the later local search strategy as *distributed 2-opt with dynamic disjoint tour partition* [QC17b], which is illustrated in Fig.5.8 and works in following way: Each partial tour has one active edge that will be optimized; Each active edge only checks its 2-opt optimization along one same tour direction until it encounters next active edge. In the opposite tour direction, every edge gets a chance to be optimized among its another local N/M neighboring edges. After these m edges' optimization, next m tour partitions are activated and the algorithm passes to next m edges' parallel optimization.

Its detailed implementation is shown in Fig.5.9. For a TSP instance with N size, the algorithm starts with a random active city π_i and pass to one of its two links to be next visited city. To make sure that multiple 2-opt optimization happens at correct direction

and do not create independent sub-tours, the algorithm needs to mark each city's tour order $i, 0 < i < N$ along one of the two orientations of TSP circle, which is named as step of "refresh TSP tour order" in Fig.5.9 and done easily by using a simple operation that each city finds its next connected link taking advantage of the doubly linked list. For opposite direction, starting from the same node p_i but pass to its another link as the secondly visited city, then the integral tour direction is inverted by just finding each city's unvisited link. After this step, the new tour information is copied to GPU global memory.

With every city's tour order being known already, m edges are activated to search and execute their 2-opt optimization simultaneously on device side with the kernel function shows in Alg.5.4.1. TSP tour input of Alg.5.4.1 is represented by DLVL (doubly linked vertex list) and is oriented with every city π_i having its unique tour order $i = 0, 1, 2...(N-1)$ according to current oriented tour solution s. For each active node $\pi_{a_m, m=(0,1,...M)}$, each thread runs the same kernel function in Alg.5.4.1.

Since we use each cities' tour order to decide which m edges are activated simultaneously along the tour, this parallel 2-opt local search with disjoint tour division has to refresh tour order after each exchange. To reduce the time taken for copying data from GPU to CPU side, we choose to refresh tour order directly on GPU side, shown as "Kernel call 2" in Fig. 5.9. As we optimize these m edges in the same tour direction, the original tour order is influenced only inside each independent N/M sub-tours, leading to the fact that each sub-tour's reversal operation can be done in parallel. After this step, the algorithm begins with next m edges' optimization in next inner loop until all cities have one chance to be optimized in one outer loop.

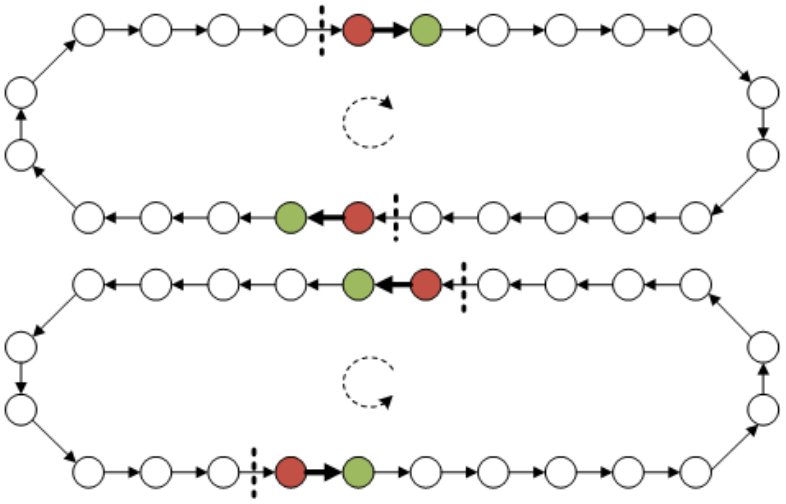


FIGURE 5.8: Distributed 2-opt local search with dynamic tour partition.

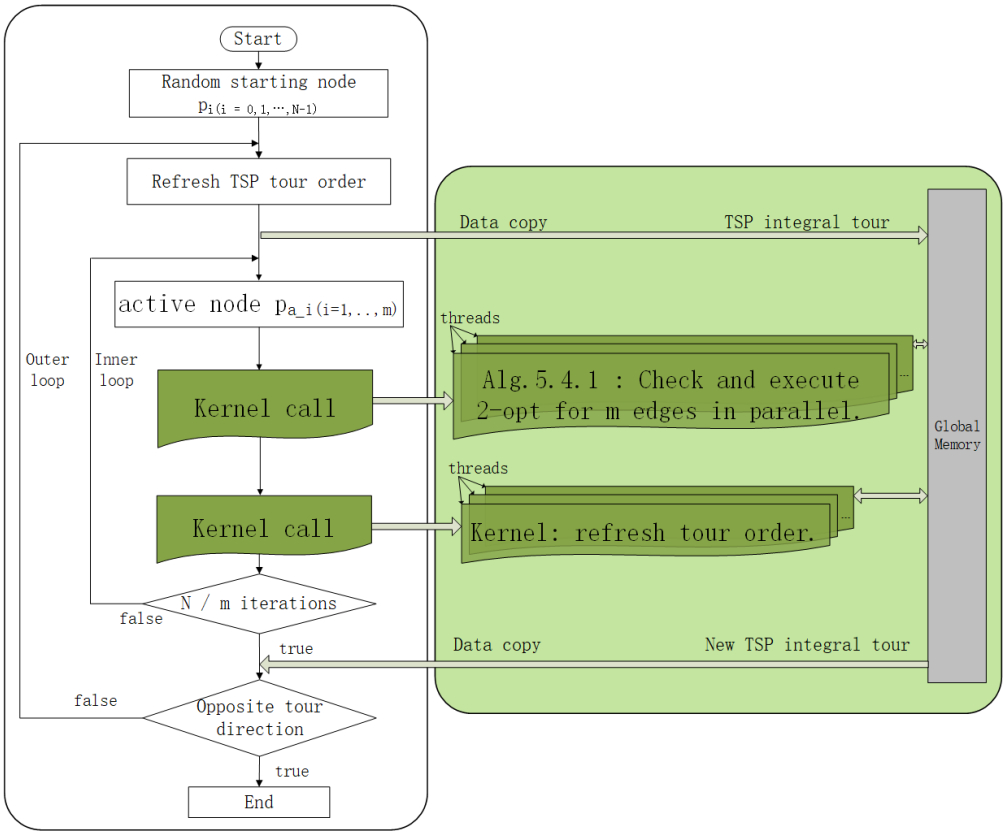


FIGURE 5.9: One run of distributed 2-opt local search with dynamic tour division. One inner loop optimizes m edges simultaneously. One outer loop optimizes all edges once in one tour orientation.

Algorithm 5.4.1 Kernel: Distributed 2-opt local optimization with dynamic disjoint tour division working on GPU device memory.

Require: Current active node π_{a_m}, marked as the first node p_1 for a candidate 2-exchange;
1: Choose one of the two links from p_1 to be p_2 according to current tour direction;
2: Mark exchange link positions for p_1 and p_2 separately;
3: Choose one of the two links from node p_2 to be p_t, make sure $p_t \neq p_1$;
4: **if** $p_t = \pi_{a_{m+1}}$ **then return**
5: **else**
6: $p_3 \leftarrow p_t$;
7: **while** $p_3 \neq \pi_{a_{m+1}}$ **do**
8: Choose one of the two links from node p_3 to be p_4 according to current tour direction;
9: Mark exchange link positions for p_3 and p_4 separately;
10: **if** $dis(p_1, p_3) + dis(p_2, p_4)$ less than $dis(p_1, p_2) + dis(p_3, p_4)$ **then**
11: Execute 2-exchange;
12: break;
13: **else**
14: $p_3 \leftarrow p_4$
15: **end if**
16: **end while**
17: **end if**

5.5 Methodology of parallel evaluation but sequential selection

With the distributed 2-opt local search explained in section 5.4, it is easy to find out drawback of this kind of distributed 2-opt local search implementations: the length of each disjoint partial tour is fixed, leading to the limitation that it is hard to get a balance between the neighborhood length of N/m for each edge' optimization and the total quantity of m edges being optimized in parallel. Namely, it is hard to obtain each edge's optimization along the global tour, meanwhile, obtain a significant acceleration over sequential implementations.

One question appears to be "can massive 2-opt moves that are found globally along the same tour be executed simultaneously ?" The answer is yes, while the necessary tour reversal operation of $k-$opt implementation makes massively concurrently global $k-$opt moves become complex. As shown in Fig.5.10, execution of the two 2-opt moves in (b,c) will cut the original integral tour.

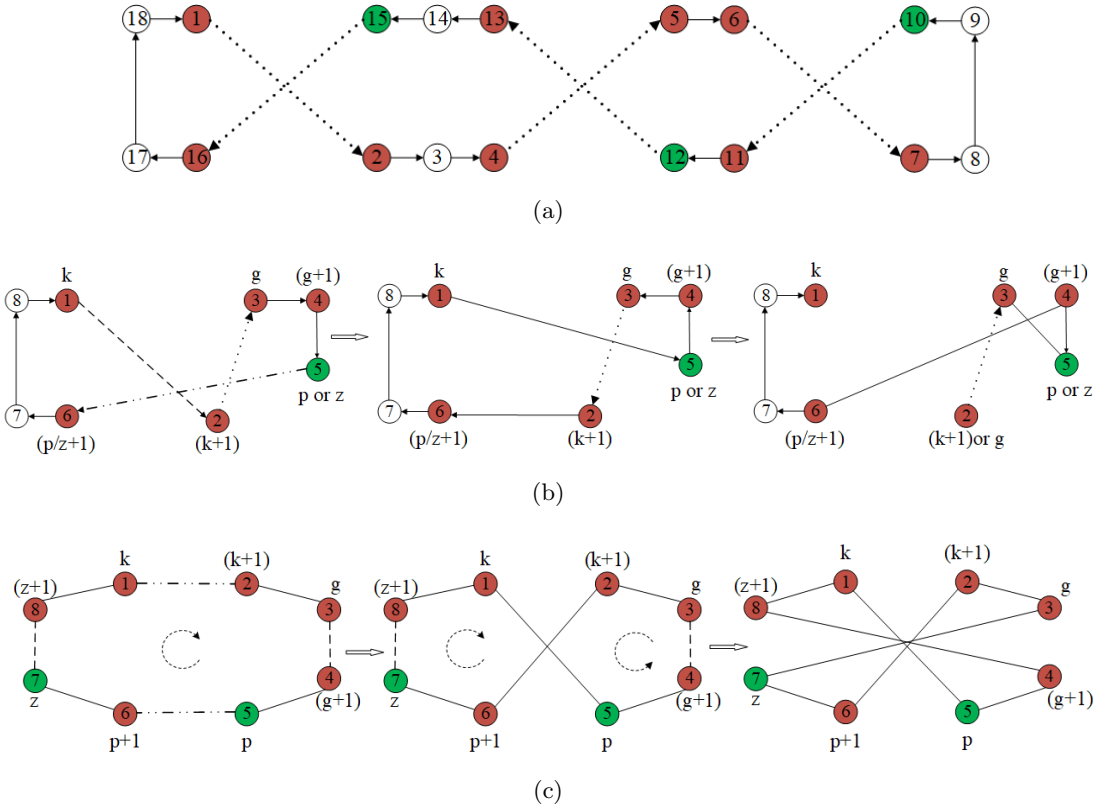


FIGURE 5.10: Cases of massive 2-exchanges found on the same TSP tour. (a) One case that massive 2-exchanges do not interact with each other; (b, c) Cases of multiple 2-exchanges interacting with each other: in (b) 2-exchange $\mathscr{N}_2^{1,5} \rightarrow (e_1, e_5)$ interacts with $\mathscr{N}_2^{2,5} \rightarrow (e_2, e_5)$; in (c) 2-exchange $\mathscr{N}_2^{1,5} \rightarrow (e_1, e_5)$ interacts with $\mathscr{N}_2^{3,7} \rightarrow (e_3, e_7)$. ① represents one city's tour order.

However, when working on doubly linked list TSP data structure, these massively *non-interacted 2-exchanges* shown in Fig.5.10(a) can be executed in parallel without cutting the tour or tour reversal operation after each 2-exchange.

These massively *non-interacted 2-exchanges* can not be simultaneously executed on TSP tour data structure using array or ordered coordinates, as we have explained in section 5.3.1. Nevertheless, in order to obtain high performance GPU $k-$opt local search implementation, it is a better choice to use array or ordered coordinates as TSP tour data structure.s

Now, the question becomes "how to efficiently select massively *non-interacted 2-exchanges* as well as keep high performance on GPU side ?"

To answer this question, we propose a judicious decision making methodology of offloading which part of the $k-$opt heuristic works in parallel on Graphics Processing Unit (GPU) while which part remains sequential, in order to simultaneously execute, without interference, massive 2-/3-exchanges that are globally found on the same TSP tour for many edges as well as keeping high performance on GPU side. Outline of this methodology is shown in Alg.5.5.1. It consists of two parts: various parallel $k-$opt implementations that can find candidate exchanges along current global tour for many (or for all) edges edges (in section 5.5.1), but only these non-interacted exchanges can be sequentially selected in order to be executed simultaneously in one iteration (in section 5.5.2), which is simplified as **"parallel local search but sequential selection"** as shown in Alg.5.5.1 [QC17a].

This methodology is judicious since intervention of a sequential $O(N)$ time complexity tour reversal operation is unavoidable for each 2-/3-opt move when using array or ordered coordinates as TSP tour data structure for high performance GPU computing that considers coalesced memory access and usage of limited on-chip shared memory.

This methodology is valuable because of an originally proposed sequential $O(N)$ time complexity non-interacted 2-exchange set selection algorithm and a new TSP data structure, array of ordered coordinates-index, in order to unveil how to use GPU on-chip shared memory to achieve the same goal as using doubly linked list and array of ordered coordinates for parallel $k-$opt implementation. Besides executing massive 2-/3-opt moves, these two innovations further reduce total amount of costly tour reversal operations and data transmission between host and GPU sides for iterative 2-/3-opt implementations.

We introduce below the two main steps of the proposed "parallel local search but sequential selection" methodology shown in Alg.5.5.1.

Algorithm 5.5.1 Methodology of offloading which part of the $k-$opt heuristic works in parallel on GPU while which part remains sequential, named as "*parallel local search but sequential selection*".

1: **while** Before termination condition **do**
2: Prepare array of ordered coordinates-index for GPU parallel computing on shared memory;
3: Data copy to GPU;
4: Various parallel $k-$opt implementations that can find candidate exchanges for many (or for all) edges along current global tour;
5: Data Copy information of these candidate exchanges to CPU;
6: Sequentially select non-interacted 2-opt exchanges that can be executed without interference;
7: Concurrently execute these selected non-interacted exchanges;
8: **end while**

5.5.1 Parallel 2-opt evaluations

A certain parallel $k-$opt implementation searches (but not executes) many edges' or all edge's $k-$opt optimization on the same global tour, which mainly takes advantages of GPU parallel reading ability on device memory. Comparing with the distributed 2-opt local search with disjoint tour partition presented in section 5.4, key difference lies in that: firstly, the searching range for one edge's optimization can be globally along the integral tour without limitation of the fixed range of each disjoint partitions; secondly, each thread only takes responsible for checking possible $k-$opt optimization, ***not*** for executing k-exchange found by these $k-$opt checks. However, various parallel local search operators should be adapted for later selecting non-interacted k-exchanges. We take 2-opt as an example:

- Firstly, each 2-opt check $\mathcal{N}_2^{kp} \rightarrow \{e_k, e_p\}$ is oriented along the same tour direction, edge $e_p = (\pi_p, \pi_{p+1})$ is visited after edge $e_k = (\pi_k, \pi_{k+1})$ in current oriented tour solution $s = (\pi_0, \pi_1, ..., \pi_{(N-1)})$.
- Secondly, for each 2-opt check between two edges $\{e_k, e_p\}$ that optimizes the original tour, the parallel 2-opt local search implementation has to memorize index of city π_p only to city π_k. This is information of candidate 2-exchanges for later selecting and executing non-interacted 2-exchanges $\mathcal{N}_2^{kp}$ only by visiting the city π_k.

To satisfy these two requirements as well as keep high performance on GPU parallel computing, the newly proposed TSP tour data structure, array of ordered coordinates-index, is an ideal choice for TSP tour representation.

In this section, we will present three GPU implementations of 2-opt evaluation strategies, namely adaptation about Rocki's high performance GPU 2-opt local search, overlapping tour partition and 2-opt search with Euclidean nearest neighborhood. Though they are not our key contribution, design of these GPU parallel 2-opt local search works based on the assumption that a GPU card possesses capability to search, by each processor, a certain quantity of 2-/3-opt checks. These parallel local search implementation takes maximum $O(N)$ threads. Instead of using $N \times N$ distance matrix, all these parallel implementations of 2-opt search take maximum $O(N)$ size of GPU global memory, $O(1)$ size of local memory for each kernel launch.

5.5.1.1 with Rocki's methods

Rocki's parallel 2-opt local search framework [RS13] can be briefly explained with Fig.5.11. The total $\frac{N*(N-1)}{2}$ 2-opt checks to find the best 2-opt move for an integral tour are distributed in the color boxes. Every pair of numbers in each box indicate current tour order of two cities π_k and π_p for $\mathcal{N}_2^{kp}$ separately. Due to the algorithm works on array of ordered coordinates data structure, each number in the box actually represent one oriented edge along current tour. One run of these $\frac{N*(N-1)}{2}$ 2-opt checks will produce massive candidate 2-exchanges along the global tour for many edges, while the basic data structure in Rocki's implementation can only enable the algorithm select one best move. For example the two red boxes marked in Fig.5.11 will find the two 2-opt moves shown in Fig.5.3(a), while Rocki's implementation has to perform two times of this parallel 2-opt runs and tour reversal operations to search and execute these two 2-opt optimization shown in Fig.5.3(a).

Adaptation about Rocki's originally high performance GPU parallel 2-opt local search mainly involves: firstly, change the basic TSP tour data structure from array of ordered coordinates to the newly proposed array of ordered coordinates-index explained in section 5.3.2; secondly, instead of registering the best 2-opt moves for the integral tour, we register the best 2-opt moves for each edge $e_k = (\pi_k, \pi_{k+1})$ by memorizing city π_p to city π_k. Rocki's parallel 2-opt framework naturally satisfies the requirement that each 2-opt check is oriented along the same tour direction.

This array of ordered coordinates-index is prepared together with the RefreshTourOrder step in Alg.5.5.1, this operation is necessary when using arrays of ordered coordinates to represent TSP tour order, which takes $O(N)$ complexity sequentially and happens only once after executing massive 2-exchanges with the proposed non-interacted 2-exchanges set partition algorithm in section 5.5.2. Here, attention should be taken when preparing

the ordered array of coordinates-index in order to satisfy the 2-opt checks listed in last row of Fig.5.11.

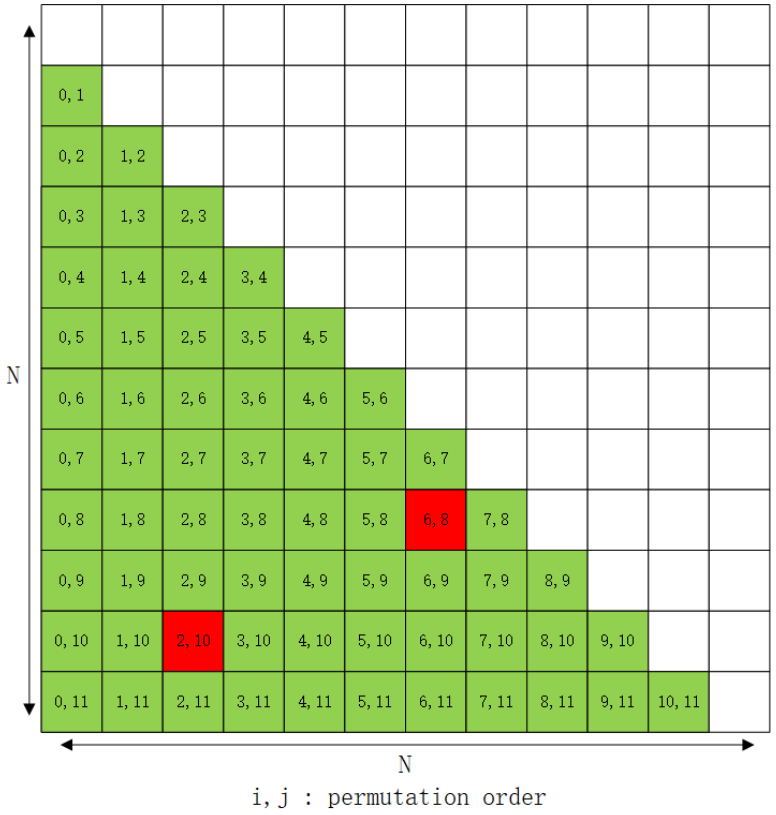
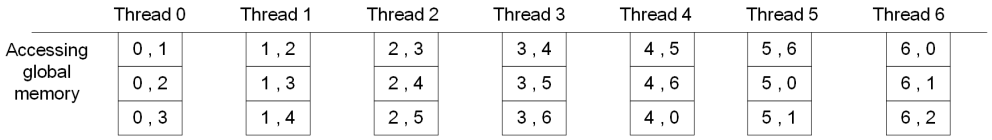


FIGURE 5.11: GPU parallel 2-opt scheme proposed by Rocki [RS13] to perform the complete $\frac{n*(n-1)}{2}$ 2-opt checks in parallel.

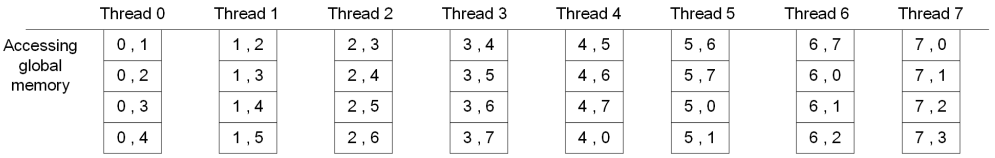
5.5.1.2 with overlapping tour partitions

Overlapping tour partition is actually a brute-force parallelism of the sequential acceleration strategy mentioned in section 2.5.2 that only keeps m local neighbors to search one edge's $k-$opt optimization [Nil03]. Here, many edges search their $k-$opt optimization along its m local neighbors that lie in the same tour direction in order to ensure each 2-/3-opt exchange follows one same tour direction, as shown in Fig.5.12. These edges' local search are executed by N processors in parallel on GPU device memory.

The maximum m is set to be $N/2$ in Fig.5.12 in order to ensure that all possible 2-/3-checks of the global tour are evaluated in one run of parallel local search, namely $\frac{N*(N-1)}{2}$ checks for one 2-opt run and $\frac{N*(N-1)*(n-2)}{6}$ for one 3-opt run explained in section 2.5.2, though this m can be set a shorter length.



(a) Input TSP size $N = 7$.



(b) Input TSP size $N = 8$.

FIGURE 5.12: Parallel 2-opt local search with overlapping tour division. Each of maximum $O(N)$ threads takes responsible for maximum $m = N/2$ 2-opt checks. Every pair of numbers in these two figures indicate current tour order of two cities π_k and π_p separately, each number represents one oriented edge along current tour.

5.5.1.3 with nearest neighborhood search

Parallel $k-$opt local search with Euclidean nearest neighborhood searching algorithm mainly takes advantages of geometrical information centering the query edge in Euclidean space. Here each edge searches its $k-$opt optimization in the same entire Euclidean space [QC17a].

Given current tour $s \in S$ according to the TSP tour formulation explained in section 2.5.1, where $s = (\pi_0, \pi_1, \pi_2, ..., \pi_{(N-1)})$, edge $e_i = (\pi_i, \pi_{i+1})$ and $i = 0, 1, ..., (N-1)$ indicates cities' current tour order along s. With the closest point finding based on Elias' methods explained in section 4.3.3, each query edge $e_i = (\pi_i, \pi_{i+1})$ searches its neighbor edges $e_j = (\pi_j, \pi_{j+1}), (j+1) mod N = 0$ through searching π_i's neighbor cities π_j. We associate one thread to exactly one edge's geometrically $k-$opt optimization, each spiral search stops when the first 2-opt optimization is found or exceed a certain searching radius.

5.5.2 Serially select multiple 2-/3-opt moves with linear time complexity

GPU parallel $k-$opt local search (explained in section 5.5.1) that aims to find candidate exchanges for many edges along the same global tour or in the same Euclidean space will produce massive candidate k-exchanges that optimize the original tour independently and correctly as shown in Fig.5.10. However, executing these massive k-exchanges concurrently or simultaneously without selection will certainly cause problems, for example, cut the initial tour into independent parts, while executing these massive 2-exchanges

one by one and refreshing tour order after each exchange is meaningless when comparing with the sequential versions. Our proposition tries to select these non-interacted 2-/3-exchanges on the same tour, in order to execute them concurrently or simultaneously.

At the preceding steps presented in section 5.5.1, various parallel local search strategies find massive candidate 2-exchanges along the same oriented tour solution $s \in S$, $s = (\pi_0, \pi_1, ..., \pi_{(N-1)})$ and register each candidate 2-exchange $\mathcal{N}_2^{k,p} \Rightarrow (e_k, e_p)$ information only to city π_k.

Here in this step, we propose an original sequential algorithm to select these non-interacted 2-/3-exchanges on the same TSP solution $s \in S$, which takes $O(N)$ complexity.

In the following sections, we begin with the definition of non-interacted k-exchanges distributed on the same tour; then we provide a detailed algorithm to select non-interacted 2-exchanges, which can be easily extended to select non-interacted 3-exchanges; Execution of these non-interacted 2-exchanges are explained in the third section; Lastly, we explain extensions to massive 3-opt moves.

5.5.2.1 Definition of non-interacted k-exchanges

Assuming massive candidate 2-exchanges that optimize the original tour have been found by a certain GPU parallel 2-opt search implementation and they would not independently cut the original tour solution s, executing random two 2-exchanges among them without reversal operation would also cut s. This is due to the reason that previous 2-exchange may have changed cities' tour order of the sub-tour (explained in section 2.5.2) related to the later 2-exchange, as shown in Fig.5.10(b,c), this case is named that these two 2-exchanges interact with each other.

However, there are cases where random two 2-exchanges do not interact with each other, one case is shown in Fig.5.10(a). These massive 2-exchanges that do not interact with each other can be executed along the same TSP tour s_i represented by doubly linked list without reversal operation for each exchange.

We give a definition of **non-interacted k-exchanges** below in Definition 1, and prove it with the typical 2-exchanges. Concept of **sub-tour** produced by each k-exchange is explained in section 2.5.2.

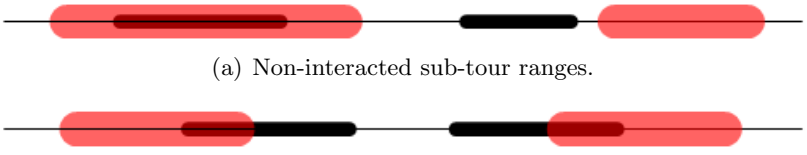
Definition 1. Given two correct candidate k-exchanges found on the same tour. It exists either of following two characteristics between two interacted k-exchanges, otherwise, they are non-interacted k-exchanges.

- There are identical edges being removed or newly inserted by these two k-exchanges.
- Two special tour segments produced by these two k-exchanges cross with each other. The special tour segment indicates that the sub-tour begins with the starting city and end with the last city of one k-exchange along current tour orientation.

For example, given two candidate 2-exchanges $\mathscr{N}_2^{kp} \Rightarrow (e_k, e_p)$ and $\mathscr{N}_2^{gz} \Rightarrow (e_g, e_z)$ that are found on the same tour s_i and do not independently cut s_i. **If** any pair of $k, p, g, z, (k, p, g, z+1) mod N$ is identical, like $p = z$ in Fig.5.10(b) *OR* there is crossing between the two sub-tour ranges $\pi_{(k+1)} \rightarrow \pi_p$ and $\pi_{(g+1)} \rightarrow \pi_z$ as shown in Fig.5.13(b), like s_i goes $(\pi_k \rightarrow \pi_{(k+1)} \rightarrow \pi_g \rightarrow \pi_{g+1} \rightarrow \pi_p \rightarrow \pi_{p+1} \rightarrow \pi_z \rightarrow \pi_{z+1})$ in Fig.5.10(c), execution of these two 2-exchanges $\mathscr{N}_2^{gz}$ and $\mathscr{N}_2^{kp}$ in parallel will cut the original tour. Otherwise, these two 2-exchanges are defined as non-interacted 2-exchanges and can be executed in parallel without cutting the original tour.

Proof. Supposing $\mathscr{N}_2^{kp} \Rightarrow (e_k, e_p)$ and $\mathscr{N}_2^{gz} \Rightarrow (e_g, e_z)$ on current oriented tour solution $s = (\pi_k \rightarrow \pi_{(k+1)} \rightarrow \pi_g \rightarrow \pi_{g+1} \rightarrow \pi_p \rightarrow \pi_{p+1} \rightarrow \pi_z \rightarrow \pi_{z+1})$, while there is crossing between the two sub-tour ranges $\pi_{(k+1)} \rightarrow \pi_p$ and $\pi_{(g+1)} \rightarrow \pi_z$. After execution of the first 2-opt move $\mathscr{N}_2^{kp} \Rightarrow (e_k, e_p)$, s becomes $s' = (\pi_k \rightarrow \pi_p \rightarrow \pi_{(g+1)} \rightarrow \pi_g \rightarrow \pi_{(k+1)} \rightarrow \pi_{(p+1)} \rightarrow \pi_z \rightarrow \pi_{z+1})$. After execution of the second 2-opt move $\mathscr{N}_2^{gz}$ on s', s' is divided into two sub-tours $(\pi_k \rightarrow \pi_p \rightarrow \pi_{(g+1)} \rightarrow \pi_{(z+1)} \rightarrow \pi_k)$ and $(\pi_g \rightarrow \pi_{(k+1)} \rightarrow \pi_{p+1} \rightarrow \pi_z \rightarrow \pi_{(g+1)})$. One example is shown in Fig.5.10 (c), s_i is divided into tow sub-tours after executing the two 2-exchanges $\mathscr{N}_2^{1,5} \Rightarrow (e_1, e_5)$ and $\mathscr{N}_2^{3,7} \Rightarrow (e_3, e_7)$ simultaneously. Same result happens to the case where any pair of $k, p, g, z, (k, p, g, z+1) mod N$ are identical, like $p = z$ in Fig.5.10(b).

□



(a) Non-interacted sub-tour ranges.

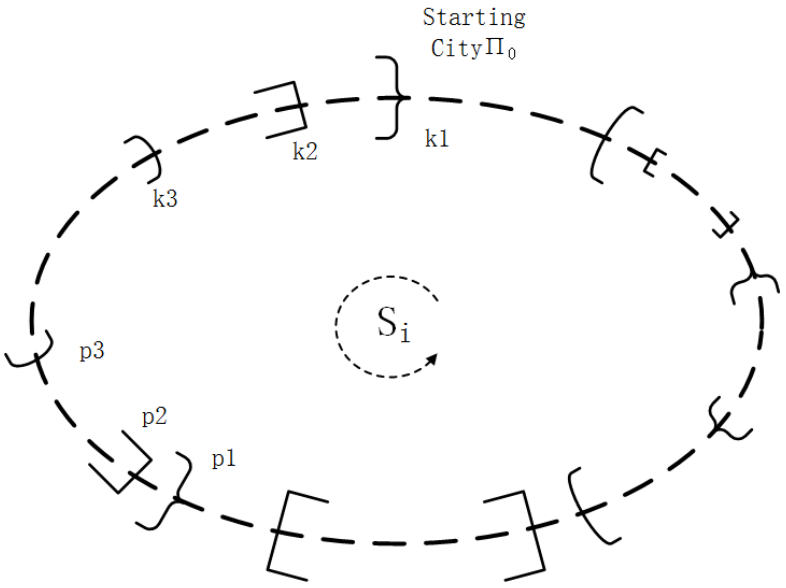
(b) Crossing between two sub-tour ranges.

FIGURE 5.13: Possible distribution of sub-tours of two candidate 2-exchanges (red and black) along the same original tour.

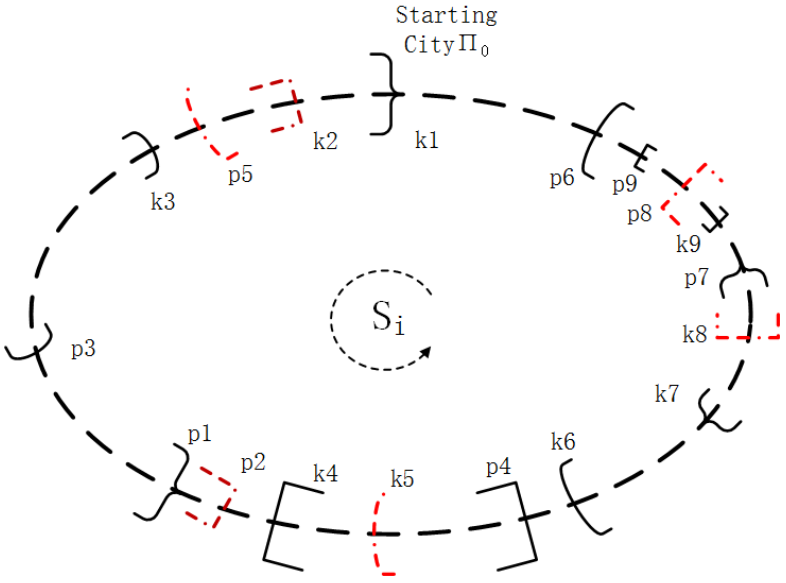
5.5.2.2 Select non-interacted 2-exchanges

We propose to sequentially select non-interacted 2-exchanges since these candidate 2-opt exchanges as well as their related sub-tour ranges are randomly distributed along the

original tour as shown in Fig.5.14, while this operation needs communication between massive candidate exchanges.



(a) None of these 2-exchanges interact with each other.



(b) Three 2-exchanges marked by red dash lines interact with others.

FIGURE 5.14: Distributions of candidate 2-exchanges $\mathscr{N}_2^{k_i p_i}$ on the same tour. A pair of corresponding brackets indicates one special sub-tour segment of $\mathscr{N}_2^{k_i p_i}$ between city π_{k_i} to city π_{p_i} along current tour direction.

This sequential selecting algorithm selects non-interacted 2-exchanges while only traverse current tour solution s_i once with $O(N)$ complexity, as shown in Alg.5.5.2. It access each city along the tour orientation same with which these massive candidate exchanges are found. The function **JudgeNonInteracted** in Alg.5.5.2 judges whether a candidate

2-exchange interacts with the previously selected one according to the definition of non-interacted 2-exchange.

Judging non-interacted 2-exchanges takes advantage of two vacant stacks to decrease complexity and only compare the newly met 2-exchange with the last selected 2-exchange stored at top of the Stacks. As show in Fig.5.14, when the algorithm steps through each city along the original tour solution, if the newly met city plays a role of π_p of a selected 2-exchange $\mathcal{N}_2^{kp}$, the algorithm POPs top element π_k from stack A as well as StackB and city π_k is semaphored (bool value) as possessing a selected non-interacted 2-exchange $\mathcal{N}_2^{kp}$. Then the algorithm steps into next city. If the newly met city possesses a 2-exchange $\mathcal{N}_2^{k_{new},p_{new}}$ does not interact with last selected one, city $\pi_{k_{new}}$ is pushed into stack A, city $\pi_{p_{new}}$ is pushed into stack B.

Here, we should amplify three main points about the sequential selecting algorithms like Alg.5.5.2. Firstly, Alg.5.5.2 traverses each city according to the same permutation tour order that generates these candidate 2-opt exchanges; Secondly, there is not concern about that multiple 2-exchanges may share one same edge found at the parallel 2-opt search step, because each city only plays one role under the "first come, first serve" principle in Alg.5.5.2: either beginning city π_k or end city π_p, of a 2-exchange $\mathcal{N}_2^{kp}$; Thirdly, the firstly selected 2-exchange influences the total quantity of non-interacted 2-exchanges along current tour solution s_i.

5.5.2.3 Execute massive 2-exchanges

For the reason that information of each candidate 2-exchange $\mathcal{N}_2^{kp}$ has been memorized to its starting city π_k during the parallel 2-opt searching step explained in section 5.5.1, and all non-interacted 2-exchanges have been marked in the sequential selection step explained in section5.5.2.2, these selected non-interacted 2-exchanges can be executed independently in arbitrary ordering on the doubly linked list data structure. This step takes $O(N)$ complexity on CPU side. Though it can be executed on GPU side, some extra data copy operation is necessary.

There is no concern about the case that city π_{p_1+1} of $\mathcal{N}_2^{k_1p_1}$ plays another role of π_{k_2} for $\mathcal{N}_2^{k_2p_2}$, because the algorithm executes massive k-exchanges on array of doubly linked route shown in Fig.5.6 where only special one of two links of each city will be changed for one k-opt move and permutation order is represented by links.

Algorithm 5.5.2 Sequentially select *non-interacted 2-exchanges along current TSP solution* s_i.

Require: From the tour's starting city π_0, pass each city π_k according to permutation order of s_i. Consider a candidate 2-exchange $\mathcal{N}_2^{kp}, p \neq (k+1), (k, p+1) mod N$, stack A is for city π_k , stack B is for city π_p.

1: **for** each city π_k from the starting city π_0 **do**
2: **if** A.size $>$ 0 && π_k is in Stack B **then** *//(City π_k has been occupied by previously selected 2-exchange.)*
3: Mark A.top possesses a selected non-interacted 2-exchange $\mathcal{N}_2^{kp}$;
4: Pop A.top;
5: Pop B.top;
6: **else if** π_k possesses a $\mathcal{N}_2^{kp}$ **then**
7: Extract π_p from π_k's information package;
8: **if** A.size == 0 && B.size == 0 **then**
9: Push π_k to stack A;
10: Push π_p to stack B;
11: Mark π_p are in stack B;
12: **else** JudgeNonInteracted($\mathcal{N}_2^{kp}$, A.top, B.top) *//(Compare the new $\mathcal{N}_2^{kp}$ with the top 2-exchange in stack.)*
13: **if** Non-interacted **then**
14: Push π_k to stack A;
15: Push π_p to stack B;
16: Mark π_p is in stack B;
17: **end if**
18: **end if**
19: **else**
20: Continue next city $\pi_{(k+1)}$ along s_i; *//(One edge only participate in one 2-exchange.)*
21:
22: **end if**
23: **end for**

5.5.2.4 Extension to massive 3-opt moves

With following four building blocks: high performance GPU parallel local search along the same tour orientation explained in section 5.5.1, doubly linked vertex list explained in section 3.2, array of ordered coordinates-index in section 5.3.2 and the proposed sequential non-interacted 2-exchange set partition algorithm in section 5.5.2.2, the above explanation for executing massive 2-opt moves can be extended to execute 3-opt or k-opt moves within the methodology of "parallel local search but sequential selection" .

For example, the GPU parallel 3-opt local search $\mathcal{N}_3^{kpq}$ should memorize city π_p, π_q to city π_k during GPU parallel local search; the sequential selection algorithm needs three stacks(stack A for π_k, stack B for π_p, stack C for π_q) to judge interference between the special sub-tour segments of two 3-exchanges $\mathcal{N}_3^{kpq}$, the special sub-tour segments of a

3-exchange begin with the starting city and end with the last city of a 3-exchange along current tour orientation, namely $\pi_{k+1} \rightarrow \pi_p$, $\pi_{p+1} \rightarrow \pi_q$.

5.6 Experiments

Massive 2-opt moves with high performance parallel local search explained in section 5.5 is our most valuable and original contribution to parallel $k-$opt GPU implementation. Besides the various proposed methods, we implement two classical sequential iterative 2-opt implementations, Serial-Best-AT shown in Alg.5.6.1 and Serial-First-AT shown in Alg.5.6.2, working on doubly linked list TSP tour data structure shown in Fig.5.5 to avoid tour reversal operation after each exchange.

Three factors affect the final TSP result quality by using one same optimization algorithm. Firstly, the quality of initial TSP solution s_0, in our test we use the given sequence of cities provided in original TSP files. Secondly, parameters of the algorithm. Thirdly, the random factor to decide the sequence of edges being optimized.

We test these different iterative 2-opt implementations on National TSP instances from TSPLIB [Rei91]. **One test** starts from the same initial solution whose permutation order is the same with the order of cities in original input files, ends with a status where no optimization can be found within two iterations. The computing time includes necessary data copy between GPU and CPU. Average values of ten tests using the same algorithm to optimize the same initial TSP solutions are evaluated.

Different matrices are used to evaluate these algorithms. "dis" indicates the length of TSP solution; "t(s)" is the average total time taken in one test, including necessary time for refreshing TSP tour order and time for copying data from GPU to CPU; "%PD" is the percentage deviation compared with the optimum TSP tour solution provided by TSPLIB; "%PDM" is the percentage deviation between the mean solution and the optimum solution; "%PDB" is the percentage deviation between the best solution and the optimum solution; "iters" indicates the average quantity of runs in one test to get the final TSP solution; "N/M" is the experimental local search range for one edge.

These tests are executed on laptop with CPU: Inter(R) Core(TM) i7-4710HQ 2.5GHz, GPU: GeForce GTX 850M with CUDA 8.0.

In this section, we separately present experimental results about parallel 2-opt local optimization with disjoint tour division, massive 2-opt moves methodology and a combinatorial optimization consisting of parallel single SOM and a kind of 2-opt local optimization.

Algorithm 5.6.1 Serial-Best-AT. Sequential iterative 2-opt to find one edge's best 2-opt improvement along the global tour, which works on doubly linked list TSP tour data structure.

Require: Current tour solution $s \in S$;

```
1: optimizeTour ← ∞;
2: while optimizeTour > 0 do
3:     for (Randomly extract each city π_i) do
4:         bestOptimize = 0;
5:         optimize = 0;
6:         Randomly get one link city of π_i as p_2;
7:         while Return back to city π_i do
8:             Traverse the rest edges along current tour to find optimization for edge (π_i, p_2);
9:             if bestOptimize <= optimize then
10:                bestOptimize = optimize;
11:                Register this 2-opt optimization;
12:            end if
13:        end while
14:        if bestOptimize > 0 then
15:            Execute this 2-opt move;
16:            optimizeTour += bestOptimize;
17:        end if
18:    end for
19: end while
```

Algorithm 5.6.2 Serial-First-AT. Sequential iterative 2-opt to find one edge's first 2-opt optimization along the global tour, which works on doubly linked list.

Require: Current tour solution $s \in S$;

```
1: optimizeTour ← ∞;
2: while optimizeTour > 0 do
3:     for Randomly extract each city π_i do
4:         optimize = 0;
5:         Randomly get one link city of π_i as p_2;
6:         while not found or Return back to city π_i do
7:             Traverse the rest edges along current tour to find optimization for edge (π_i, p_2);
8:             if found then
9:                 Execute this 2-exchange;
10:                optimizeTour += optimize;
11:            end if
12:        end while
13:    end for
14: end while
```

5.6.1 Experiments of distributed 2-opt local search

As it has been discussed in section 5.4, drawback of this parallel 2-opt local search with disjoint tour division lies in that it is hard to get a balance between quantity of m edges being optimized simultaneously and the neighborhood length N/M for one edge's optimization. In our tests of other TSP instances, we put emphasis on the quality of final TSP solution, so the length (N/M) is set relatively large and m is set to 2 or 3.

Visual result of one test using this parallel 2-opt local search algorithm with dynamic tour division strategy is shown in Fig. 5.15, from initial TSP solution with total distance 261879 in Fig. 5.15 (a), each inner loop of the algorithm in Fig. 5.9 optimizes two or three edges simultaneously on GPU side with local search range $N/M = 322$; after eight runs, the TSP solution for lu980.tsp reduces to 12460 and can not be optimized further using this method, as shown in Fig. 5.15 (i).

Average results of using this algorithm to optimize the same initial TSP solutions are shown in Table 5.1. From Table 5.1, we can conclude that with appropriate local search range (N/M) for one edge, the parallel 2-opt local search algorithm with dynamic disjoint tour division has the ability to produce similar (see %PDM) or even better (see %PDB) results compared with sequential 2-opt optimization along the integral tour.

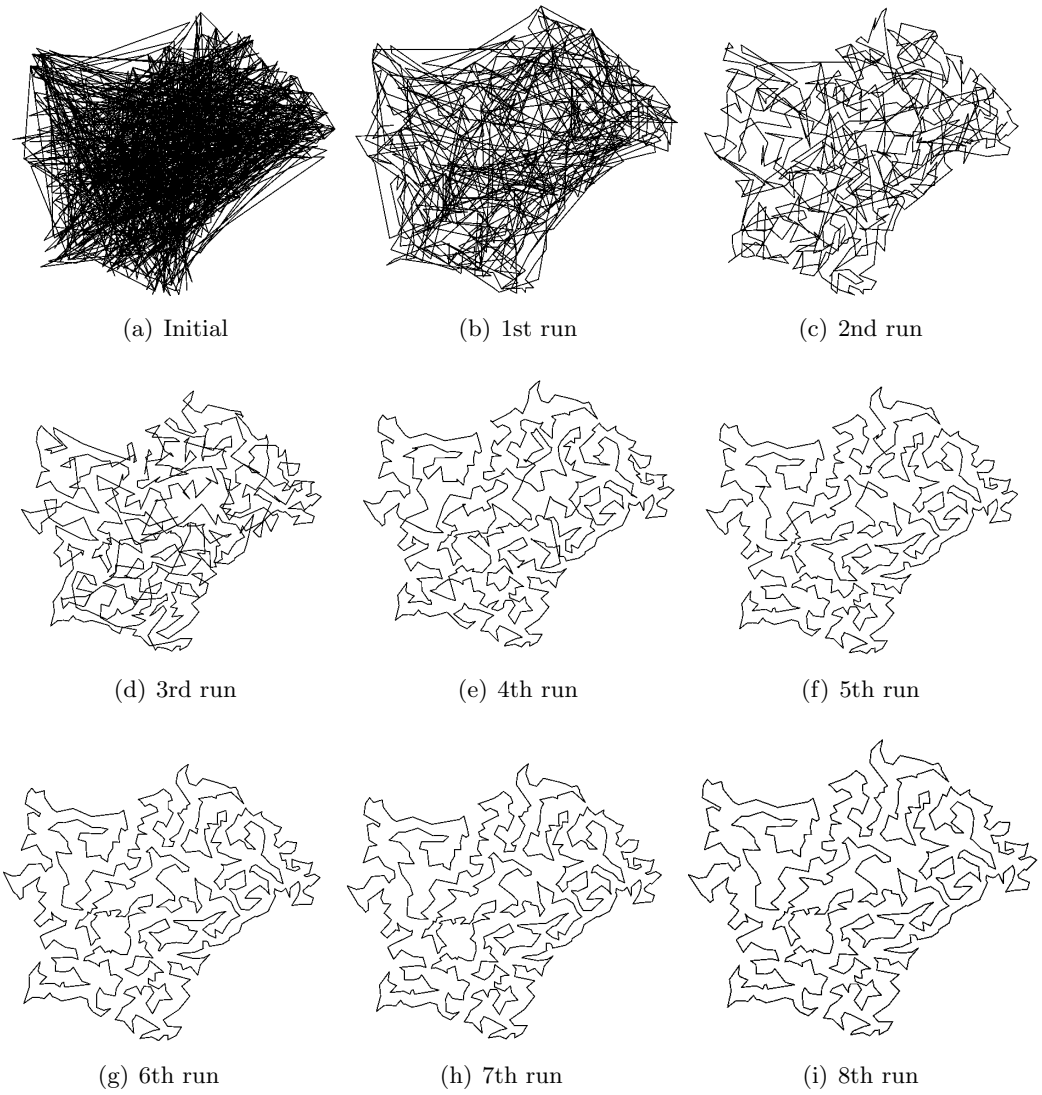
TABLE 5.1: Statistics of Parallel 2-opt Local Search with Dynamic Tour Division Working on GPU

instance	distance	time(s)	% PDM	% PDB	iters	N/M
lu980	12724	3.12	12.20	10.03	7.4	437
rw1621	29407.87	14.01	12.89	10.77	8.3	837
mu1979	97603	14.05	12.33	10.68	8.6	837
nu3496	109624.2	28.08	14.03	12.24	9.3	973
tz6117	455373.8	75.86	15.36	13.21	8.7	2000
eg7146	196279.8	121.46	13.85	12.49	9.6	2347
fi10639	596419.8	207.76	14.57	13.38	8	2837

5.6.2 Experiments of massive 2-opt moves

GPU parallel local search with adaptation about Rocki's parallel 2-opt framework explained in section 5.5.1.1 and with overlapping tour division of $N/2$ length explained in section 5.5.1.2 are two different ways to check same complete $N*(N-1)/2)$ 2-opt checks in one run. We implement the former state-of-art method to test our proposed massive 2-opt moves methodology, namely "parallel local search with sequential selection".

We present comparisons among four 2-opt implementations below, Serial-Best-AT shown in Alg.5.6.1, Serial-First-AT shown in Alg.5.6.2, Rocki's original implementation and our



(a) Initial (b) 1st run (c) 2nd run

(d) 3rd run (e) 4th run (f) 5th run

(g) 6th run (h) 7th run (i) 8th run

FIGURE 5.15: Test results of different runs of parallel local search with dynamic disjoint tour division shown in Alg.5.9: (a) Initial lu980.tsp solution according to original TSP files from TSPLIB [Rei91]; (i) After eight runs of Alg.5.9, the TSP solution can not be further optimized by using this algorithm and reaches to final distance of 12460 in this test.

proposed massive 2-opt moves with Rocki's adaptation explained in section 5.5.1.1, while the two sequential implementations take more than 6 hours per test to approach TSP instances bm33708 and ch71009, we do not show their experimental statistics in the experiments.

The first comparison in Fig.5.16 shows the efficiency of three algorithms on the same two TSP instances, lu980.tsp and sw24978.tsp. We present how the quality of TSP results after each 2-opt run evolves with time in one test. The vertical coordinates show percentage deviation compared with the optimum TSP tour solution provided by

TSPLIB (%PD) after each 2-opt run, the horizontal ordinate shows the time taking totally by previous 2-opt runs before the end of one test.

The second comparison in Fig.5.17 shows TSP results' quality of four algorithms to the same national TSP instances. The TSP results quality is presented by percentage deviation from the optimum of the mean results over 10 tests, namely %PDM. The TSP results' quality of our proposed massive 2-opt moves methodology is slightly degraded.

The third comparison in Fig.5.19, Fig.5.20 and Fig.5.21 show the average running time of one test on these 22 national TSP instances. The running time is hugely accelerated by using our proposed massive 2-opt moves methodology. For the largest "ch71009.tsp", our proposed method only runs 362.54 seconds in one test to optimize an initial brute TSP solution until the first local minimum.

In order to present average quantity of 2-exchanges being executed per run and total quantity of 2-exchanges per test, we present statistic of this "massive 2-opt Moves with Rocki's adaptation" on these 22 national TSP instances in TABLE5.2. Taking the "ch71009" TSP instance in Table 5.2 as an example, average totally 306631 2-exchanges are executed while the proposed methodology only iterates 786 2-opt runs (iterations), since average 392 non-interacted 2-exchanges can be found and executed on the same tour at each iteration. This methodology economizes more than 300000 times of parallel 2-opt runs on GPU side checking the complete $\frac{N(N-1)}{2}$ 2-opt checks as well as more than 300000 times of necessary $O(N)$ sequential tour reversal operations.

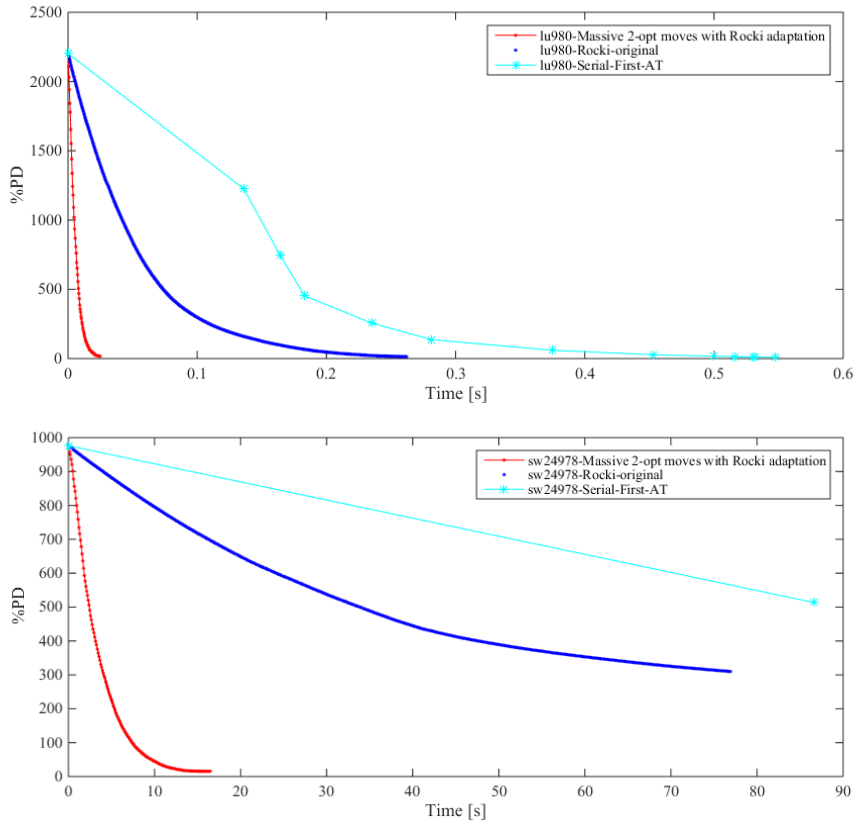


FIGURE 5.16: Comparison of three iterative 2-opt implementations on two TSP instances lu980.tsp and sw24978.tsp separately.

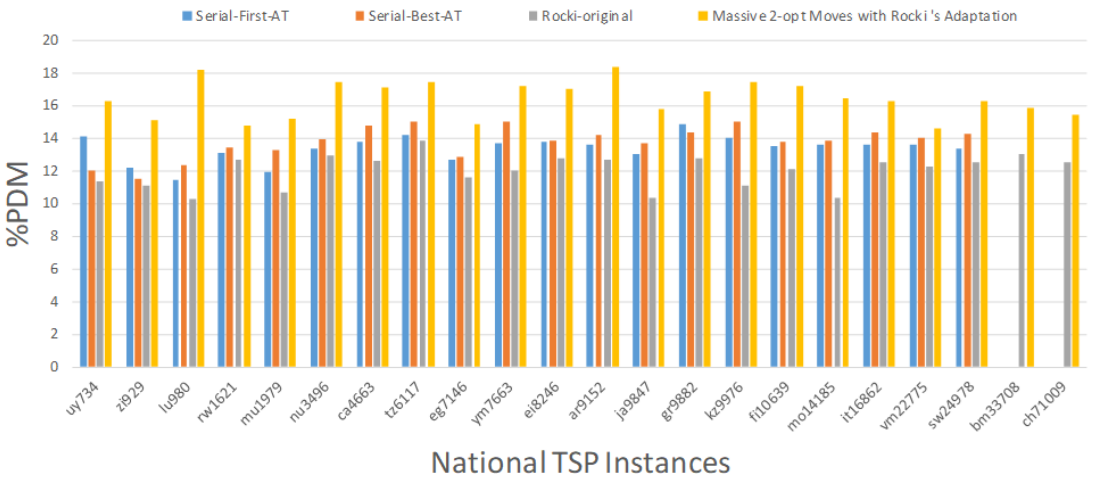


FIGURE 5.17: PDM of TSP results with different iterative 2-opt implementations.

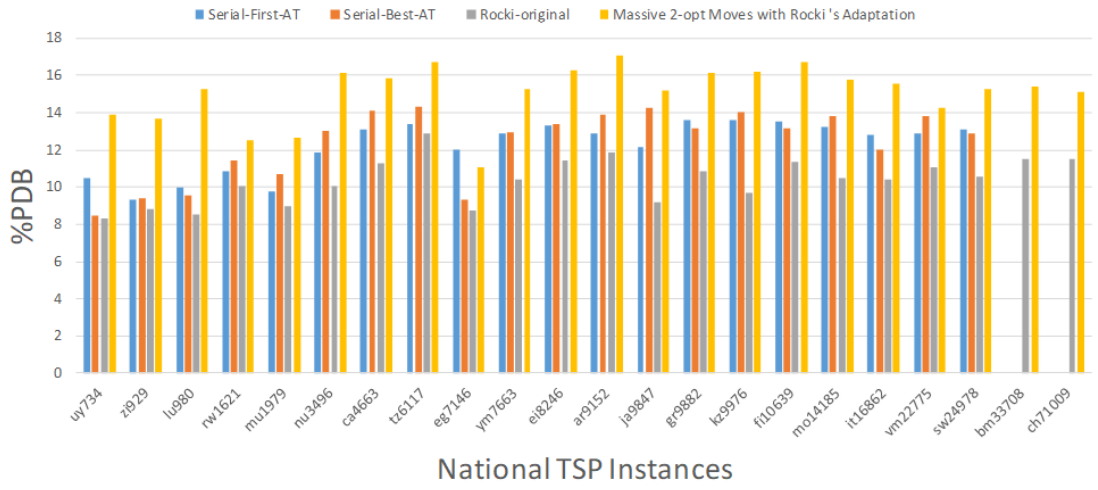


FIGURE 5.18: PDB of TSP results with different iterative 2-opt implementations.

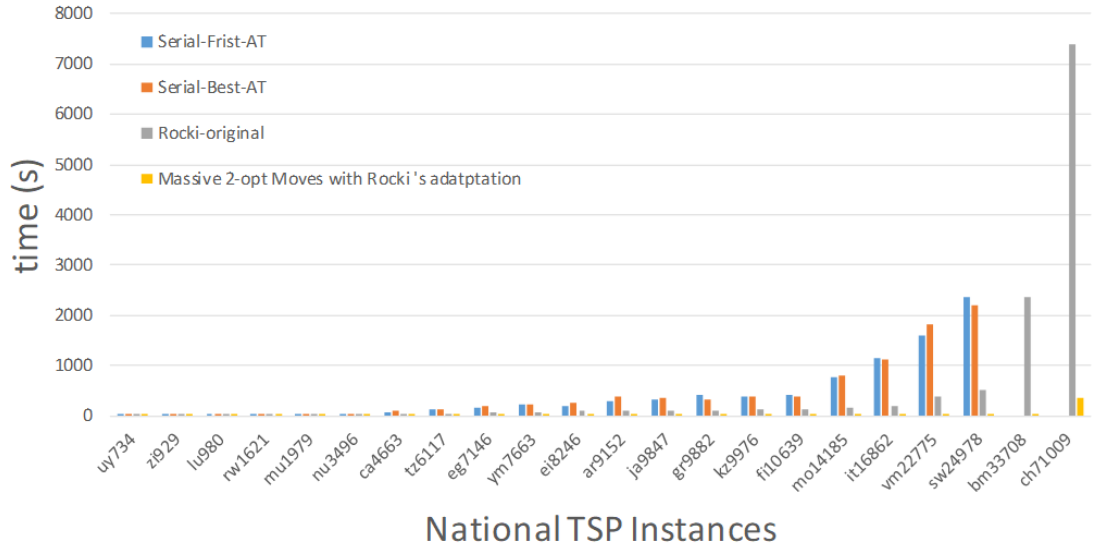


FIGURE 5.19: Average running time of one test using different iterative 2-opt implementations. The sequential algorithms take more than 6 hours per test on personal laptop for TSP instances bm33708 and ch71009.

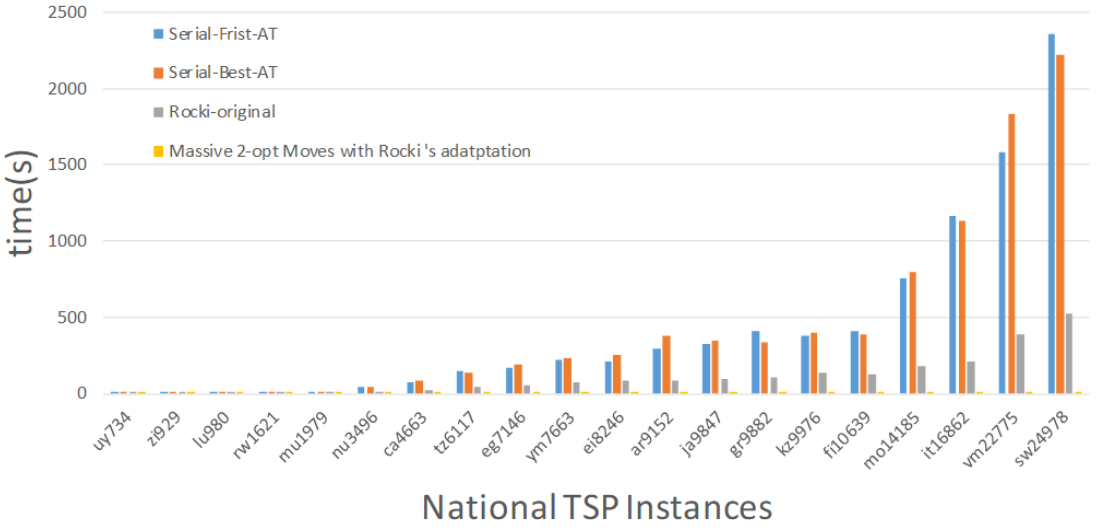


FIGURE 5.20: Zoom-in of Fig.5.19.

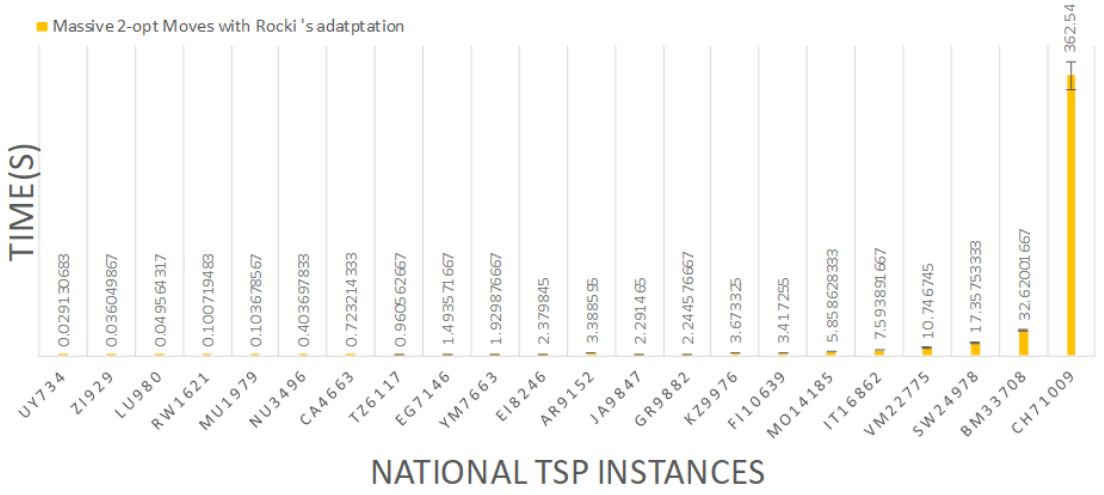


FIGURE 5.21: Single running time statistic of our proposed massive 2-opt moves methodology in Fig.5.19 and Fig.5.20.

TABLE 5.2: Quantity of non-interacted 2-exchanges on the same TSP tour.

BenchMark (TSPLIB)	Total Quantity of 2-exchanges per Test	Iterations per Test	Maximum Quantity of 2-exchanges per Run	Time to Select Non-interacted 2-exchanges per Run(ms)	Time to Refresh Tour per Run (ms)	Time to Execute per Run (ms)	Time GPU H2D per test (ms)	Time GPU D2H per Test (ms)	Time GPU Kernel per Test (ms)
zi929	2606.500	62.000	83.000	0.124	0.078	0.018	6.848	4.070	11.854
lu980	3065.500	77.333	73.333	0.140	0.086	0.019	9.610	5.589	15.451
rw1621	5498.167	96.167	112.833	0.214	0.130	0.025	11.835	6.565	46.765
mu1979	6029.833	78.167	146.167	0.245	0.161	0.033	10.231	5.333	53.719
nu3496	12508.167	130.500	202.500	0.475	0.289	0.045	16.137	8.196	273.854
ca4663	18454.000	145.667	267.333	0.630	0.382	0.061	20.411	10.729	535.812
tz6117	19676.000	126.333	335.167	0.726	0.476	0.070	18.804	9.315	771.705
eg7146	23057.667	148.000	285.167	0.848	0.564	0.076	23.240	11.301	1238.893
ym7663	28052.000	168.500	351.167	0.919	0.585	0.076	25.738	12.317	1625.443
ei8246	32080.500	179.833	375.667	1.026	0.653	0.083	29.077	13.940	2019.640
ar9152	37966.667	208.667	388.167	1.237	0.754	0.090	34.663	16.727	2903.560
ja9847	27827.667	129.833	476.833	1.083	0.730	0.096	22.205	10.271	2010.693
gr9882	23175.500	126.667	470.667	1.104	0.742	0.088	20.825	9.913	1969.257
kz9976	39261.667	196.667	454.500	1.285	0.780	0.095	32.494	15.123	3200.037
fi10639	29691.833	164.667	432.000	1.178	0.790	0.087	28.447	12.877	3037.362
mo14185	38892.833	163.833	524.667	1.585	1.088	0.114	29.352	13.916	5358.477
it16862	45734.833	180.667	647.333	1.880	1.293	0.129	34.745	16.137	6947.097
vm22775	44112.667	146.167	806.167	2.302	1.679	0.156	30.272	12.980	10098.467
sw24978	62462.500	193.500	734.333	2.913	1.951	0.182	43.468	18.150	16320.600
bm33708	73088.000	206.167	866.500	3.776	2.526	0.220	56.512	22.379	31196.967
ch71009	306631.500	786.167	996.333	8.654	6.004	0.373	334.502	118.361	350319.333

5.7 Conclusion

In this chapter, we first discuss both advantages and drawbacks of the two basic TSP tour data structures, array of ordered coordinates and doubly linked list, for parallel GPU $k-$opt implementations. Our first valuable contribution lies in the proposition of a new TSP data structure, the array of ordered coordinates-index, for parallel $k-$opt implementation, in order to unveil how to use GPU on-chip shared memory to achieve the same goal as using doubly linked list and array of ordered coordinates, only the later one is preferred by traditional GPU implementations. This basic data structure can be applied to various applications that needs combinatorial advantages of doubly linked list and array of ordered coordinates.

Our second valuable contribution is a judicious methodology called "parallel local search but sequential selection", which contains a $O(N)$ time complexity sequential algorithm to select massive non-interacted 2-/3-exchanges that are found along the same global tour for many edges. We provide three parallel local search implementations that can profit from the same sequential selection algorithm. The proposed methodology of massive 2-/3-opt moves with high performance GPU local search gets huge speed acceleration while at cost of little quality degradation.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The parallel hierarchical Euclidean minimum spanning forest presented in Chapter 4 forms naturally independent data clusterings from bottom-to-up within divide-conquer framework. Characteristic of this EMSF/EMST algorithm lies in that it combines the local spiral search and Borůvka's algorithm, both of them have natural attributes to be parallelized with technique of data decomposition and decentralized control. This work provides basic component to hierarchical graph minimization applications.

Different TSP data structures in Chapter 5 provides an opportunity to design high performance GPU parallel iterative 2-/3-opt local search algorithms. We test four parallel local search implementations on GPU, and propose a judicious decision making methodology of offloading which part of the k-opt heuristic on GPU while which part remains on CPUs. This "parallel local search but sequential selection" methodology is valuable because we originally propose a $O(N)$ sequential non-interacted 2-/3-exchange set partition algorithm so that massive 2-/3-exchanges that are globally found along the same tour can be executed without interferences. According to our experiment, this newly proposed massive 2-/3-opt moves methodology get the fastest k-opt heuristic so far and similar results quality compared with its sequential version.

6.2 Future Works

Several future works can be extended from the work in this thesis. The cellular partition used in Elias' nearest neighbor search or Bentley's spiral search could be further improved to fit for arbitrary data distribution.

With the parallel Euclidean hierarchical minimum spanning forest, parallel implementations in terms of hierarchical solutions for large scale TSP, stereo matching, optical flow can be envisaged.

Based on the massive 2-/3-opt moves with high performance GPU parallel local search and the parallel EMST implementation, various parallel combinatorial metaheuristics will be proposed to solve large scale optimization problems.

Bibliography

[ABCC11] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[ADJ+98] Micah Adler, Wolfgang Dittrich, Ben Juurlink, Mirosław Kutyłowski, and Ingo Rieping. Communication-optimal parallel minimum spanning tree algorithms. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 27–36. ACM, 1998.

[ADM82] Hassan M. Ahmed, J-M Delosme, and Martin Morf. Highly concurrent computing structures for matrix arithmetic and signal processing. *Computer*, (1):65–82, 1982.

[AESW91] Pankaj K Agarwal, Herbert Edelsbrunner, Otfried Schwarzkopf, and Emo Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete & Computational Geometry*, 6(3):407–422, 1991.

[AXC00] Li An, Qing-San Xiang, and Sofia Chavez. A fast implementation of the minimum spanning tree method for phase unwrapping. *IEEE transactions on medical imaging*, 19(8):805–808, 2000.

[BC04] David A Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 39. IEEE, 2004.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BF75] Jon Louis Bentley and Jerome H Friedman. Fast algorithms for constructing minimal spanning trees in coordinate spaces. *IEEE Trans. Comput.*, 27(STAN-CS-75-529):97, 1975.

[Bor26] Otakar Boruvka. O jistém problému minimálním. 1926.

[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.

[CC96] Sun Chung and Anne Condon. Parallel implementation of bouvka's minimum spanning tree algorithm. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 302–308. IEEE, 1996.

[CCGC07] Guolong Chen, Shuili Chen, Wenzhong Guo, and Huowang Chen. The multi-criteria minimum spanning tree problem based genetic algorithm. *Information Sciences*, 177(22):5050–5063, 2007.

[Cha00] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047, 2000.

[Chr76] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[Cle79] John Gerald Cleary. Analysis of an algorithm for finding nearest neighbors in euclidean space. *ACM Transactions on Mathematical Software (TOMS)*, 5(2):183–192, 1979.

[Cro58] Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.

[DG98] Frank Dehne and Silvia Gotz. Practical parallel algorithms for minimum spanning trees. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 366–371. IEEE, 1998.

[FBF77] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[FJMO95] Michael L Fredman, David S Johnson, Lyle A McGeoch, and Gretchen Ostheimer. Data structures for traveling salesmen. *Journal of Algorithms*, 18(3):432–479, 1995.

[GGT00] Michael Greenspan, Guy Godin, and Jimmy Talbot. Acceleration of binning nearest neighbor methods. *Proceedings of Vision Interface 2000*, pages 337–344, 2000.

[GJ79] Michael R Garey and David S Johnson. A guide to the theory of np-completeness. *WH Freemann, New York*, 70, 1979.

[Glo89] Fred Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[Hel00] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[HJ99] David R Helman and Joseph JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Workshop on Algorithm Engineering and Experimentation*, pages 37–56. Springer, 1999.

[HVN09] Pawan Harish, Vibhav Vineet, and PJ Narayanan. Large graph algorithms for massively multithreaded architectures. *International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74*, 2009.

[JM97] David S Johnson and Lyle A McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1:215–310, 1997.

[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.

[JTP+09] Piotr Juszczak, David MJ Tax, Elżbieta Pe, Robert PW Duin, et al. Minimum spanning tree based one-class classifier. *Neurocomputing*, 72(7):1859–1869, 2009.

[Kar77] Richard M Karp. Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Mathematics of operations research*, 2(3):209–224, 1977.

[Kru56] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[LA11] Bakir Lacevic and Edoardo Amaldi. Ectropy of diversity measures for populations in euclidean space. *Information Sciences*, 181(11):2316–2339, 2011.

[Lap92] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, 1992.

[Lin65] Shen Lin. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44(10):2245–2269, 1965.

[Lin94] Andrzej Lingas. A linear-time construction of the relative neighborhood graph from the delaunay triangulation. *Computational Geometry*, 4(4):199–208, 1994.

[LKC+12] Ke Li, Sam Kwong, Jingjing Cao, Miqing Li, Jinhua Zheng, and Ruimin Shen. Achieving balance between proximity and diversity in multi-objective evolutionary algorithm. *Information Sciences*, 182(1):220–242, 2012.

[LMS03] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[Luo11] TV Luong. Métaheuristiques paralleles sur gpu. *Université des Sciences et Technologie de Lille*, page 2, 2011.

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

[MRG10] William B March, Parikshit Ram, and Alexander G Gray. Fast euclidean minimum spanning tree: algorithm, analysis, and applications. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 603–612. ACM, 2010.

[MW03] Samuel A Mulder and Donald C Wunsch. Million city traveling salesman problem solution by divide and conquer clustering with adaptive resonance neural networks. *Neural Networks*, 16(5):827–832, 2003.

[Nil03] Christian Nilsson. Heuristics for the traveling salesman problem. *Linkoping University*, pages 1–6, 2003.

[NMN01] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete mathematics*, 233(1-3):3–36, 2001.

[Nvi10] CUDA Nvidia. Programming guide, 2010.

[Önc07] Temel Öncan. Design of capacitated minimum spanning tree with uncertain cost and demand parameters. *Information Sciences*, 177(20):4354–4367, 2007.

[PPP04] J-S Park, Michael Penner, and Viktor K Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[Pri57] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957.

[QC17a] Wen-bao Qiao and Jean-charles Créput. Massive parallel self-organizing map and 2-opt on gpu to large scale tsp. In *International Work-Conference on Artificial Neural Networks*, pages 471–482. Springer, 2017.

[QC17b] Wen-Bao Qiao and Jean-Charles Créput. Parallel 2-opt local search on gpu. *World Academy of Science, Engineering and Technology, International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering*, 11(3):281–285, 2017.

[QQ94] Michael J Quinn and Michael Jay Quinn. *Parallel computing: theory and practice*, volume 2. McGraw-Hill New York, 1994.

[Raj04] Sanguthevar Rajasekaran. On the euclidean minimum spanning tree problem. *Computing Letters*, 1(1), 2004.

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

[Riv74] Ronald L Rivest. On the optimality of elia's algorithm for performing best-match searches. In *IFIP Congress*, pages 678–681, 1974.

[ROB+18] Eyder Rios, Luiz Satoru Ochi, Cristina Boeres, Vitor N Coelho, Igor M Coelho, and Ricardo Farias. Exploring parallel multi-gpu local search strategies in a metaheuristic framework. *Journal of Parallel and Distributed Computing*, 111:39–55, 2018.

[RP15] Swaroop Indra Ramaswamy and Rohit Patki. distributed minimum spanning trees, 2015.

[RS94] Gabriel Robins and Jeffrey S Salowe. On the maximum degree of minimum spanning trees. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 250–258. ACM, 1994.

[RS12] Kamil Rocki and Reiji Suda. Accelerating 2-opt and 3-opt local search using gpu in the travelling salesman problem. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 489–495. IEEE, 2012.

[RS13] Kamil Rocki and Reiji Suda. High performance gpu accelerated local optimization in tsp. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1788–1796. IEEE, 2013.

[SH75] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 151–162. IEEE, 1975.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan primitives for gpu computing. In *Graphics hardware*, volume 2007, pages 97–106, 2007.

[SS10] Shyam Sundar and Alok Singh. A swarm intelligence approach to the quadratic minimum spanning tree problem. *Information Sciences*, 180(17):3182–3191, 2010.

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

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[VAS95] MGA Verhoeven, Emile HL Aarts, and PCJ Swinkels. A parallel 2-opt algorithm for the traveling salesman problem. *Future Generation Computer Systems*, 11(2):175–182, 1995.

[VHPN09] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and PJ Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 167–171. ACM, 2009.

[Wan15] Hongjian Wang. *Cellular matrix for parallel k-means and local search to Euclidean grid matching*. PhD thesis, Université de Technologie de Belfort-Montbéliard, 2015.

[WHG11] Wei Wang, Yongzhong Huang, and Shaozhong Guo. Design and implementation of gpu-based prim's algorithm. *International Journal of Modern Education and Computer Science*, 3(4):55, 2011.

[WMC+15] Hongjian Wang, Abdelkhalek Mansouri, Jean-Charles Cr, et al. Massively parallel cellular matrix model for self-organizing map applications. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 584–587. IEEE, 2015.

[WMC17] Hongjian Wang, Abdelkhalek Mansouri, and Jean-Charles Créput. Cellular matrix model for parallel combinatorial optimization algorithms in euclidean plane. *Applied Soft Computing*, 61:642–660, 2017.

[WWW09] Xiaochun Wang, Xiali Wang, and D Mitchell Wilkes. A divide-and-conquer approach for minimum spanning tree-based clustering. *IEEE Transactions on Knowledge and Data Engineering*, 21(7):945–958, 2009.

[WZC13] Hongjian Wang, Naiyu Zhang, and Jean-Charles Créput. A massive parallel cellular gpu implementation of neural network to large scale euclidean tsp. In *Mexican International Conference on Artificial Intelligence*, pages 118–129. Springer, 2013.

[XOX02] Ying Xu, Victor Olman, and Dong Xu. Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, 18(4):536–545, 2002.

[XU97] Ying Xu and Edward C Uberbacher. 2d image segmentation using minimum spanning trees. *Image and Vision Computing*, 15(1):47–57, 1997.

[Yan04] Li Yang. K-edge connected neighborhood graph for geodesic distance estimation and nonlinear data projection. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 1, pages 196–199. IEEE, 2004.

[Zah71] Charles T Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on computers*, 100(1):68–86, 1971.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126, 2008.

[ZMF11] Caiming Zhong, Duoqian Miao, and Pasi Fränti. Minimum spanning tree based split-and-merge: A hierarchical clustering method. *Information Sciences*, 181(16):3397–3410, 2011.

[ZMMF15] Caiming Zhong, Mikko Malinen, Duoqian Miao, and Pasi Fränti. A fast minimum spanning tree algorithm based on k-means. *Information Sciences*, 295:1–17, 2015.

[ZMW10] Caiming Zhong, Duoqian Miao, and Ruizhi Wang. A graph-theoretical clustering method based on two rounds of minimum spanning trees. *Pattern Recognition*, 43(3):752–766, 2010.