RUHR
UNIVERSITÄT
BOCHUM

RUB

PARALLEL DELAUNAY MESH GENERATION USING MULTIPLE GRAPHIC CARD PROCESSORS

vorgelegte

Dissertation

zur

Erlangung des Grades

Doktor-Ingenieur (Dr.-Ing.)

der

Fakultät für Bau- und Umweltingenieurwissenschaften der

Ruhr-Universität Bochum

von

M.Sc.-Ing.Vishnukanthan Kandasamy

Bochum, im März 2014

Tag der Einreichung: 04.März 2014

Tag der mündlichen Prüfung: 22. August 2014

Referenten: Prof. Dr.-Ing. Markus König

Lehrstuhl für Informatik im Bauwesen

Fakultät für Bau- und Umweltingenieurwissenschaften

Ruhr-Universität Bochum

Prof. Dr.-techn. Günther Meschke

Lehrstuhl für Statik und Dynamik

Fakultät für Bau- und Umweltingenieurwissenschaften

Ruhr-Universität Bochum

Acknowledgements

First of all I would like to thank my “Doctor Father” Prof. Dr.-Ing. Markus König with a great appreciation for this valuable opportunity to work with him as a researcher and a teaching assistant. He was always flexible to select my research theme and helped me in many ways to develop my knowledge in my fields. For me, it is a very valuable experience to work with him. I will remember always how he encouraged and helped me in my difficult times to complete this PhD.

I express my gratitude to Prof. Dr.-techn. Günther Meschke, to accept and review my thesis. His valuable feedbacks really helped me to improve my work. I am very proud to say that he was my first FEM teacher. My third referee Prof. Dr.-Ing. habil. M. Wichern has kindly accepted my request to review my work. I am really thankful to him.

I am always thankful to my colleagues; they helped me in several ways to improve my work by giving valuable comments and motivations. Specially apl. Prof. Dr.-Ing. K. R. Leimbach, Dr.-Ing. Christian Koch and Dr.-Ing. Karlheinz Lehner, they gave me lot of advices to improve my research work and documentation. I had a wonderful five years’ time at the institute with lot of colorful memories. I remember with a great pleasure all of my friends and their encouragements to successfully finish my work.

I am glad to thank with lot of loves to my parents and family for their prayers and well wishes. I would love to dedicate my thesis to my parents, family and my lovely wife Manoja, even though she is not travelling with my life anymore, her pure love and four years long sacrifices of her dream married life allowed me to get my doctorate. She made me as a new person and has reserved a permanent place in my heart.

Finally as always I would like to thanks my almighty God “Pillayar”, he guides me and gives me strength along my journey of life.

Vishnukanthan Kandasamy Bochum, March 2014

Abstract

Efficient parallel mesh generation is an active research area to satisfy the computational demand created by large scale, dynamic, and real-time Finite-Element-Method (FEM) based simulation problems. Even though many different CPU-based parallel meshing algorithms and implementations exist, the parallel computing potential of modern graphic processing units (GPUs) has not yet been fully exploited by meshing applications. Parallel computing with general purpose GPU (GPGPU) is widely used in various numerical computing applications and realizes a large performance improvement over traditional CPU-based computing. Application of GPGPUs in computational geometry problems (e.g., triangulation and mesh generation), is a challenging and relatively new field of research. So far, this kind of applications exploit the graphic processors are very rare.

The main objective of this thesis is to present a research concept for parallel Delaunay mesh generation using multiple GPUs. The Delaunay triangulation is a triangulation technique which always produces the best mesh. There are only few approaches of Delaunay triangulation which employ graphic processors for parallel computing. Currently there is no parallel FE mesh generation application using multiple GPUs. This thesis is focused on developing a 2D Delaunay mesh generator to be computed on a multiple GPUs computing system by incorporating the existing GPU-based Delaunay triangulation (GPU-DT), which only supports for computation on single GPU system.

For parallel computation, the CUDA computing architecture is used in this thesis. Parallelization is focused to upgrade a part of the triangulation algorithm to support on a multiple GPUs system, namely the construction of a digital Voronoi diagram, which is the core algorithm to build the Delaunay triangulation. For this mesh generator, suitable partitioning technique, construction of interface-mesh between subdomains, and merging of sub-meshes are implemented.

Apart from facilitating the portability for multiple GPUs, the GPU-DT implementation is also extended as a Delaunay mesh generator by incorporating the internal and external non-convex boundaries for any arbitrary mesh domain. In this implementation, a priori mesh partitioning method based on a coordinate-bisecting technique is incorporated. The interfaces between sub domains are constructed beforehand in a pre-process step using an incremental Delaunay construction technique. The developed parallel mesh generation concept to support on multiple GPUs computing system is illustrated and verified with meshing examples.

Zusammenfassung

Die effiziente parallele Netzgenerierung ist ein aktives Forschungsfeld. Durch die Analyse von dynamischen oder Echtzeit-basierten numerischen Problemen mittels der Finite-Element-Methode (FE-Methode) müssen umfangreiche Berechnungsnetze immer schneller erzeugt werden. Auch wenn es heute bereits sehr viele unterschiedliche CPU-basierte parallele Vernetzungsalgorithmen und Implementierungen gibt, ist das Potenzial moderner Hardware noch lange nicht ausgeschöpft. Insbesondere sehr leistungsfähige Grafikprozessoren (GPUs) ermöglichten in den letzten Jahren eine sehr große Leistungssteigerung und können auch für die Netzgenerierung eingesetzt werden. Allgemeine GPUs (so-genannte GPGPUs) werden für verschiedene Ingenieurberechnungen bereits eingesetzt und bewirken eine große Steigerung der Leistungsfähigkeit gegenüber des Parallel-Computing mit herkömmlichen CPUs. Die Anwendung von GPGPUs für Geometrieberechnungen, wie etwa die Triangulation oder die Netzgenerierung, ist jedoch noch ein sehr neues Forschungsfeld. Bisher gibt es nur sehr wenige Algorithmen, welche die Potentiale der Grafikprozessoren ausnutzen.

Das Hauptziel dieser Arbeit ist es, ein Forschungskonzept zu erarbeiten, um die parallele Netzgenerierung auf mehreren Grafikprozessoren ausführen zu können. Hierzu wird die Delaunay-Triangulation verwendet, welche eine sehr verbreitete Triangulations-Technik ist, um qualitativ sehr gute Netze generieren zu können. Es gibt allerdings nur wenige Ansätze, die die Delaunay-Triangulation auf Grafik-Prozessoren implementieren. Aktuell existieren keine Anwendungen zur effizienten Generierung von großen FE-Netzen unter Verwendung von mehreren Grafikprozessoren. Erst durch die parallele Verwendung von mehreren Grafikkarten kann jedoch das volle Potential dieser Hardware ausgenutzt werden. Aufbauend auf einer bereits vorhandenen GPU-basierten Delaunay-Triangulations-Methode (GPU-DT) für einfache GPU-Systeme wurde ein 2D-Delaunay Netzgenerator für mehrere GPGPUs umgesetzt. Als Basis dient dazu die CUDA Computer-Architektur der Firma Nvidia. Der Fokus der Parallelisierung liegt dabei auf der parallelen Erzeugung eines diskreten Voronoi-Diagramms, welches die Grundlage der Delaunay-Triangulation bildet. Hierzu mussten Konzepte zur Zerlegung der Netzpunkte und Zusammenführung der Teilnetze unter Verwendung von Interface-Elementen entwickelt werden.

Abgesehen von der Einsatzfähigkeit auf mehreren GPUs, wird die GPU-DT Implementierung auch als ein Delaunay Netzgenerator weiterentwickelt, um innere und äußere, nicht-konvexen Grenzen in beliebigen Netzbereichen erfassen zu können. In dieser Implementierung wird eine a-priori Partitionierungstechnik mit einbezogen, die auf einem Koordinaten-Bisektionsverfahren basiert. Die Schnittstellen zwischen Subdomains werden vorher in einem Pre-Prozessorschritt mit einer inkrementellen Delaunay Konstruktion errichtet. Das entwickelte Parallelisierungskonzept wird anhand von Beispielen verifiziert und validiert.

Contents

Chapter 1

1 Introduction

1.1 Motivation and Problem Statement

Numerical simulations used in finite element analyses (FEA) are becoming continuously more complex. They are used in the simulations of product designs or in the simulations of real-time processes. The main reasons for the dramatic increase of simulation-size are the increasing complexity of the products and the expected precision in the solutions. To produce more reliable and accurate solutions, models are further refined to reach high resolution results. This kind of nature in current FE-simulations always increases the size of the simulations. Faster and efficient mesh generation techniques are important to analyze large-scale, dynamic, and real-time FE problems. To satisfy the computational requirements for an efficient mesh generation, high performance computing power is required. The continuous development in computer technology and supporting parallel computation techniques already allow large scale FE-simulations.

Even though various CPU-based parallel meshing algorithms exist, the parallel computing potential of modern graphic processing units (GPUs) has not yet been fully exploited by meshing applications. General purpose GPU-based parallel computing (GPGPU) is widely used in diverse numerical computing applications and it experiences a high performance improvement over the traditional CPU-based parallel computing techniques. The computing systems accelerated by graphic hardware are cheap and energy efficient compared to CPU-based systems. General aspects of efficient GPU parallel computation are the availability of huge numbers of light weight parallel threads, reduced communication requirements between the CPU and GPU, and no internal synchronization requirements between parallel threads. Unfortunately, computational geometry or triangulation algorithms contain parallel threads, which are heavy-weight, involve lots of synchronization and are highly dependent on the communication between the CPU and GPU. This communication and synchronization overheads affect the parallel performance of these algorithms.

The Delaunay triangulation method always produces relatively best initial meshes to solve partial differential equations problems. This is a mostly applied triangulation technique in FE mesh generations. One of the efficient GPU-based parallel Delaunay triangulation

implementation is GPU-DT [QI ET AL., 2013]. However, this implementation only supports for parallel computation using single GPU. The application of multiple GPUs in computational geometry problems (e.g., triangulation and mesh generation) is a challenging problem and researches in this area are relatively new. Finding new GPU-based parallel mesh generation algorithms is highly motivated among the research community.

The defined problem statement of this thesis is to develop a parallel mesh generator for FE meshing applications based on Delaunay triangulation method. This mesh generator is expected to be computed in parallel on multiple GPUs computing system. For the Delaunay triangulation, the existing GPU-DT implementation is expected to be further upgraded to support on multiple GPUs environment. To compute the mesh generator on a parallel computing system, necessary techniques for mesh-partitioning, meshing of interface between subdomains and merging of subdomain-meshes need to be implemented. The CUDA programming environment is selected to develop this parallel mesh generator.

1.2 State of the Art

1.2.1 Developments in Finite Element Analysis (FEA)

The era of FEM in numerical analysis obviously starts with the introduction of computing technology. In 1943, for the first time a solution for finite element analysis problem was developed by R. Courant. He introduced a numerical analysis technique to find an approximate solution for a vibration problem [WIDAS, 2013]. In the history of FEM the paper published by R.W. Clough and his group in 1956 had demonstrated the first reported FEA solution [CLOUGH, 2004]. The development of FEM related software and improved analysis techniques were gradually developed along with the computer technology development. FEM related computer programs or software is more widely accessible to anyone with the competitive availability of powerful computers. Before the early 70's, FEA was restricted to main frame computers which were generally available only to large scale industries related with aeronautics, automobile, defense and nuclear research [WIDAS, 2013]. The competitive development in microprocessor technology influenced the production of powerful personal computers and the rapid decrease in computer prices. This rapid development in computer technology not only made the FEA solutions faster, but also allowed more precise results.

Generally an FE problem analysis can be divided into three main steps. As a first step, the problem is defined with respect to its geometry, material conditions, loading and boundary conditions. This process is often called "pre-process" in FEA. In pre-process the solution

domain is discretized into a number of small regions, called finite elements. This discretization of the geometry using nodes and elements is called mesh or grid generation. This mesh data contains the material and structural properties of the system.

The second step in FEA deals with solving of FE system which is represented by a system of equations. These linear equations are solved for approximate solutions. In a general FEA, solving a system of linear equation is computationally intensive and consumes a large part of time. The accuracy of the results depends on the selection of a suitable equation solver and the computing capacity of the computer. According to the size of the problem, the required solving time can vary from a few milliseconds up to several days. This time cost has always been the motivation in the development of faster equation solvers and better computation techniques in the numerical computing world. In the continuous research and development of finite element techniques, a very important part is developing faster equation solvers and pre-conditioners which enable the solutions to be more precise and calculated faster. There have been various sophisticated linear algebra library developments over the past few decades contributing to the FEA world [WIKIC, 2013].

The third step in FEA is called “post-process”. In this step the analyzed results are visualized and studied. These results are analyzed using engineering sense and where required the complete problem or part of the problem is reanalyzed to get satisfactory results. In dynamic FEM analyses, post-process results are calculated in each time step and values are visualized as continuous animation. There are many freely and commercially available tools and packages for FEM post-processing [PARAVIEW, 2013].

1.2.2 Continuously Increasing Numerical Complexity

Engineering design becomes increasingly complex over the time. As shown in a comparison [COTTRELL ET. AL, 2009] in figure 1.1, few of the engineering products are compared according to their production complexity. As an example, an engineering design of an automobile deals with around 3000 different parts. The sizing of a Boeing 777 has over 100,000 parts and modern nuclear submarine consist of over a million parts.

The approximate solution mainly depends on how fine the problem is discretized. To get a better solution finer meshes are required, at the same time this increases the size of the numerical problem which become larger and larger. Nowadays a finite element problem simply handles an order of a few billions of unknowns [HEISTER ET AL., 2010]. Currently available computer technology motivates and enables this type of large scale problems.

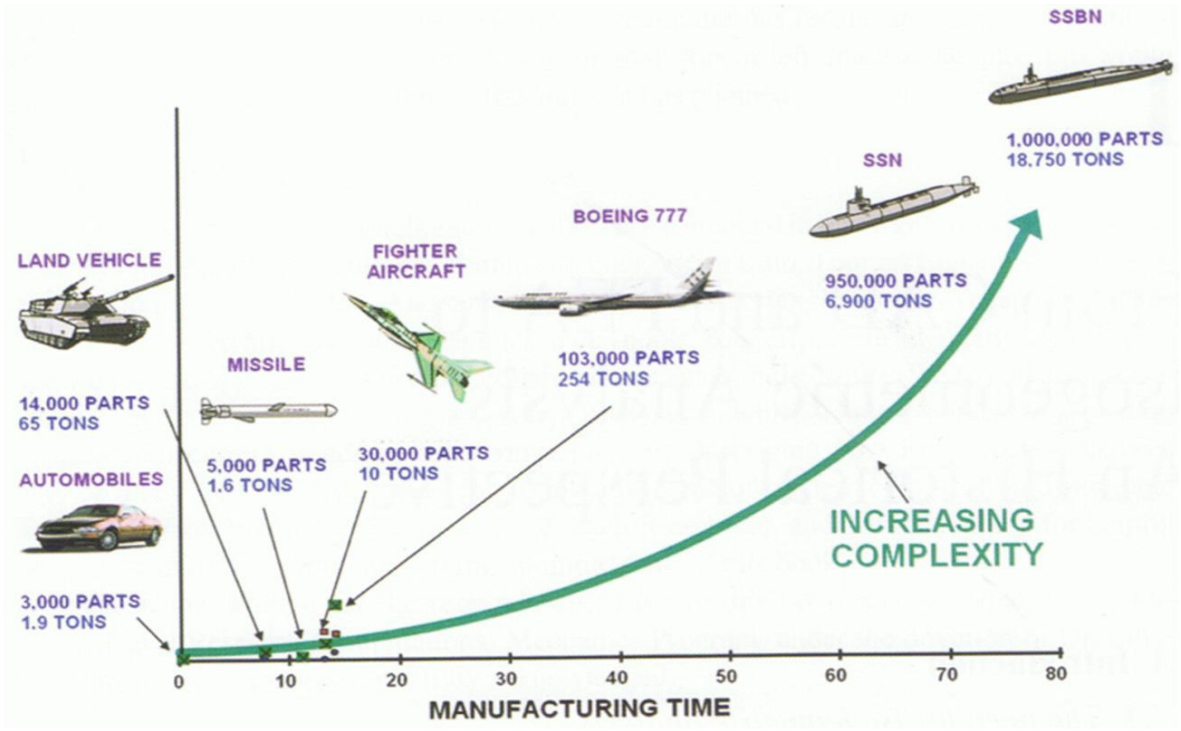

Figure 1-1 Increasing complexity in numerical modeling and FEA of different products [COTTRELL ET AL., 2009]

1.2.3 Increasing Computational Demand for Mesh Generation

Mesh generation requires knowledge of different disciplines such as geometry, computation and numerical requirements. “Meshing things are increasingly recognized as a subject of interest in its own right, not only in engineering but also at universities as well” [FREY AND GERGE, 2008]. The research aspects surrounding the subject of meshing are enormous. Some of the main aspects are optimized generic geometric algorithms, efficient data structure usage, faster computational algorithms and numerical prerequisites.

Triangulation is a geometric representation of given point-set using set of simplexes with certain properties. These geometric simplexes are triangles in two dimensions and tetrahedrons in three dimensions. In a triangulation, if certain edges are defined as constraint edges, then this triangulation falls under the category of constrained triangulation.

In the course of time, there have been many triangulation techniques used in FE mesh generation processes, such as the Delaunay technique, advancing front technique and quadtree or octree technique. Among these, the Delaunay triangulation technique is mostly preferred in meshing applications. The general observation of best possible geometry discritization of a given point cloud is experienced by the Delaunay technique [GEORGE AND BOROUCHAKI, 1998].

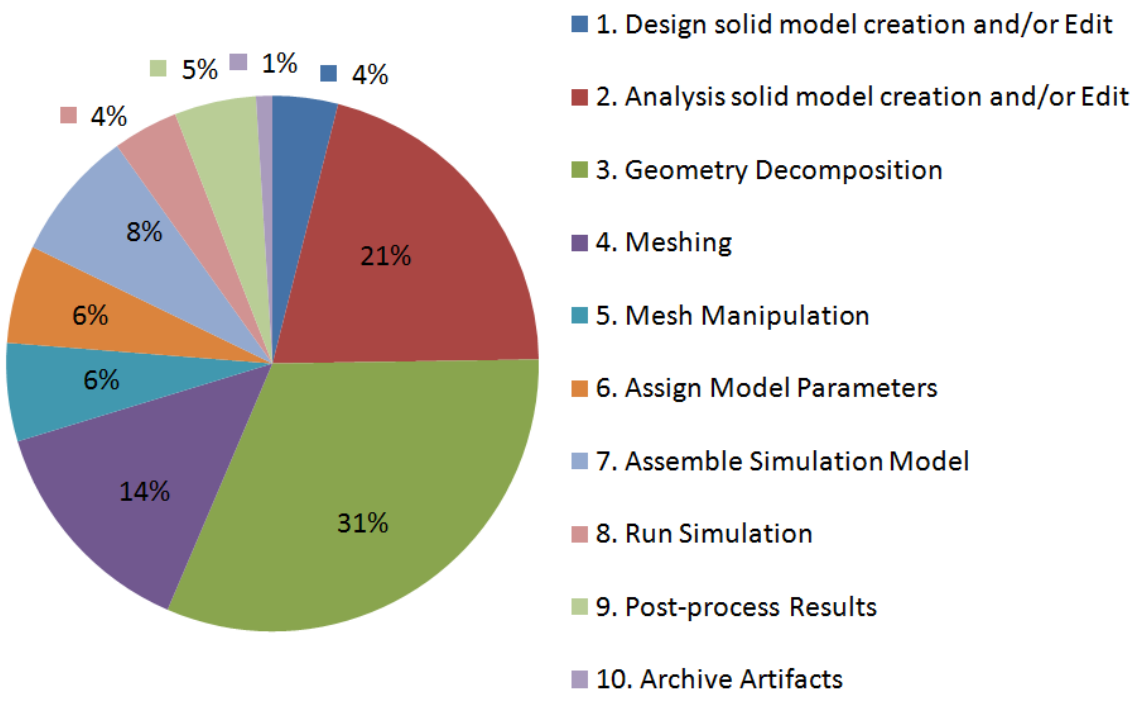

Figure 1-2 Finite element simulation life cycle [COTTRELL ET AL., 2009]

After the wide application of parallel computing in FEA, triangulation and mesh generation were also attracted by the parallel computing technique. The importance of parallel computing in mesh generation for FEA is realized when FE models become larger and larger. In different kinds of FE analyses, such as dynamic FEA, there is a greater requirement of mesh computation. These requirements lead the engineers to search for parallel solution in mesh generation. Parallel mesh generation is a process, which divides the original mesh problem into many sub problems, then performing meshing in parallel using different computing processors. The domain decomposition technique is used to divide the task into almost equal sized sub tasks.

From the long experience of the Sandia laboratory in FE simulation, their reports evidence the importance of the mesh generation in a FE simulation life cycle [COTTRELL ET AL., 2009]. As shown in the figure 1.2, mesh generation consumes in average around 20% of the overall analysis time. As another important part, around 60% of the lifecycle times are generally spent for suitable geometry selection for analysis. In other words, around 20% of the overall time is consumed by problem analysis and the rest of the time (80%) is consumed by modeling, according to common industrial FE simulation experience.

Due to the increase of the size of the numerical simulation and supporting high performance computing resources, mesh generation has become an area of greater concern in recent decades. The idea for parallel computation in mesh generation is attracted by the computational geometry society to develop parallel geometry algorithms.

1.2.4 Development of High Performance Computing (HPC)

After the introduction of computer technology around 1960s the requirement of supercomputers was realized by scientific community. The gradual increased importance of numerical computations for various numerical simulations was a significant motivation in the development of HPC. According to the historical development of this computer system, earliest supercomputers in 1970s were limited with only a few processors. In 1980s this number was increased to thousands of processors and around end of 20th century massively parallel super computer systems were introduced with tens of thousands processors [WIKIB, 2013]. Currently, the world's fastest super computers are equipped with tens of thousands of multi-core CPUs and GPUs processors. Performance of a processor or computing system is generally measured on the basis of the number of operations it can execute per second, in other words, the number of floating point operations per second (FLOPS).

In the technology development of high performance computing history, important properties changed significantly over time. Earlier HPC system were limited in performance but currently available systems reach to more than 20 petaFLOPS (to date world's fastest super computer system, Tianhe-2, theoretical computing performance is around 33 petaFLOPS). The introduction of high performance, energy efficient and multiple-core processors chips made the overall cost of HPC systems very low. The energy efficiency of the current system makes the maintenance and running of such computing systems cheaper [TOP, 2013].

In the research and development of more efficient computer performance, numerical computation using graphic cards (GPUs) is a very recent approach. Modern GPUs are developed in such a way that they can be used for numerical calculations in parallel at the scale of many hundreds processors. There are advanced application programming interfaces (API) like CUDA and openCL available for various modern programming languages. CUDA is an extended version of the C language and especially supported by graphic cards developed by NVIDIA. OpenCL is a similar API which is capable of operating on various graphics cards from different vendors. The application fields use the computing power of GPUs are enormous, including linear algebra, image processing, computer vision, medical imaging,

data-mining, computational mechanics and others. A detailed study of GPU computing and the CUDA architecture is described in section 3.2.

1.3 Structure of this thesis

Three main areas including computational geometry, the finite element method and parallel computing, are partially addressed with this thesis. A Delaunay triangulation problem falls under the computational geometry area. Since the mesh generation is focused for FE application, FEM is briefly introduced in this work. The third field is parallel computing, in which especially the modern parallel computing technique using GPUs is used in this thesis.

Chapter two documents the state of the art of parallel Delaunay triangulation algorithms and their implantations. This related work is discussed in two sections with respect to CPU and GPU parallel computing approaches. In chapter three, is dedicated to give some elementary information of parallel computing. Detail of traditional parallel computing and GPU-based parallel computing are briefly discussed in this chapter.

Chapter four covers some computational geometry fundamentals in three sub-sections, in which Delaunay triangulation basic details, a GPU-DT algorithm overview and domain decomposition techniques particularly related with mesh generation are discussed. In chapter five the multi GPU-DT algorithm concept is presented. This research concept is explained with the help of a meshing problem. Chapter six give the algorithmic and implementation details of the presented multi GPU-DT algorithm. In chapter seven, different case studies are presented to explain the algorithm and parallel implementation using the CUDA parallel computing technique and available preliminary results. Chapter eight discusses the conclusion and outlook related with this work. In chapter nine the references used in this document are listed.

Chapter 2

2 Related Work

In this chapter, two main sections are included to describe the existing related works. In section 2.1, some of the existing researches and implementations in the field of parallel Delaunay triangulation and mesh generation are discussed. This section mainly focuses on the applications of the traditional CPU-based parallel computing architecture in this field. The section 2.2 is dedicated to the modern GPU-based parallel computing application in the field of Delaunay triangulation and mesh generation.

2.1 CPU-Based Parallel Delaunay Triangulation

Generally in Delaunay triangulation technique, it is difficult to divide the problem into independent sub problems and then merge them into the whole problem as a complete Delaunay triangulation. The implementation of a parallel algorithm for Delaunay triangulation is very complicated compared to its serial version [HARDWICK, 1997]. In the application of Delaunay triangulation, it is expected that the algorithm should be stable and easy to implement. In the design of parallel algorithms the scalability of the algorithm is an important consideration. To cope with the overwhelming demand of large data size and fast computation for current numerical simulations, it is motivated to determine parallel algorithm versions of their existing serial counterparts. Continuous development in computer hardware technology also allows finding completely new parallel triangulation algorithms [KOHOUT AND KOLINEROVA, 2003].

Parallel mesh generation and its application is still an active research area. One of the major application areas includes computationally intensive finite element and finite volume methods, particularly in CFD [KOLINEROVA AND KOHOUT, 2002]. A detailed survey of unstructured parallel mesh generation methods is presented by Chrisochoides [CHRISOCHOIDES, 2006]. His survey covers the existing research details of parallel mesh generations up to the year 2005. That survey is focused only on Delaunay, advancing front and edge-subdivision triangulation methods in unstructured mesh generations. Since the objective of this research is limited to a parallel Delaunay unstructured mesh, Chrisochoides's research survey suffice the requirement until 2005. This section summarizes his part of the survey results which relate to Delaunay triangulation. In addition to that, reviews of the latest

research approaches after the year 2005 in the field of unstructured parallel mesh generation are discussed.

As mentioned by Chrisochoides, the challenges in parallel mesh generation are managed to make the mesh generator satisfy properties such as stability, code re-usability and scalability. The stability in the aspect of parallel mesh generation is understood by producing a quality of finite elements like state of the art sequential generators. The hundred percentage of code re-use is the repetitive usage of mesh generators without substantial deterioration of scalability [CHRISOCHOIDES, 2006]. During the parallel processing of meshing on different processors, according to the data communication and synchronization, requirements between adjacent sub-problems are classified as tightly coupled, de-coupled or partially coupled mesh problems. According to Chrisochoides's survey, he categorizes the sequential triangulation technique into Delaunay, advancing front, edge sub division and the interface between sub-problems divided into tightly coupled, decoupled and partially coupled. Another detailed survey about the 2D parallel Delaunay triangulation using shared memory computer architecture can be found in [KOHOUT ET AL., 2004].

Table 2.1 Some parallel Delaunay triangulation algorithms and implementations using CPU based parallel architecture

Delaunay Algorithm	CPU parallel computing technique		
	Shared Memory -openMP	**Distributed Memory -MPI**	**Combined**
Incremental insertion BW algorithm	Vicente et al., 2009	Chew et a., 1997	Chrisochodes et al., 2009
	Kolinerova et al., 2004	Okusanya and Peraire, 1996	
	Blandford et al., 2006		
Incremental Construction	Kohout and Kolinerova, 2003	Beichl and Sullivan, 1991	
		Cignoni et al., 1993	
		Lee, 1997	
Divide & Conquer	Blelloch et al., 1999	Davy and Dew, 1989	Hardwick, 1997
		Cignoni et al., 1993, 1994	
		Hardwick, 1997	
		Chen et al., 2001, 2001, 2006	
Edge-flipping / Edge-subdivision / Refinement		Okusanya and Peraire, 1996, 97	
		Spielman et al., 2002	
		Nave et al., 2002	

The literature survey is mainly concentrated with parallel algorithms and implementations related to Delaunay triangulation and unstructured mesh generation. The existing parallel Delaunay triangulations and mesh generations are grouped into different categories with respect to triangulation techniques and parallel computing techniques used. With the ever evolving computing techniques there are various parallel techniques used in the finite element method and the parallel mesh generation process. Out of this historical development of

parallel computing techniques the two well-known techniques are distributed memory computing (also called message passing interface or MPI) and shared memory computing (called open multi-processing or openMP). In the application of these techniques in Delaunay mesh generation, partitioning the domain is called the domain decomposition technique. It plays a major role in parallelizing the meshing process on different processors. In Delaunay triangulation the major techniques such as incremental insertion, incremental construction, divide & conquer, edge flipping and sweeping plane algorithms are considered. In the following section, some of the existing parallel Delaunay algorithms and implementations are briefly discussed. As presented in Table 2.1, it maps the each work according to the used parallel computing technique and the type of the Delaunay triangulation technique.

Even though there is a general concept that Delaunay triangulation is difficult to parallelize, the divide and conquer (D&C) technique is relatively easily parallelizable. There are many parallel Delaunay algorithms and implementations based on the D&C technique [CHEN ET AL., 2006, CIGNONI, 1994, CIGNONI ET AL., 1993]. The D&C algorithm is based on recursive partitioning of original problem until the number of point size in each partition reaches a defined smallest threshold. In each partition, triangulation is performed in parallel using one of the Delaunay triangulation techniques. The resulting triangulation in each partition is joined during the merging phase of each level of partitions until the complete triangulation is obtained.

The D&C algorithm seems simple to parallelize, hence the merging of triangulation sub problems influences the parallel performance. When the problem is divided into very small parts, requirement of merging process increases. There are different algorithms used for the merging process. The incremental construction is one of the technique [LEE AND SCHALTER, 1980] used in merging. This merging technique compromises the parallel performance of D&C technique. One of the drawbacks in this technique is that the merging part cannot be performed in parallel. It is computed by only one processing element [KOHOUT ET AL., 2004].

One of the earliest approaches to parallelize the Delaunay triangulation, proposed by [DAVY AND DEW, 1989], is based on the D&C technique. The presented approach in their paper is a two dimensional domain split into a number of equal sized subdomains where all the point-sets contained in every subdomain are processed on different processing elements (processors). In each processor, local Delaunay triangulation is built using incremental insertion algorithm. The triangulation subsets are merged pair-wise using a binary tree by rising up to complete the whole problem [CIGNONI ET AL., 1993]. According to Cigononi's

paper the algorithm was implemented on eight processing nodes and realized a speed up from 4.09 to 5.7.

As discussed in [CIGNONI ET AL., 1993], one of the first implementation of parallel Delaunay triangulation by [BEICHL AND SULLIVAN, 1991] shows not an impressive speed up. The triangulation approach was an incremental construction algorithm on a data parallel architecture. According to the reported results, the parallel algorithm was only twice as faster as the sequential version for 1000 points on a machine with thousands of processors.

The Dewall algorithm from [CIGNONI, 1994, CIGNONI ET AL., 1993] is one of the best implemented, often cited parallel Delaunay triangulation algorithms. This is a merge-first-then-triangulate type of parallel approach for D&C algorithm. In this work, they have used an incremental construction method in both merging and triangulating the points of individual partitions. The main difference of this algorithm compared to the original D&C algorithm is that merging is performed in the first phase before the triangulation starts. The triangles on merging region are the simplexes which intersect the partition plane. The interface region (merge region) triangles are passed as a parameter to a triangulation routine to complete both side triangulations. The parallel version of Dewall has no significant difference from the serial version. Since the merging phase is excluded in this algorithm, there is only one synchronization point in the partitioning phase. After that, each processing element independently constructs their part of triangles. The reported result of this group shows a speed up range of 1.70 - 3.35 for 2-16 processing elements for a uniformly distributed point-set with 8000 points [CIGNONI ET AL., 1993].

From the same publication of [CIGNONI ET AL., 1993], one more parallel solution for Delaunay triangulation is presented. This is called ParInCoDe for “Parallel Incremental Construction Delaunay algorithm”. The details of the serial version of the Incremental Construction Delaunay (InCoDe) algorithm are published in [CIGNONI, 1994]. In their original algorithm, the Delaunay triangulation of a given point-set is incrementally built by adding a new simplex in each step, without modifying the existing triangulation. Finding an initial triangle simplex in this algorithm is proposed by the authors by either randomly selecting one point, then finding one closest point after that the third point is decided to satisfy the Delaunay property. Implementation of this algorithm is easy; however the time complexity of this algorithm significantly compromises the performance. A few speeding techniques (3D gridding and face list management via hash tables) are proposed and implemented by the same authors in their optimized version of InCoDe algorithm.

The parallel version of InCoDe is employed by partitioning a bounding box of a given point-set into rectangular boxes, then the whole point-set is passed to each box to independently process the sub triangulation on different processors using InCoDe. The whole point-set is accessed by each processing element. Each partition must contain at least one point inside a box. Triangles of inner boundary regions are constructed on both adjacent partitions. Merging of subdomain triangles can be simply done by removing redundant triangle from one of the neighbors. The parallel efficiency is compromised by the redundant construction of merging region triangles. According to the reported implementation results of authors [CIGNONI ET AL., 1993], the ParInCoDe algorithm realizes a speed up ranging from 1.79 to 19.01 on a nCUBE system model with 2-64 processing elements. From the [HARDWICK, 1997] test result of InCoDe algorithm, a 10 times slower performance was noticed for a non-uniform point-set.

In the work of [BLELLOCH ET AL., 1996] a practical parallel Delaunay triangulation is presented which shows a good performance on general point distributions commonly found in most scientific computations. This algorithm uses a projection based approach to reduce the 2D Delaunay problem to a 3D convex hull points. The 2D points are projected on a sphere or paraboloid to get convex hull point-set.

In the [HARDWICK, 1997] parallelization approach, triangulation was implemented with a combination of [BLELLOCH ET AL., 1996] a projection based approach and based on the fastest serial D&C algorithm [DWYER, 1987]. The author uses a D&C serial algorithm from the fastest CPU triangulation work (Triangle) of [Shewchuk, 1996]. The partitioning of point-set is decided according to an orthogonal line that is passed through the median of x or y coordinates. With respect to this orthogonal partition line, a paraboloid is constructed and all the points are transformed onto the paraboloid. A set of line segments on a plane is formed by projecting the lower convex hull of the parabolid back to the plane. These line segments build a joint of both partitions. According to the reported results of this author, the algorithm realizes a very good speed up which ranges from 1.8 – 5.8 for a 2 – 8 processors for a uniformly distributed point-set of 128,000 points. This speed up was experienced with a SGI Power Challenge with shared memory.

In [CHEW ET AL., 1997], a parallel construction of Delaunay triangulation is presented by dividing the mesh domain into several subdomains and triangulating in parallel using distributed computing processors. The subdomain boundaries act as constrained edges which allow the independent triangulation calculation in adjacent subdomains. During the preprocessing step, these subdomain boundaries are created. To get a good parallel

performance and quality triangles, the identification of these common boundaries should be done with great care. Dividing boundaries should be established in such a way that each processor get nearly the same amount of work load. Boundaries are not supposed to make a small angle between them and they have to be spaced well enough to produce good quality triangles. Since the interface between neighbor subdomains is a constrained edge, there is no requirement for synchronization between different processors and it requires only a minimal communication between adjacent subdomains. During the parallel meshing process, constrained edges are kept as common to both neighboring sub domains; required communication is only a splitting of edge message which is passed between both domains. Even though this parallel work produces good quality Delaunay triangles according to this group, artificially inserted constrained edges in the mesh is not preferable in all cases.

Along the time line of parallel Delaunay triangulation research, many of the approaches were using the fastest and easiest D&C technique. In the work of [LEE, 1997], he has tried a different approach. He has used an incremental construction method which is suitable for large scale parallelization. To each processor a certain number of points (point-set P) and complete point-set S is passed. Each processor calculates edges from its own point-set P using the closest point criteria. Each time the complete point-set is used to test the closest point. After each processor derived the edges within its point-set, the complete edge list is collected by the first processing element. After removing redundant edges, the respective edge lists are passed to each processor. Two closest points of each edge are calculated as one in one half-space and the other is in other half-space. From these two points two new triangles are constructed. This step is repeated until no new triangles are found. According to the author, the implementation was done on an Intel Paragon system and there were no results published.

There are few parallel approaches to refine the existing Delaunay triangulation (DT(S)) and form an improved quality Delaunay triangulation. The parallel algorithm from [OKUSANYA AND PERAIRE, 1996] is an upgraded version of the best sequential algorithm of [CHEW, 1989]. This algorithm decides to insert additional points (the circumcenters of triangles) into existing triangles to create better shaped triangles. In Okusanya's parallel version existing triangulation is partitioned and distributed among processors. During the concurrent insertion, if points do not fall within their partitions, these points are passed to other processors. To avoid inconsistency, these points are allowed after all concurrent insertions. This algorithm includes some load balancing processes too. This algorithm is implemented using the MPI technique. According to the authors report, they only achieved a speed up range of 1.6-3.0 for 2-8 PEs for uniformly distributed one million points. The test was done on an IBM SP2

system. The obvious reason for a poor speed up can be a higher communication cost among the distributed system. The same authors have published a similar algorithm [OKUSANYA AND PERAIRE, 1997] for 3D triangulation. The reported parallel speed up here is nearly 1.2 – 2.3 for 2 – 8 PEs for 144,600 points.

The parallel D&C Delaunay triangulation approach presented by [CHEN ET AL., 2006, 2001, 2001] is implemented using both D&C and incremental construction techniques. This group has brought the parallelism using the message passing MPI technique on a distributed computing system. Each processing element triangulates in parallel using the fastest serial D&C triangulation algorithms [DWYER, 1986]. This algorithm is called by this group as Independent Parallel Delaunay Triangulation (IPDT). The interface region, which is a single layer of triangles on each internal boundary, is built using an incremental construction method. This algorithm is based on [CIGNONI, 1994] a DeWall algorithm. The argument of these authors is that though their algorithm leads to a higher time complexity in the interface construction, this phase of calculation is limited only to a small set of boundary triangles. Calculation of the interface is performed sequentially. According to the author's reported results, the algorithm realizes an outstanding speedup for uniformly distributed points. For example, a 96K point-set shows a speed up from 1.57 – 4.95 for computer system with 2-8 processing elements with IBM SP2 with High performance FORTRAN.

A work from [LEE ET AL., 2001] is a combination of the work of [HARDWICK, 1997] and the work of [CIGNONI ET AL., 1993]. The projection based partitioning of a point-set into sub regions in x and y direction is used as in the [HARDWICK, 1997] implementation. The triangulation of sub regions is implemented in parallel using an incremental construction method. There is a difference between Hardwick's partitioning approach and Lee's approach. The first one partitions the point-set into two sub regions in every iteration according to medial line. The second one implements partitioning as a pre-process (before meshing starts) into number of sub regions according to the number of available processors. According to their presented results, their algorithm realized a speed up development from 1.36-12.5 for a computing facility containing 2 -32 processing elements.

Parallel Delaunay triangulation solutions using an incremental insertion technique are relatively rare. In the work of [KOLINEROVA AND KOHOUT, 2002], they suggest two methods which are suitable for shared memory computing on multi-core CPU parallel architecture. Their algorithm derives shared triangulation using a shared DAG (Directed Acyclic Graph) structure. Different parallel elements can simultaneously modify the DAG. Each PE inserts its

own subset of points. According to the authors reported results, their parallel performance realized a maximum speed up ranging from 1.73 – 5.84 with 2 – 8 PEs for up to one million points. The test was done on a Dell Power Edge 8450 system.

[NAVE ET AL., 2002] presents a parallel Delaunay refinement algorithm which computes on a distributed memory computing system. Their proposed algorithm is valid for a simple polyhedral domain where conforming boundary edges or surfaces must be separated by a large angle from 90° to 270°. They claim that their algorithm gives efficient results even for certain domains without this conforming boundary limits. This algorithm consists of two main steps, first, sequential mesh initialization and, second, parallel mesh refinement. The parallel algorithms starts with an initial domain decomposition, in this step the initial mesh M_0 is element-wise partitioned into N number of sub meshes S_k and distributed to P processing elements. The authors advise that over decomposing into a greater number of partitions (N>>P) can improve the memory utilization and enable efficient parallelism using dynamic load balancing. According to the authors this parallel refinement algorithm is unaffected by the interface triangulation between adjacent subdomains. Their reported results only compare the parallel refinement against the distributed sequential version of refinement implementation and show a six times improvement over the sequential version. There are faster 2D and 3D sequential refinement algorithms, which are not included to test against their parallel refinement algorithms.

[SPIELMAN ET AL., 2002] presented a study about parallel Delaunay refinement algorithms and analyses. According to the authors, there are two steps involved in this algorithm, which generates an independent point-set for insertion and updating Delaunay triangulation in parallel. Both of these steps are executed in parallel. The authors guarantee that their algorithm can produce the same good-quality triangulation like Rupperts' method (2D) and Shewchuk's method (3D). When compared with the performance of Chew's method for a quasi-uniform mesh, this algorithm is faster, reducing the iteration number to $O(log(L/s))$, where L is the diameter of the domain and s is smallest edge in the output mesh. By applying this algorithm in other parallel refinement approaches probably a better result can be achieved. Even though the authors have presented an algorithm and proof, no implementation results were reported.

The parallel Delaunay algorithmic finding from [KOHOUT ET AL., 2004] is targeted for a limited number of multiple CPU cores and shared memory which are available in general purpose PCs. The parallel algorithm of this group is based on randomized incremental

insertion. The three phases of this algorithm are location, subdivision and legalization. In the location phase, the triangle is identified to be subdivided. To find the triangle location there are different algorithms available, such as a random walk technique [GUIBAS, AND STOLFI, 1985], a technique by utilizing quadtrees and bucketing techniques [EDAHIRO ET AL., 1984]. The most frequently applied technique is the random walk technique for its low memory requirement. The expected complexity to find the location of one point using this technique is $O(N^{\frac{1}{4}})$. From [DEVILLERS, 1998] approach, to quickly find the triangle location, a hierarchical data structure like DAG is used which require less memory. After finding the triangle location, subdivision is followed. Finally, in the legalization phase, the circumcircle condition is validated on tested triangles by flipping necessary edges. This algorithm is used with the Directed Acyclic Graph (DAG) data structure and experiences an optimal complexity of $O(Nlog(N))$. The worst time complexity of this algorithm is $O(N^2)$. By maintaining the point insertion history using DAG, locating a point is possible in the optimal case with $O(log(N))$ and worse case with $O(N)$. Authors suggest that this worst case time complexity is only possible when DAG is "totally imbalanced" and this situation is improbable in their randomized insertion technique [KOHOUT ET AL., 2004]. The simplicity of the algorithm and the numerical robustness are the profiting features of the incremental insertion technique [KOHOUT ET AL., 2004]. With this technique only very few non-Delaunay triangles are possible due to incorrect or inconsistent Delaunay criterion. In the incremental insertion technique, incorporating the constrained edges is simply possible with small modifications. According to the authors, their proposed two methods (batch method and circumcircle) experience a good linear speed-up for an experiment with more than 100,000 points using two and four processer.

From the work of [BLANDFORD ET AL., 2006], they reached the 3D Delaunay triangulation of 10 billion tetrahedrons using shared memory programming with 64 CPU processor cores and 200 GB RAM. This algorithm uses concurrent insertion of points using the Bowyer-Watson insertion method.

From the latest parallel algorithms for computational geometric problems by [VICENTE ET AL., 2009], this group has developed some efficient algorithms with the aim to exploit the latest computing power of multi CPU processor cores and available large shared memory. According to these authors, these algorithms can give a significant performance improvement for this type of computing environment in the application of Delaunay mesh generation for any dimensional problems.

One of the recent parallel Delaunay mesh generation work from [CHRISOCHOIDES ET AL., 2009] is given, which deals with multi-layered parallel approach by using a hybrid algorithm. This algorithm is capable of handling around 10^{18} concurrent work units to process a tetrahedral or triangular mesh generation. According to these authors this mesh generation code can perform with the increased concurrency which is at least 10 orders magnitude of the state-of the art mesh generation codes. This algorithm is executed on a highly-scalable architecture as a combination of shared memory multi-core CPUs and distributed memory computer clusters. The Delaunay triangulation used in this implementation is the incremental insertion (B-W) scheme.

As a concluding remark, various parallel Delaunay triangulation techniques are already explored to exploit the large, medium and small scale CPU based parallel computing systems. These parallel techniques are available as different algorithms and implementations.

2.2 GPU-Based Parallel Delaunay Triangulation

With the introduction of GPU computing, parallel algorithms are explored to bring data parallelism by exploiting multi-grain processors of GPU devices. The major challenges in dealing with unstructured mesh generation result from frequent inter-thread communication requirements and the requirement of memory to deal with large data structures. GPU performance is poor for calculations that deal with large memory handling and frequent communication between threads. Due to these hardware limitations, unstructured mesh generation algorithms using GPUs are not favorable at first glance. Application of GPU parallel computing techniques in Delaunay triangulation is in the very early stage. As listed in the Table 2.2, there are very few preliminary and promising approaches based on GPUs giving significant performance improvement against their sequential counterpart. GPU-based Delaunay triangulation GPU-DT by constructing digital Voronoi diagram [RONG ET AL., 2008, QI ET AL., 2013] gives better performance for uniformly distributed random point-sets. Transforming any given arbitrary triangulation into a Delaunay triangulation by flipping edges which is implemented parallel using GPU [NAVARRO ET AL., 2011]. Another research approach is from [NELSON AND LUNNAN, 2012], in which it is intended to study the parallel performance of GPUs in constructing Delaunay triangulation using incremental construction. A recent unpublished approach of [VASILEIOU ET AL., 2012] constructs Delaunay triangulation by partitioning point-sets into several partitions and calculating Delaunay triangulation on GPU using "order recursive insertion algorithm". The brief details of these GPU parallel mesh

generation approaches are discussed below, for further details, the reader is advised to refer to the respective references.

GPU-DT is a hybrid parallel implementation for generating two dimensional Delaunay triangulations. The most recent updated release of this triangulation is integrated with constrained edges for a planar straight line graph (PLSG) [QI ET AL., 2013]. Most parts of this implementation are parallel and accelerated using a single GPU. Still, some algorithmic parts are serially computed in the CPU. Detailed information about the GPU-DT algorithm can be found in [RONG ET AL. 2008], and [QI ET AL., 2013].

Table 2.2 Some parallel Delaunay triangulation algorithms and implementations using GPU based parallel architecture

Type of Delaunay algorithm	GPU parallel computing technique	
	Single GPU	Multiple GPUs
Delaunay triangulation using digital Voronoi diagram	QI ET AL., 2013	——
	CAO ET AL., 2010	
	RONG ET AL., 2008	
Delaunay triangulation by edge flipping of arbitrary triangulation	NAVARRO ET AL., 2011	——
	NAVARRO ET AL., 2011	
Delaunay meshing	NELSON AND LUNNAN, 2012	——
	VASILEIOU ET AL., 2012	

The triangulation performance of GPU-DT for uniformly distributed random point-sets is significantly faster compared to any other serial CPU Delaunay triangulator. The algorithm of GPU-DT is implemented as a hybrid parallel process, using CPU and GPU. Currently existing GPU-DT implementation works with a single GPU and uses global and texture memory. Since the available global memory in GPU devices is limited, this single GPU version of Delaunay triangulation can handle only a limited number of point-sets. However, there is a lack of concepts using multiple GUP clusters for Delaunay triangulation and unstructured mesh generation.

In the [NAVARRO ET AL., 2011, 2011] research approach, this group implements a GPU parallel edge flipping algorithm which transforms any given arbitrary triangulation into a Delaunay triangulation. The edge-flipping technique to derive a Delaunay triangulation from any triangulation was first introduced by [LAWSON, 1972]. The latest work of GPU-DT [QI ET AL., 2013] is incorporated with a completely parallel edge-flipping implementation at the GPU level. In the GPU-based parallel work from [CERVENANSKY ET AL., 2010], a similar edge

flipping GPU parallel technique is implemented. In Navarro's paper they state their approach differs from both of the above existing approaches in the data structure and the GPU memory used. From their reported result, performance comparisons show that the GPU base edge-flipping algorithm gives a very high speed up over a CPU based serial implementation.

From the [VASILEIOU ET AL., 2012] research concept to construct Delaunay triangulation using GPUs, they propose partitioning point-sets into specified number of vertical and horizontal level partitions. The parallelization concept in this algorithm contains three main steps: first, partitioning and sorting points; second, triangulating each subset in parallel using GPUs and third, merging the sub meshes. Each partition contains a nearly equal number of points which are sorted in their x coordinate value. Their proposed algorithm is performed on a Gaussian and mixture of Gaussian distributed point-sets. Delaunay triangulation is calculated using order recursive insertion algorithm using GPUs. In the merging phase, merging vertical subsets in a horizontal level is performed by mapping onto a binary tree. Similarly, horizontal zones are merged into a complete mesh. The presented results from this proposed algorithm shows a good speed up over the best CPU serial Delaunay triangulator Triangle [SHEWCHUK, 1996]. They report their proposed algorithm performs better than the GPU-DT algorithm.

From a recent research approach from [NELSON AND LUNNAN, 2012], they have studied the parallel GPU performance of Delaunay triangle construction using an incremental construction technique for a two dimensional point-set. This work is almost similar to the original data-parallel algorithm from [TENG ET AL., 1993]. To reduce the required time of independent thread calculation during expanding triangulation, a bucketing technique is proposed to limit the point searching time within a small set of points. The report from Nelson's work is basically a preliminary result of performance comparison for a small point-set (2^32) and an experience in implementing this algorithm for GPU parallelization.

As a concluding remark, parallel Delaunay triangulation techniques to utilize the modern GPU based parallel system are very new. There are only very few implementations available. All of them are only able to utilize a single GPU system. To date there is no parallel Delaunay triangulation implementation to support for multiple GPUs in a computing system.

Chapter 3

3 Parallel Computing

This chapter gives a small introduction to general parallel computing and modern parallel computing using graphics card processors (GPUs). The important parallel computing aspects are discussed in this section. Traditional CPU-based parallel computations are carried out using shared memory and distributed memory techniques. The open multi-processing (openMP) API technology [OPENMP, 2013] to program on shared memory architecture and the message passing interface (MPI) standard API technology [MPI, 2013] to operate the distributed memory architecture are briefly discussed in section 3.1. In this section some of the fundamentals of parallel computing concepts are included. In the section 3.2 GPU parallel computing is described with necessary information to understand this thesis. Since the GPU implementation is based on CUDA technology, the basic details of GPU computing and CUDA programming models are also discussed.

3.1 Development of Parallel Computing

As shown in figure 3.1, the records of the performance history over last three decades show the growth in computer performance. According to [HENNESSY AND PATTERSON, 2007], the annual growth of more than 50% in computer performance was sustained for around 16 years from 1996 until around 2002. The continuous development in microprocessor technology was the main reason for this performance improvement. Since 2002 the growth rate slowed down to around 20% due to the limitation in power and long memory latency in uni-processor technology. Around 2004, microprocessor technology reached its peak performance, and then the computer research community started to think about a different approach to improve computing performance. The result of this research for new technology leads to the parallel execution of multi-core processor technology.

Parallel computation is an extension of traditional sequential computing, by dividing a computational problem into several small problems, then computing each one in parallel using many processing elements. The historical motivation for parallel computing is the limitation of further improvement in microprocessor speed. When the frequency scaling of processers reached its maximum clock speed, the idea of parallelism came into existence to increase the computing performance [ASANOVIC ET AL., 2006].

Though this concept already existed, the first variant of this computing paradigm using the MPI was introduced in 1994 [MPI, 2013]. Distributed computing using MPI is commonly used today for general purpose numerical computing in research and study applications. To perform parallel computing using multi-core CPUs, a further computation technique was developed. This is called openMP, which enables the parallel execution of computation using shared memory computing. The first release of the openMP standard was in 1997 [OpenMP, 2013]. The continuous developments of computing standards and continuously falling computer prices have attracted the scientific community to incorporate parallel computing in their research. There are many facilities available for parallel computing such as standards, advanced parallel compilers and high level language support.

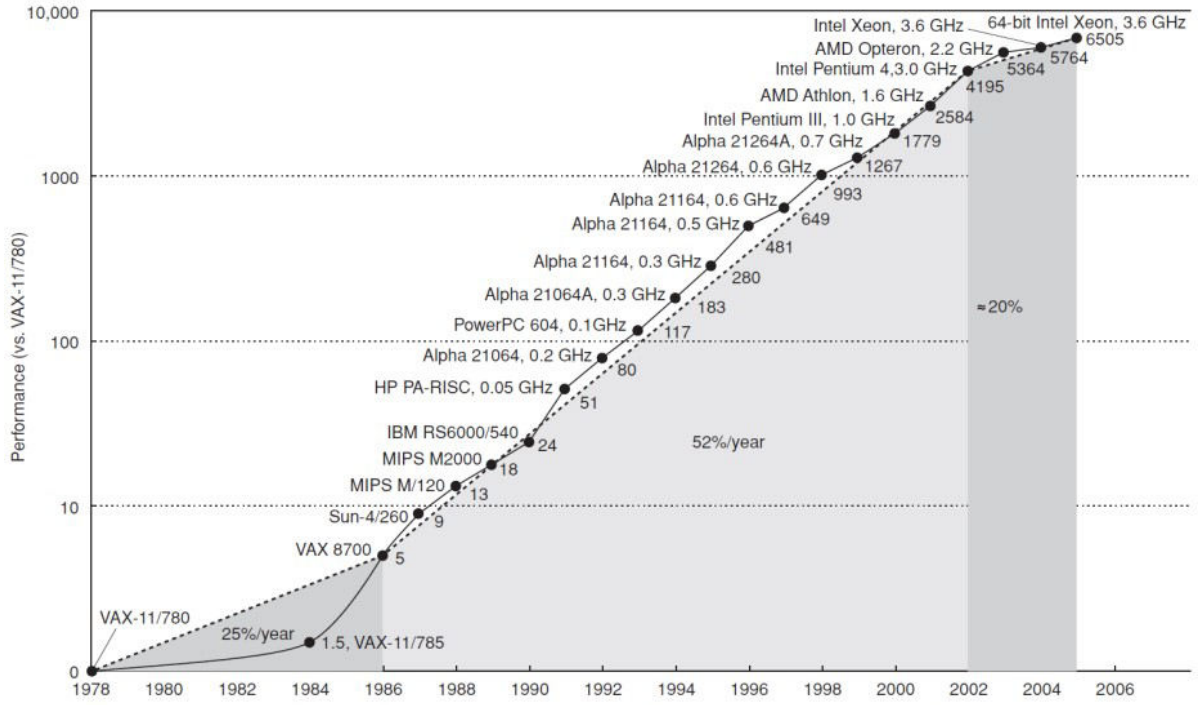

Figure 3-1 The growth of computing performance over the last three decades according to parallel computing landscapes [HENNESSY AND PATTERSON, 2007]

3.1.1 Flynn's Taxonomy

The classification of hardware in parallel computing systems is done using the main three characteristics. First is the memory type, it can be distributed, shared or a combined type. Second is the number of instruction streams, it can be single or multiple. The third one is granularity, which is given by the number of processors.

According to the Flynn's taxonomy [FLYNN, 1972], parallelization architecture can be differentiated under these categories as shown in the table 3.1. It classifies programs and computers by whether they are operating using a single set or multiple sets of instructions, whether or not those instructions are using a single or multiple sets of data. The SISD (single-instruction-single data) architecture is a traditional sequential computing method using a single processor. The MISD (multiple-instruction-single data) type is not frequently used, can be used for specialized applications of large data sets in large computing systems or super computers. The SIMD type is the kind of computing done by executing the same operation repeatedly on multiple data sets. The MIMD architecture is a commonly existing parallel computing program. This computing is carried out using asynchronous parallel techniques to handle different operations on different data.

Table 3.1 Flynn's taxonomy

	Single Data	**Multiple Data**
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

3.1.2 Classification of Parallel Algorithms

In general parallel algorithms are classified into main three types (1) is processor farming, (2) geometric parallelism and (3) algorithmic parallelism [TOPPING AND IVANYI, 2010]. In the processor farming parallelization method, each processor in the network executes code in isolation from all other processors except for a master or root processor. This parallelism is also called task farming or event parallelism. In general, with the processor farming algorithm technique, communication between worker processors is not permitted.

In geometric parallelism, each processor executes the same code on data corresponding to a sub-region or sub-domain being simulated or processed. In this type of parallelism, communication between inter-processors is possible to allow the exchange of boundary data between neighboring processors which represent the connections between the subdomains. The other type of parallelism is algorithmic parallelism in which each processor is responsible for a part of the algorithm and the data passes through each processor in turn. Different parts of the algorithm are executed on each processor. To avoid a bottleneck in parallel processing,

the computational load on each processor should be same or balanced [TOPPING AND KHAN, 1996]

3.1.3 Shared Memory Parallel Computing

The openMP parallel computing model utilizes the multi-core CPU processors and shared memory of the computing system. It is structured as an application programming interface (API) standard and uses the multi-threading capability of multi-core processor computers. This standard supports multi-platform portability and programmability with different programming languages (C, C++ and Fortran etc.). Basically, the openMP standard consists of main three components, which are compiler directives, a small set of runtime libraries and environment variables [OPENMP, 2013]. It supports data and task parallelism. As shown in figure 3.2 (a), the data-shared parallel architecture stores the processing data into commonly available shared memory and data is shared between different processors.

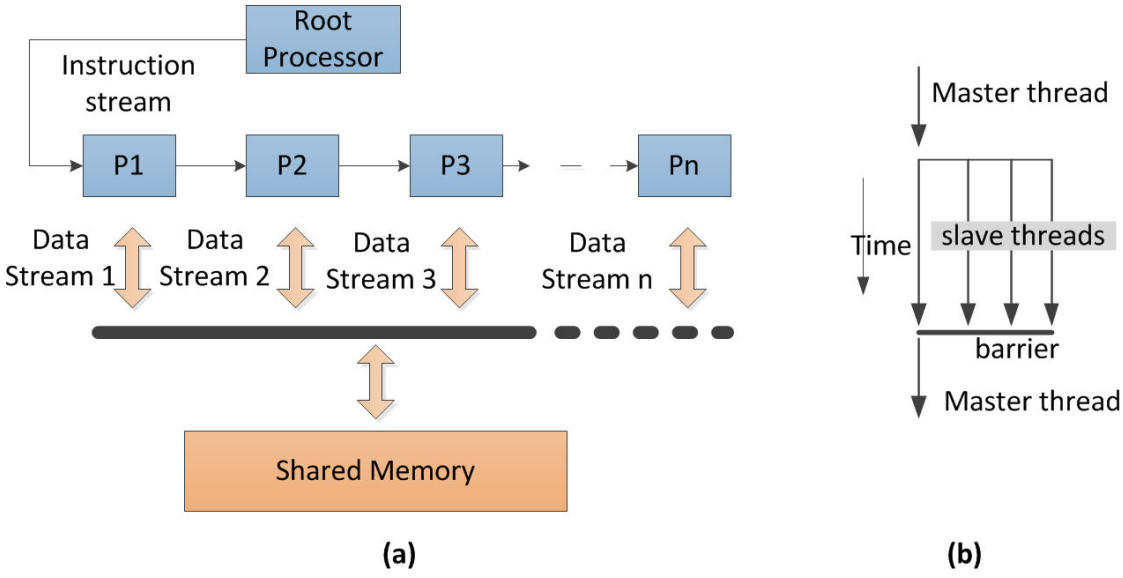

Figure 3-2 (a) Shared memory programming model, **(b)** OpenMP parallel execution model (fork-join)

A typical parallel problem, which is highly suitable to this parallel programming model, is a computation loop where operations are repeated multiple times. By integrating the openMP standard into the necessary programming segments, loop iterations are divided and assigned into multiple threads. As shown in figure 3.2 (b), the well-known fork-join parallel execution model suits the openMP invocation in computing. The root processor controls the master thread and creates the necessary slave (worker) threads to be executed on other processors by using openMP directives at the necessary locations along the program execution time line.

Master and slave threads execute the computation in parallel. A Master thread establishes a barrier until all other threads finish the computation and proceed to other sequential steps.

3.1.4 Distributed Memory Parallel Computing

The distributed memory parallel programming architecture is the most common parallel programming model used among diverse parallel computing systems. This model uses the message passing technique to coordinate the computation among several computing nodes in parallel. The MPI parallel standard is associated with the message passing programming. This is a popular and demanding parallel computing standard used in the HPC-cluster applications [CHEN ET AL., 2001]. This parallel technique handles mostly multiple instructions of multiple data (MIMD) using the distributed memory architecture. But the SIMD model is also well suited to this parallel technique. As shown in figure 3.3, a typical distributed computing architecture is a network of many computing nodes and all the nodes have their own memory space. There are different MPI implementations available. Among these OpenMPI and MPICH are well-known and frequently utilized in message passing computation. These standards make possible the multiple platform portability and different programming language support. The MPI technique is facilitated with different sophisticated library support for coding, debugging and testing the performance of message passing programming. The highly standardized nature of the MPI method has a lot of advantages, such as enabling portability, reusability of code on different hardware and software platforms [YANG ET AL., 2011]. The MPI system provides a sophisticated API, which permits a user to develop parallel programs using the message passing technique.

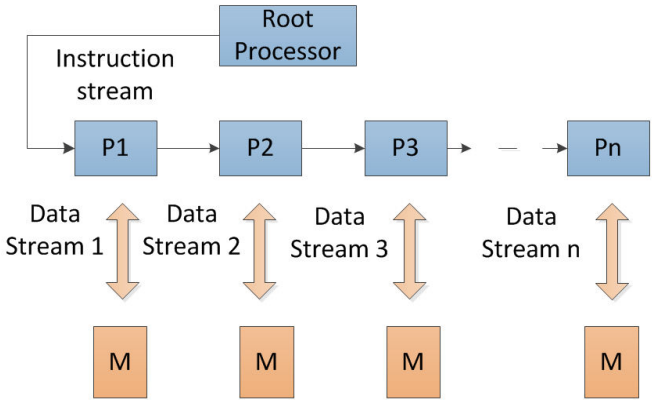

Figure 3-3 Distributed memory programming model

3.1.5 Commonly Used Terms in Parallel Computing

The major objective of parallel computing is executing a program faster than a corresponding non-parallel version. Theoretically, if a parallel program runs on a network of N_p processors, then it should be N_p times faster than the time consumed on single processor. But in reality, this theoretical speed performance is not always achieved by parallel implementations. The main reasons for this limitation are the increased communication overhead due to the increased number of processors and the possible load imbalance in distributing the parallel tasks among processors. To experience an optimum parallel performance, a user has to take care of load balancing and minimizing the requirements of inter-processor communications.

For a homogeneous parallel system the speed-up (S) is defined as

$$S = \frac{T_{seq}}{T_{N_p}}, \tag{3.1}$$

where T_{seq} is the time taken by the code to execute on a single processor. T_{N_p} is the time taken for the same program to be executed on a parallel system with N_p number of processors. A homogeneous parallel system is a term for a network of processors equally capable in computing performance. In parallel programming, parallel efficiency (E) is also considered as an assessment factor in comparing parallel computing systems and parallel algorithms. The efficiency of a parallel system for a particular algorithm is defined as

$$E = \frac{S}{N_p}. \tag{3.2}$$

Amdahl's law [HILL AND MARTY, 2008] states that the maximum speed-up S_{max} by a parallel computing system with N_p processors, in which fraction f of operations is still in sequential form, is

$$S_{max} \leq \frac{1}{f + \frac{1-f}{N_p}}. \tag{3.3}$$

According to this law, one can understand that the existence of a significant percentage of sequential operations in a parallel algorithm can negatively influence the speed-up.

The scalability of a parallelization problem is an important aspect. It represents the ability to scale the parallel problem on any particular parallel architecture. This means that the scalability of a parallel algorithm is a measure of how well the performance increases as the number of processes increases. If an application is scalable, then the run time will remain the

same when the size of the problem and the number of processors are increased by the same factor n.

3.2 GPU Parallel Computing

3.2.1 GPGPU Computing

In the current computing era, GPUs (Graphic Processing Units) are attractive to parallel computing application developers. The enormous computing capability of GPUs is utilized in general purpose computing in addition to CPU computing. General purpose computing using GPUs (GPGPU) is the greatest concern of research and developers. There are diverse application fields exploiting GPU capabilities [GPGPU, 2013]. Traditionally GPUs have been targeted only for graphic applications which accelerate the image processing in visualization. Modern graphic cards are specially designed for multi purposes such as graphic processing and numerical computing. When compared with the CPU, the GPU is a low cost hardware which has many hundreds of computing cores and tremendous memory bandwidth [KIRK AND HWU, 2010].

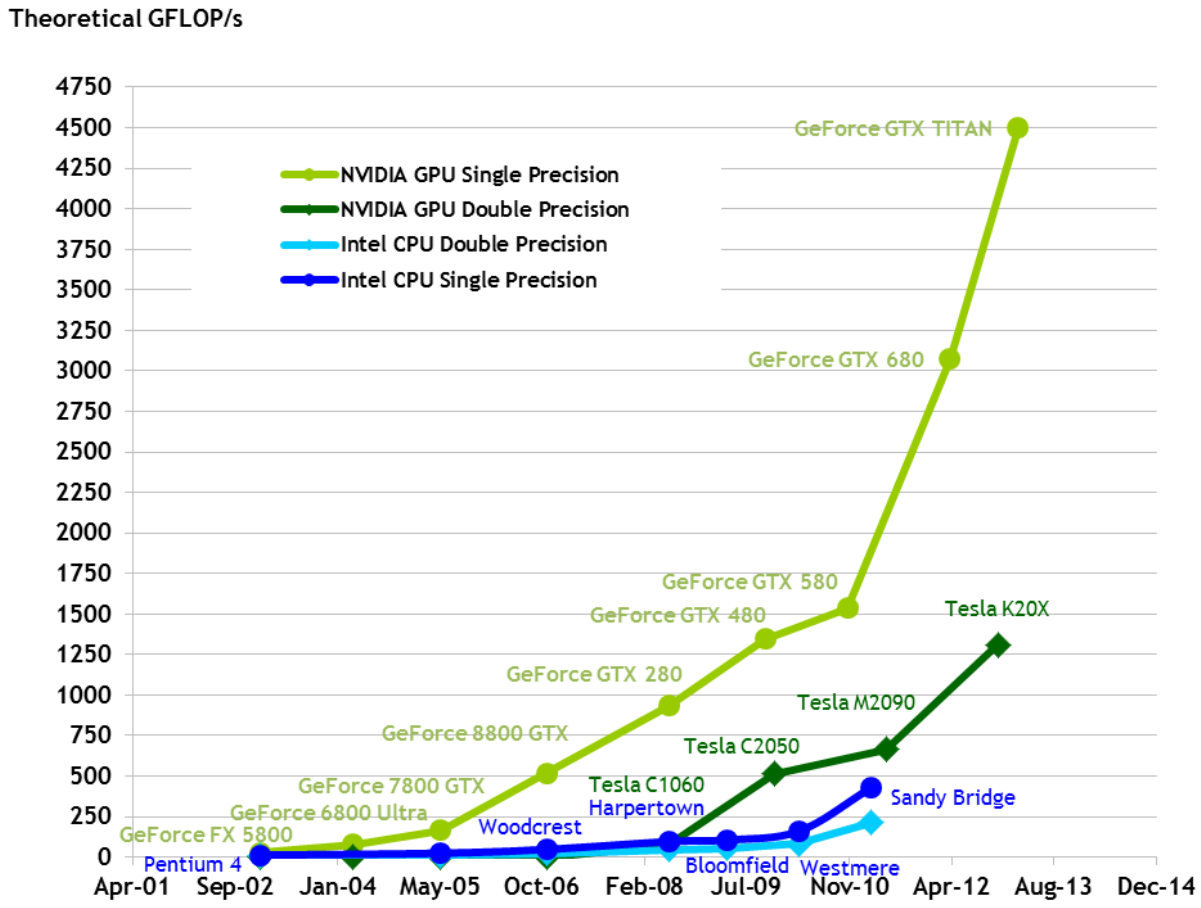

Figure 3-4 Floating point operations per second (FLOPS) of CPU and GPU [NVIDIA3, 2014]

The historical development of GPUs shows that computing power increases with time for both single and double precision calculations. As shown in figure 3.4 the theoretical performance of modern GPUs and CPUs are compared against the time. In this benchmark, the best known graphics cards from Nvidia are compared against historical best CPU releases. Nvidia is one of the market leaders in graphic hardware development compatible for GPGPU computing. The performance of CPUs increases slowly over time for double and single precision calculations. But the GPUs performance is rapidly increasing over time. Particularly single precision performances of GPUs increases by a factor of second order in current GeForce cards compare to CPU processors. The double precision performance of Tesla cards also shows the increased performance over modern CPUs. As an example, the single precision performance of GeForce GTX 680 has reached more than 3TFLOPS. The memory band width capability has increased very much over the recent years (see the plot in figure 3.5). For example, the technology in GeForceGTX680 has reached a bandwidth of around 190GB/s.

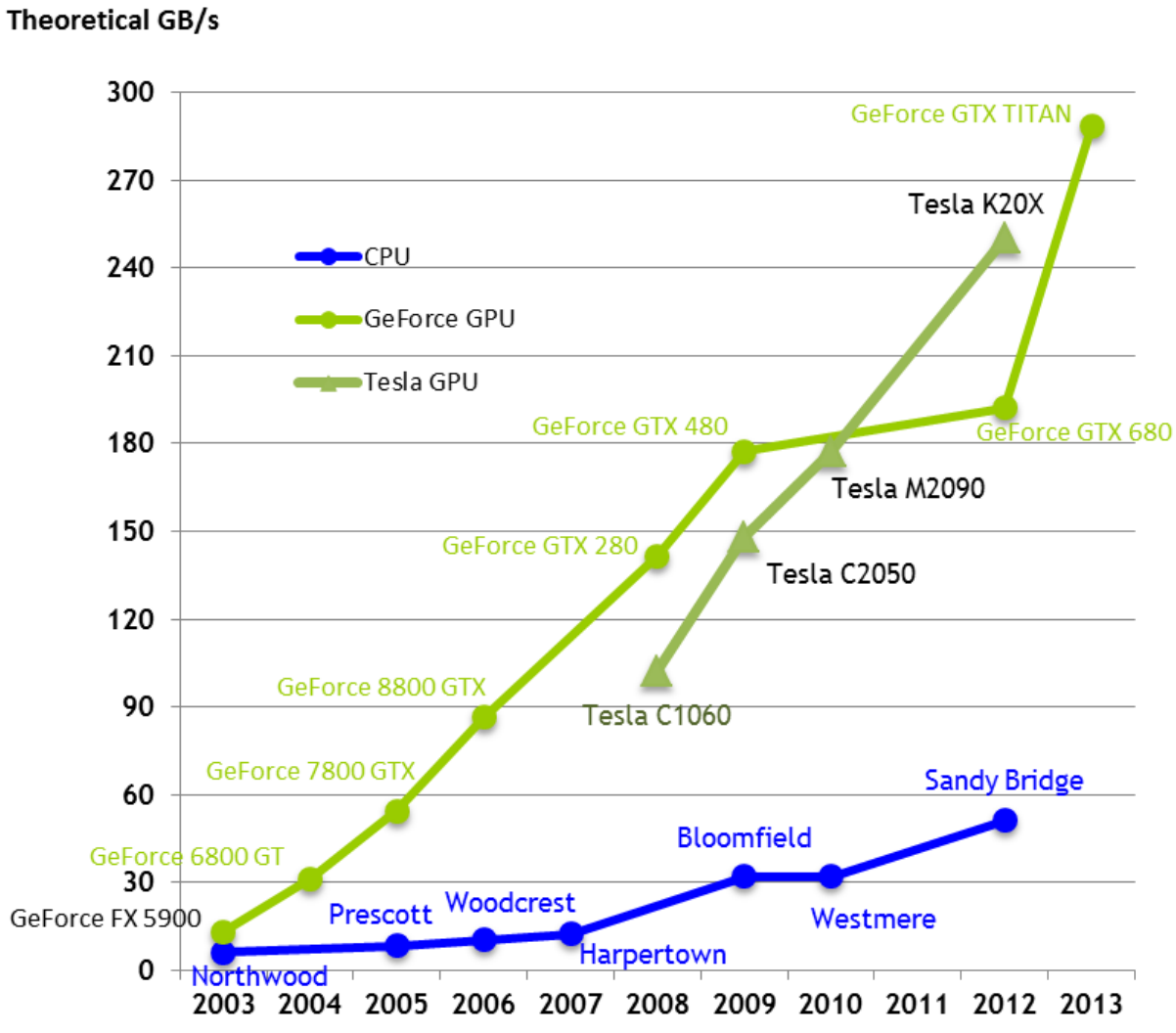

Figure 3-5 Comparison of memory bandwidth for the CPU and GPU [NVIDIA3, 2014]

GPUs are best suitable for computational intensive data-parallel problems [FARBER, 2011]. The current GPUs are several orders of magnitude faster than the modern CPUs and integrate many hundreds of streaming multi processors. As an example the very latest (to date) Kepler graphic card (called Tesla K20x) contains 2688 streaming multi processors [NVIDIA2, 2013].

Traditionally used CPU computing belongs under the category of latency oriented design philosophy. Latency is a measure of time delay for an instruction to be processed in a system. Modern GPUs are designed according to a throughput oriented design philosophy. The design goal of GPU technology is to maximize the instruction throughput. That means processors execute as many instructions as possible at a same time [KIRK AND HWU, 2010].

In figure 3.6 (a) CPU processor chips are designed to optimize the latency of instructions. The current CPUs come with on-chip large cache memories. The design is intended to convert long latency memory accesses to short latency cache accesses. These chips are facilitated with sophisticated control mechanisms to handle branch prediction for reduced branch latency and data. Also, CPUs contain powerful ALUs (arithmetic and logic units) which have reduced operation latency [KIRK AND HWU, 2010].

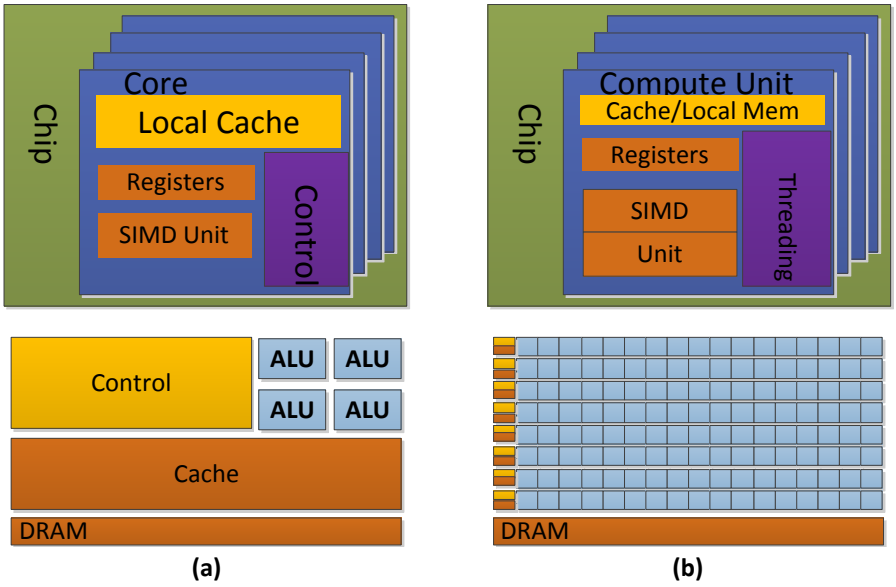

Figure 3-6 (a) Latency oriented CPU cores **(b)** Throughput oriented GPU cores

CPUs but they are capable of boosting memory throughput. The control mechanism is simple and does not have branch and data forwarding facilities like CPUs. ALUs are energy efficient and available in large numbers. They are heavily pipelined for high throughput. Computing is required by a massive number of threads to tolerate the long latencies. For efficient GPU

parallel computing both CPUs and GPUs have to be utilized in such a way that CPUs are for sequential parts, where latency is a matter of concern, and GPUs are for parallel parts to gain more throughput [KIRK AND HWU, 2010].

3.2.2 GPU Computing Using CUDA

In GPU technology, advancement in hardware performance and programming capability for parallel applications have been significantly developed in recent years. Nvidia's CUDA (compute unified device architecture) is one of the parallel programming development environments available for GPU computing applications. CUDA is supported by most of the modern graphic cards from Nvidia. The CUDA toolkit is available with developing facilities using high level languages such as C++, C and FORTRAN. This includes Nvidia's nvcc compiler, sophisticated math libraries and tools which are capable of supporting debugging, optimizing and profiling during the programming development. The CUDA programming model is simple and easy to learn by a programmer. A CUDA API is an extended version of the C language with additional built-in keywords, function qualifiers, variable qualifiers, intrinsic functions and function calls.

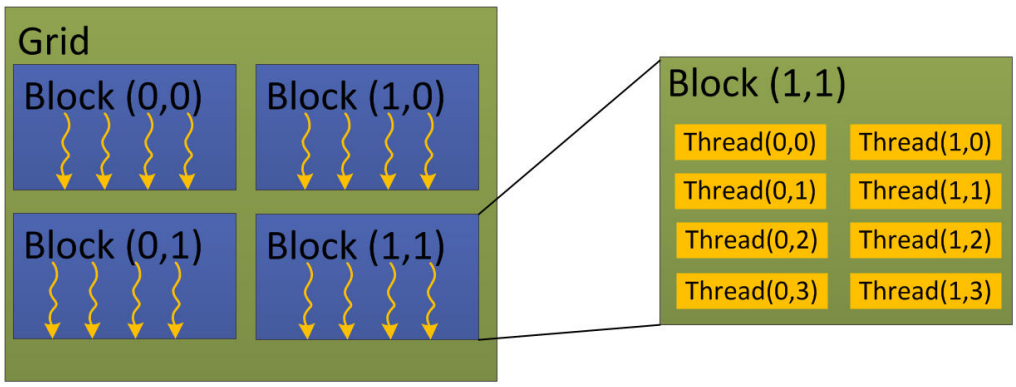

Figure 3-7 CUDA block and grid layouts

The CUDA parallel architecture is a similar to the SIMD type of parallel scheme. The CUDA technology developer NVIDIA denotes this as a SIMT (single instruction multiple threads) architecture. The multiple streaming processors available in the CUDA architecture handle the SIMT technique by allocating each processor to one thread. Threads within a so-called warp process the same operations together. If a thread within a warp tries to do a different operation, then the whole thread's execution become a serial operation. To get a maximum parallel performance, branching within a warp should be reduced or avoided [FARBER, 2011].

In the CUDA programming model, the GPU is viewed as a computing device which operates as a coprocessor to the main CPU (host). The functions which are data-parallel and computation intensive should be off-loaded to the device. The most suitable candidate functions are executed many times, but independently, on different data. As an example, a function containing a for-loop is one of the preferred problems. CUDA is designed in such a way that thousands of threads can be executed in the kernel simultaneously. Each thread executes the same code, but processes over multiple pieces of data [FARBER, 2011].

In general, CUDA applications are a combination of several data-parallel blocks and non-parallel parts. The CUDA execution model is a heterogeneous host and device application. The serial parts are executed on the host and parallel parts are processed in device. The programming parts which are compiled for the device are kernel functions. The kernel is executed on devices by generating many different threads. Each thread that executes the kernel has a unique thread id which is accessible within the kernel through built-in *threadIdx* variable [KIRK AND HWU, 2010].

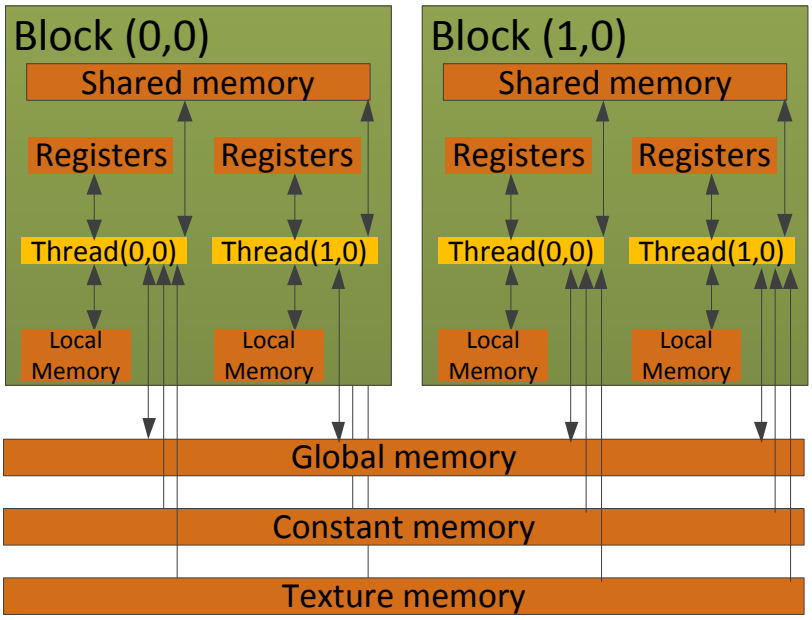

Figure 3-8 CUDA memory model (according to [NVIDIA1, 2012])

In CUDA, execution of each thread does a part of the same kernel program. The concurrent execution of many threads is called multithreading [NVIDIA1, 2012]. Within a CUDA framework threads are organized as blocks of threads. These thread blocks are organized within a grid. As shown in the figure 3.7, computational grids are built by an array of thread blocks. The layout of a grid can be defined as one or two dimensional. The block layout can

be defined with three dimensional indexes. Each block has a unique block ID and each thread has a unique thread ID. The maximum sizes of grid and block dimensions are selected according to the kernel complexity and available of GPU memory. The task of a CUDA programmer is to write a kernel function and assign the thread block dimension for the optimized execution of the largest number of parallel threads as possible.

3.2.2.1 GPU Memory Model

Both the host (CPU) and the device (GPU) maintain their own memory, which are called host memory and device memory, respectively. Data is copied back and forth between the host and device memories during CUDA execution (as shown in figure 3.8). On GPUs there are different types of memories available. The types of memories used in CUDA are differentiated mainly according to size, access delay and visibility (scope) to programming parts. The data transfer between the host and device and the selection of a suitable memory type should be decided by a programmer to get the best CUDA performance.

During CUDA programming, the programmer has to care about the memory transfer between the device and host. A frequent data transfer significantly influences the overall parallel performance of CUDA execution. To minimize the low bandwidth memory transfer between the host and device, the programmer has to decide in such a way that necessary data should be copied to the device at once, and then exploit the faster access of data from the device itself. The different usages of GPU memory types are discussed below according to the collected references [KIRK AND HWU, 2010, NVIDIA1, 2012, and FARBER, 2011].

Global Memory

The global memory is an off-chip memory, also called the device memory. This is a dynamic random access memory (DRAM) and available in largest size when compared to other memory types. Memory access by a thread to the global memory is slower than access to other memory types. The accessing speed can be improved by keeping a memory access pattern (as an example, memory coalescing). In general, the memory access latency is around 200 clock-cycles.

Constant Memory

This is an on-chip cache memory as a read only type. This memory type has an access scope to the threads within a kernel as a read-only access. The size of memory (around 64

Kilobytes) is relatively small compared to global memory. The access speed is significantly faster than the global memory (only around 4 clock-cycles).

Registers

Registers and local memories are used for simple and less durable calculations (life time of thread). They have thread level scope to write and read data. This on-chip memory is very small in size and gives extremely fast access to CUDA local variables during kernel execution. Overloading of local variables in registers can negatively influence the parallel performance.

Shared Memory

The shared memory is located on-chip directly with the multi-processors. This offers a very low latency memory access. When compared to global memory access, utilizing shared memory has an attractive performance gain over 100x [FARBER, 2011]. Shared memory is shared between threads within a block during kernel execution. This memory is only allowed for threads within a block, there is no possibility of access by threads from different blocks. In general, the size of the shared memory is around 16 Kilobyte per block. The access delay for shared memory is only around 1 clock-cycle.

Texture Memory

This memory type is a cache type read only memory and available for both device and host. Data should be populated before the CUDA execution begins. This provides a great amount storage with relatively fast access speed compared to global memory. The access speed for a data element in this memory is dependent on the requested element. In case the element is already in cache, then the access time is the same as the latency of the cache, otherwise, it will be equal to global memory latency.

3.2.2.2 Main consideration for better GPGPU performance

According to Farber's observations [FARBER, 2011], there are three important rules to get a high performance from a CUDA GPGPU program. They are:

1. All necessary data should be copied to the GPU and kept there as long as possible.
2. Exploit the GPU by assigning enough jobs.
3. Try to reuse data in the GPU by considering the band-width limitations.

The first rule can be illustrated as follows. Since GPU devices are external hardware, which is plugged to the host computer through the Express PCI bus. This PCI bus is very slow (more than 20 times) compared to the GPU memory transfer system.

The explanation for the second rule is illustrated as follows. The modern GPGPU system can realize more than one teraflop performance. For a large number of simple operations, the GPGPU is faster than the host in performing the same computation. In general, the programmer has to care about the time overhead to start a kernel on the GPU and data transfer between the host and the device. To overcome these overhead penalties, many hundred thousand of threads have to be involved in computation in parallel. Keeping the threads idle during kernel execution will compromise the parallel performance. To keep the GPU busy, there are possibilities to execute multiple small kernel functions (on a single card) simultaneously in modern Fermi CUDA architectures [NVIDIA1, 2012].

The third rule can be explained as follows. Data reusability is a general philosophy, even in conventional computing, to attain better performance. In the GPU system the limited memory band-width between the device and the host is a bottle-neck for overall performance. The availability of diverse memory subsystems (cached on-board and off-board memories) which have low latency access and a high band-width bus system facilitates optimum memory access. During a CUDA program execution, exploiting these memories for intermediate recording and reusing intermediate results, rather than copying back and forth to host and device, will give a significant performance improvement.

Chapter 4

4 Research Background

This chapter is dedicated to give some background information in the field of computational geometry and the application of parallelization. In section 4.1, frequently used basic concepts in the field of triangulation and mesh generations are described. Following this, in section 4.2, the detail of GPU-DT algorithm and the necessary GPU based algorithms which are used in GPU-DT computation are described. Finally in section 4.3, different domain decomposition methods which are mainly used in mesh partitioning are discussed.

4.1 Triangulation and Meshing Concept

This section, the concepts and basic properties of computational geometry which are used in this thesis are presented. Detail information can be read from various computational and meshing related references such as [BERG ET AL., 2008, GEORGE AND BOROUCHAKI, 1998, FREY AND GEORGE, 2008].

4.1.1 Computational Geometry Basics

A triangle is a three sided polygon. One of the well known and basic geometry figures in computational geometry. This is defined by list of three vertices (denoted by P_i in equation 4.1) as shown in figure 4.1 (a) which are generally oriented as default in a counter-clockwise orientation by computational geometry community [SHEWCHUK, 1996].

$$T = (p_1, p_2, p_3) \tag{4.1}$$

The surface area (S_T) of a triangle is one of the useful properties. It is denoted with

$$S_T = \frac{1}{2}\begin{vmatrix} x_2\text{-}x_1 & x_3\text{-}x_1 \\ y_2\text{-}y_1 & y_3\text{-}y_1 \end{vmatrix} \text{ or } S_T = \frac{1}{2}\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \tag{4.2}$$

Where x_i, y_i represent the coordinates of the vertices P_i (i=1, 2, 3) and |.| stands for the determinant. It usually considered with positive or negative sign with respect to the counter-clockwise or clockwise orientation respectively.

There are many ways to determine the circum-center and circum-radius of a given triangle. As an example, in the bisector method the perpendicular bisectors of the triangle edges intersect

at the circum-center. From this intersection, the circum-center can be determined. The circum-radius can be obtained by calculating the distance between circum-center and one of the vertices.

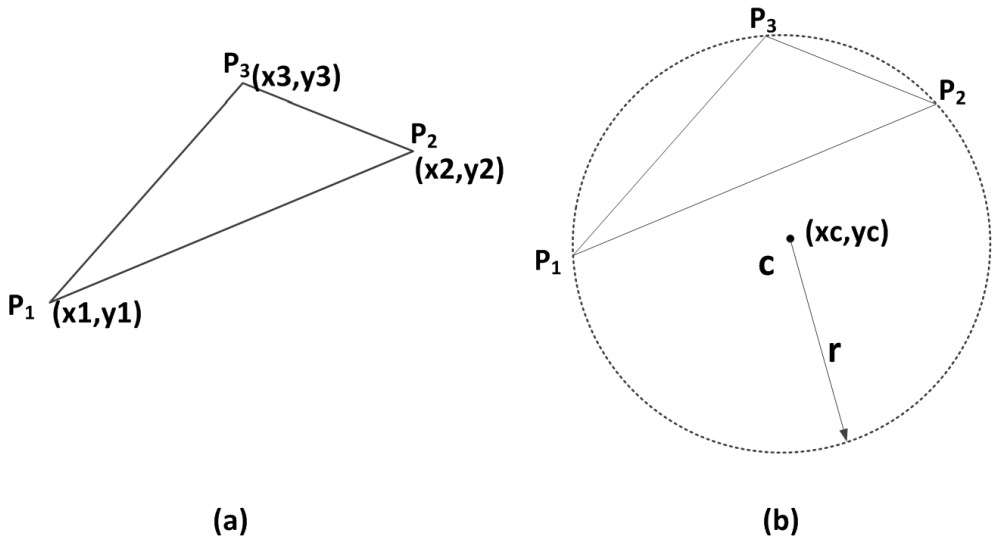

Figure 4-1 (a) Triangle, **(b)** Circumcircle of a triangle

$$\begin{pmatrix} x_3\text{-}x_1 & y_3\text{-}y_1 \\ x_3\text{-}x_2 & y_3\text{-}y_2 \end{pmatrix} \begin{pmatrix} x_c \\ y_c \end{pmatrix} = \frac{1}{2} \begin{pmatrix} (x_3^2+y_3^2) & -\ (x_1^2+y_1^2) \\ (x_3^2+y_3^2) & -\ (x_2^2+y_2^2) \end{pmatrix} \quad 4.3$$

By solving the above linear system equation 4.3 one can get the circum-center. The circum-radius r_T can be obtained by using the formula in the equation 4.4 without calculating circum-center. The lengths l_a, l_b, l_c are the lengths of edges which are opposite to vertices a, b, c respectively.

$$r_T = \frac{l_a l_b l_c}{2S_T} \quad 4.4$$

4.1.1.1 Orientation Test

In computational geometry algorithms, the numerical tests such as orientation and in-circle test of vertices are frequently utilized. In the orientation test as shown in figure 4.2, the third vertex c is tested whether it is located on the right side or left side of the line vector $\overrightarrow{ab}$ or on that line $\overrightarrow{ab}$. The value of the determinant shown in the equation 4.5 is checked for the sign to decide the side of the third vertex. According to the determinant value, which is positive, negative or equal to zero, then the vertex c is located left, right or on the line $\overrightarrow{ab}$ respectively.

$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = \begin{vmatrix} a_x\text{-}c_x & a_y\text{-}c_y \\ b_x\text{-}c_x & b_y\text{-}c_y \end{vmatrix} \quad 4.5$$

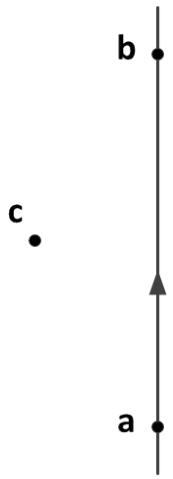

Figure 4-2 Orientation of a vertex with respect to a given line

4.1.1.2 In-circle Test

As shown in the figure 4.3, the in-circle test is performed to check the fourth vertex d lies whether inside the circle which passing through vertices a, b, and c or not. If the value of the determinant in the equation 4.6 is positive, then the vertex d lies inside the circumcircle of a, b and c.

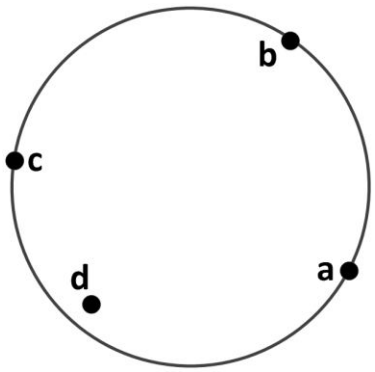

Figure 4-3 In-circle-property of four non-linear points

$$\begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{vmatrix} \quad 4.6$$

These numerical tests are performed by calculating the determinant using coordinate values of vertices. By testing values of determinant, these tests are decided. The numerical accuracy in calculating the values of determinants plays a major role in test results. As mentioned in the arithmetic robust predicates work [SHEWCHUK, 1996], the round-off error plays a significant role and which can lead an incorrect results when the actual determinant value is close to zero. This can influence on the accuracy of the numerical algorithms. The precision of the calculated coordinate values also depends on the computing using single or double floating-point computation. To solve this numerical issue Shewchuk has introduced a robust numerical algorithm which is reported that it produces a correct and faster performance when compared with traditional arithmetic libraries.

4.1.1.3 Triangulation

According to the definitions described in [FREY AND GEORGE, 2008], the convex-hull, conformal and non-conformal triangulations are defined below. For a finite point-set S in $\mathbb{R}^d$ (d = 2 or d= 3) the convex-hull of S defines a domain Ω in $\mathbb{R}^d$ and which is denoted as $Conv(S)$. If a simplex in the domain Ω denoted as K (triangle or tetrahedron according to d), the simplicial covering-up of domain Ω is $\mathcal{T}_r$ and $\mathcal{T}_r$ is defined by mean of simplexes according to the following definitions.

- (H_0) The set of element vertices in $\mathcal{T}_r$ is exactly S
- (H_1) $\Omega = \bigcup_{K \epsilon \mathcal{T}_r} K$, where K is a simplex
- (H_2) The interior of every element K in $\mathcal{T}_r$ is non empty
- (H_3) The intersection of the interior of two elements is an empty set
- (H_4) the intersection of two elements in $\mathcal{T}_r$ is either reduced to

  - the empty set or

  - a vertex, an edge or a face (for d =3)

A conforming triangulation or simply a triangulation of a finite point-set S is a conforming covering-up. To satisfy this conforming property another condition (H_4) should be satisfied.

In figure 4.4 (a) a conforming triangulation of a convex-hull domain is given. When a triangulation does not satisfy the 4th condition (H_4) , then that triangulation is called as non-conforming triangulation as shown in figure 4.4 (b). The hanging nodes are commonly formed in a non-conforming triangulation. In a constrained triangulation of a convex-hull, the provided edges or faces (in d =3) must be contained in the triangulation as it is. As shown in figure 4.4 (c), the given constraint edge should not be intersected and exists as it is in the triangulation.

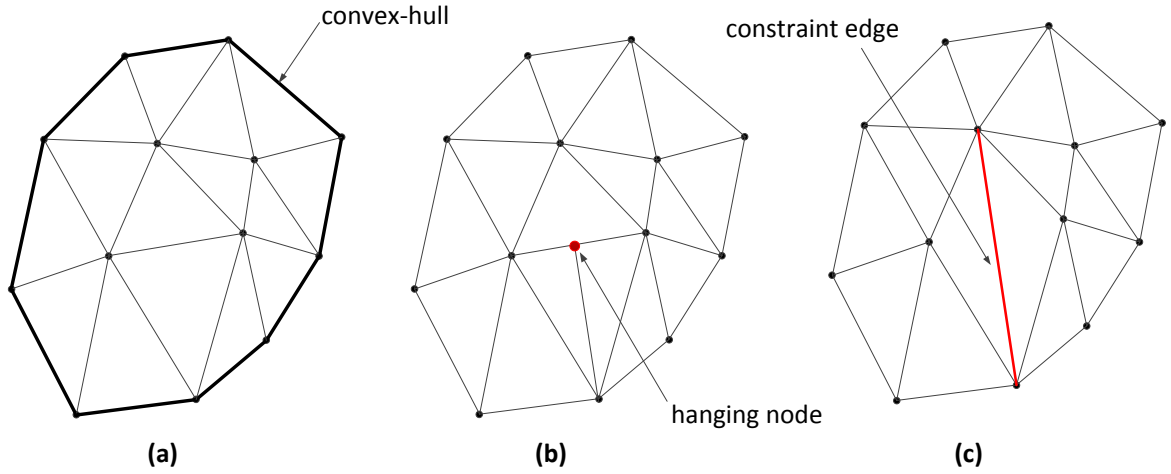

Figure 4-4 2D triangulation **(a)** conforming triangulation, **(b)** non-conforming triangulation, **(c)** constraint triangulation

4.1.2 Voronoi Tessellation

The Voronoi tessellation is a one of the basic representation in computational geometry. This represents a given point-set $S = \{x_1, x_2, \dots, x_n\}$ in a plane domain Ω into a combination of n subdivided polygonal regions. Each sub region represents the closest region to that point than any others points. In simple word, any point within a Voronoi cell belongs to point x_i is closest to x_i than any other points. As shown in figure 4.5 (a), a Voronoi representation of a 2 D point-set is shown as colored regions of each input points. Boundary regions are infinitive because those Voronoi regions have their boundaries outside the domain Ω. The line segments shared by two Voronoi cells are called Voronoi edges. Any point on these edges is in equal distance from the points which belongs to neighbor Voronoi cells. When a point is shared by three or more Voronoi cells then it is called as Voronoi vertex.

Mathematical representation of a Voronoi diagram of a point-set S is given as shown in the equation 4.7. Voronoi cell is denoted by $R(S; x_i)$ for a point $x_i \epsilon S$.

$$R(S;x_i)=\{x\epsilon\ \Omega : \|x-x_i\|< \|x-x_j\|, x_j \epsilon\ \Omega, x_j \neq x_i\} \quad 4.7$$

Where $\|x_i-x_j\|$ is used to denote the Euclidean distance between two points x_i and x_j. The region Ω is partitioned into regions $R(S;x_1)$, $R(S;x_2)$, …, $R(S;x_n)$ and the boundaries. These partitions represent the Voronoi diagram of S and denoted by $V(S)$.

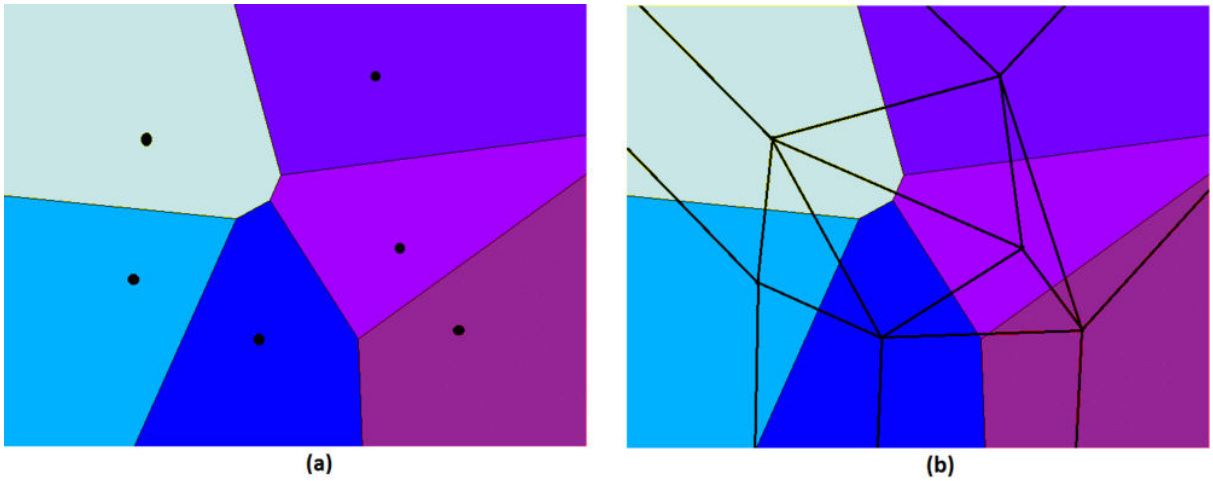

Figure 4-5 (a) A 2D Voronoi diagram of given point-set, **(b)** Corresponding Delaunay diagram of that point-set

4.1.3 Delaunay and Constraint Delaunay Triangulation

This widely used triangulation technique starts its history after a pioneering paper published by Boris Delaunay in 1934 [FREY AND GEORGE, 2008]. From this time onwards, the application of this triangulation technique and different algorithms to construct this triangulation have been continuously developed by numerous researchers. In the application of engineering problems, Delaunay triangulation is the most preferred algorithm due to several reasons. Some of the main reasons are its theoretical property to investigate numerical issues analytically, efficient construction techniques and flexibility in mesh generation [TOPPING AND KHAN, 1996]. The quality properties of triangulation are mostly realized by good shape of triangles, which are checked for smallest edge over radius ratio or smallest or largest angle criteria. The major Delaunay triangulation techniques are incremental insertion, incremental construction, divide and conquer, advancing front technique, local improvement and projection based methods [CIGNONI ET AL., 1993].

According to the definition of Voronoi diagram, each Voronoi cell $R(S;x_i)$ is non-empty and associated with one point x_i in S. According to the Delaunay's finding, by connecting the

adjacent points x_i of Voronoi cell a triangulation is produced. This dual representation is the desired Delaunay diagram of that given point-set S (as shown in figure 4.5 (b)). In general the construction of a triangulation of a given point-set P is achieved by a triangulation of the convex-hull of that point-set. This can be easily achieved by using this dual transformation from Voronoi to Delaunay representation [TOPPING AND KHAN, 1996].

In a triangulation and a mesh generation process, in-circle test is one of the frequently used properties. The empty circle or empty sphere criterion is a fundamental property of a Delaunay triangulation. This is also called Delaunay criterion. The circumcircle passing through three points in two dimensions or a circumsphere in three dimensions, if it is not contain any fourth point inside, then it is called empty circle or empty sphere respectively.

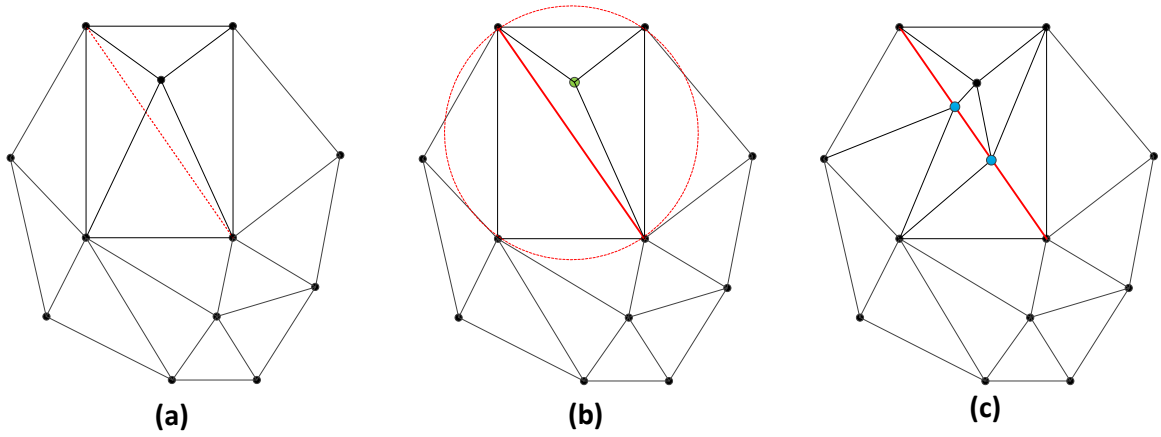

Figure 4.6 (a) Delaunay triangulation without considering constraint edge, **(b)** constraint Delaunay triangulation by considering a constraint edge, **(c)** conforming constraint triangulation by refining that constraint edge

The general lemma of Delaunay triangulation says that a triangulation T of a given convex-hull of point-set S is Delaunay triangulation, if each and every pair of adjacent triangles holds the empty circle property globally [GEORGE AND BOROUCHAKI, 1998]. The proof of this fundamental lemma and other Delaunay triangulation theorem one can find in any common computational geometry books.

As shown in figure 4.6, a constraint Delaunay triangulation is explained by introducing a constraint edge (red dash line) into a Delaunay triangulation. As shown in the figure 4.6 (a) this triangulation is already a Delaunay triangulation. If the given constraint edge should be included into the triangulation as shown in figure 4.6 (b), then the Delaunay property is no more valid to few triangles. The vertex (green color) is not satisfying the empty circle

property. In this case, the produced triangulation is not globally Delaunay and this triangulation is called a constraint Delaunay triangulation. By permitting the constraint edges to be refined with additional vertices (blue color) (as shown in figure 4.6 (c)), then a valid Delaunay triangulation can be formed [SHEWCHUK, 1996]. This type of triangulation is called conforming Delaunay triangulation.

4.1.4 Algorithms and Data Structures

For a triangulation and mesh generation process, selection of suitable algorithms and appropriate data structures are vital process. In a triangulation process, various computational geometry algorithms are involved. The paradigm experienced by [WIRTH, 1986] is that a triangulation or meshing program is a combination of algorithms and data-structures which are closely linked to each other. On the one hand a usage of complex and large data-structure makes the computational geometry algorithm simple. On the other hand the selection of simple data structure increases the complexity to program the algorithm.

There are many Delaunay triangulation algorithms available. Some of them are divide & conquer, plane sweeping, incremental insertion (Bowyer-Watson), incremental construction, advancing front and quad-tree etc. The details of these algorithms are available in majority of the computational geometry books [FREY AND GEORGE, 2008, BERG ET AL., 2008].

It is meaningful to indicate here some of the significant computational geometry algorithms which are partially used in the Delaunay triangulation process of this thesis. First of all there are different sorting algorithms available and commonly used to sort the point coordinates to partition the triangulation domain or group the points as different regions. Another frequently used algorithm is locating a vertex within a triangulation. The most commonly used algorithm is walking-algorithm to find a triangle in which a given point is located [BERG ET AL., 2008]. For the point insertion in a triangulation process there are various approaches used. Many of the algorithms are used centroid, circumcenter of the triangle or on the longest edge of the triangle.

In the selection of a suitable algorithm and optimum data structures, usually a programmer should approach with an intelligent choice. Particularly in the triangulation problem, the final triangulation or mesh contains the list of triangles which are defined using three vertex indices and the three neighbor triangle indices of each triangle. This is the most common output of any triangulation or meshing program. But for the computation of triangulation during the runtime, there many intermediate data are necessary. If a sophisticated data structure is

handled by keeping the records of all intermediate information, this will make the algorithm simple, but each and every change in data should be updated among all the data structures. This will consumes large part of the computation time. At the same time for each and every intermediate data, if the algorithm is supposed to calculate again and again, then this will also increase the total computation time. In general increasing the data size in the random access memory (RAM) during runtime is not preferred. Deciding for a large and sophisticated data structure is also not intelligent choice. Therefore the selection of data structure to a particular algorithm is always done in an optimum way by compromising both the aspects.

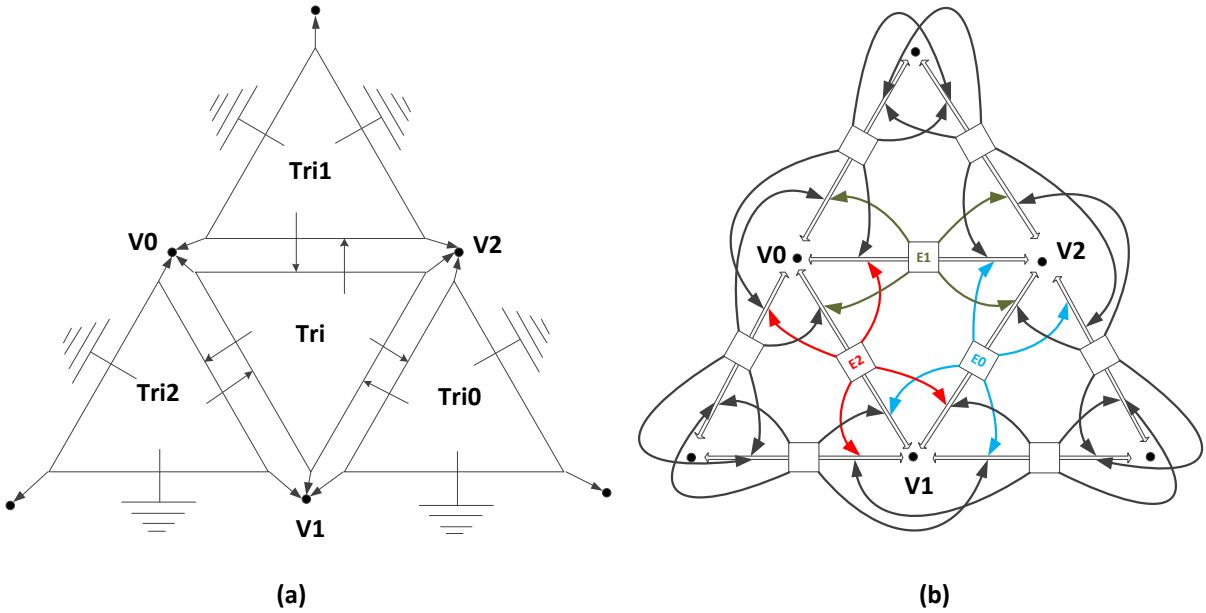

Figure 4-6 (a) Triangle based data structure used in *Triangle* implementation [SHEWCHUK, 1996] **(b)** quad-edge based data structure

In the implementation of this work, the data structure used during the triangulation is similar to the data structures used in the GPU-DT implementation. The data structure used in GPU-DT implementation is a triangle based data structure (as shown in figure 4.7 (a)) which is the same data structure as used in Shewchuk's Triangle implementation. The triangle based data structure is originally used in the implementation of *Triangle* by [SHEWCHUK, 1996]. The original *Triangle* implementation was implemented using a quad-edge base data structure. The quad-edge data structure was introduced by [GUIBAS AND STOLFI, 1985] with six pointers to represent four neighbor edges and two end vertices of an edge of a triangle. As shown in figure 4.7 (b) the four neighbor relations of edges E0, E1, E2 are marked with blue, green and red colors respectively. This data structures is attracted by computational geometers due to its properties such as elegant usage and rich of detail algorithm which easily allows for different

triangulation implementation (divide-and-conquer and incremental Delaunay triangulations). With this type of data structure, representing graph and its dual relations is simple as example Delaunay triangulation and its corresponding Voronoi diagram [SHEWCHUK, 1996].

As shown in figure 4.7, the triangle data structure also uses six pointers to represent each triangle record. Three pointers represent the triangle vertices and another three pointers point to the three neighbor triangles. According to the reported experience by Shewchuck the divide-and-conquer, the incremental and Rupperts's Delaunay refinement algorithms are realized almost double the speed up using triangle data structure [SHEWCHUK, 1996]. However the implementation of the same algorithms devoted with more effort and used roughly twice length of codes for triangle data structure. In general a triangulation contains roughly three edges for every two triangles. This fact indicates that *Triangle* base data structure is more efficient than quad-edge based data structure.

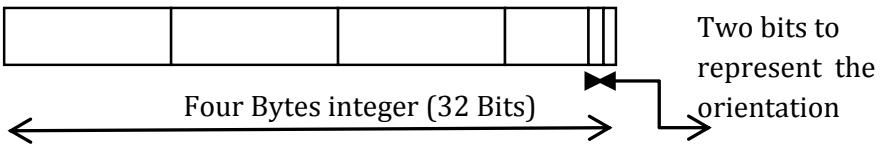

Figure 4-7 Detail of oriented-triangle data structure

These data structures are used not only to represent the neighbor triangle relations but also the orientation of the connected triangles. Each pointer to the neighbor triangle is represented by an integer number. This number contains the number of the triangle index and the associated orientation of the edge. The orientation can be represented by a number between zero and two. This can be represented using only two bits of data. The memory usage is optimized by only using four-bytes (32 bits) of integer to represent one neighbor triangle. Out of these 32 bits last two bits are used to represent the orientation and remaining 30 bits integer represent the neighbor triangle index as shown in figure 4.8. By storing the both information within a single pointer, the memory traffic can be reduced. Since the memory traffic is more expensive than bit operations [SHEWCHUK, 1996], by using this single data record the memory space as well as operation speed can be optimized.

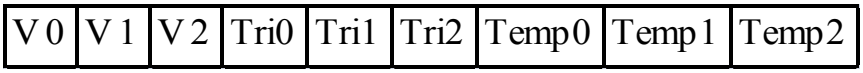

Figure 4-8 Triangle data structure used in GPU-DT and this implementation

In the GPU-DT implementation, in addition to triangle based data structure, there are three more pointers added for temporary usage during the runtime process. As shown in figure 4.9, first three pointers (V0,V1,V2) represent triangle vertex indices, the next three pointers (Tri0, Tri1, Tri2) represent the neighbor triangles with orientation and the last three pointers (Temp0, Temp1, Temp2) used for temporary purpose.

4.1.5 Unstructured Delaunay Mesh Generator

In general unstructured mesh generators use automated discretizing process of a given mesh geometry domain using simplexes (triangle, tetrahedron). This process involves with creation of vertices and establishing connectivity between them. This is performed through different stages such as domain boundary definition, specification of element size distribution, triangulation, boundary fixing and finally mesh optimization by establishing quality elements everywhere in the mesh domain.

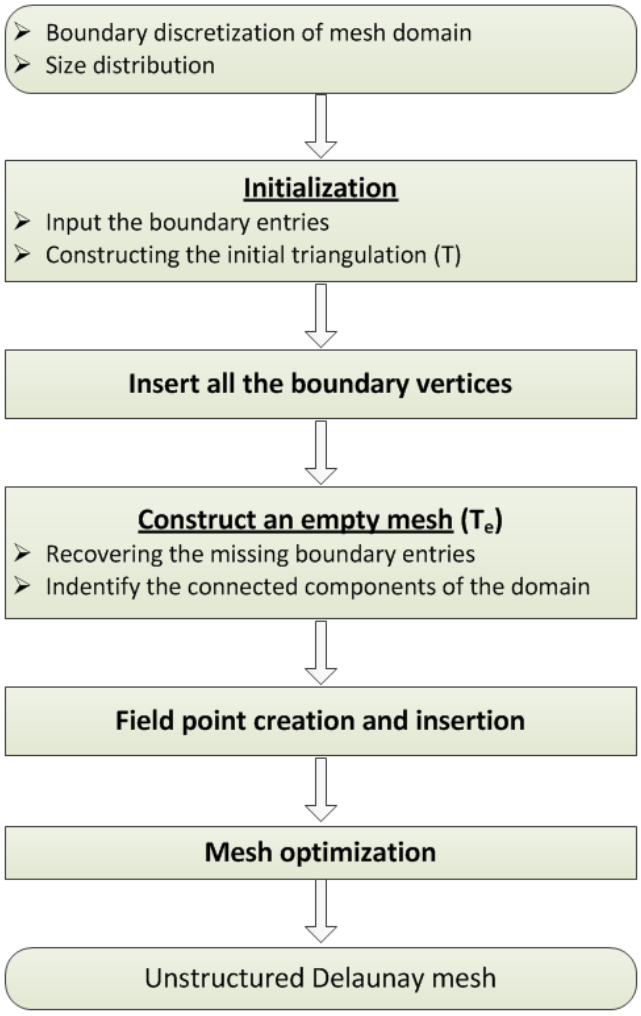

Figure 4-9 Flow chart of process in a general Delaunay mesh generator

According to the reference [FREY AND GEORGE, 2008], a general scheme of Delaunay unstructured mesh generation process is given as a flow chart in figure 4.10. The mesh generator starts the process with the provided information such as domain boundary discritization and mesh element size distribution. During the initialization phase, the initial triangulation is produced according to the boundary details. Then all the boundary vertices are inserted into the triangulation. Then by using all the boundary components an empty mesh of the complete domain is produced. After that, all the field points are generated and inserted into the domain using any algorithm. According to [FREY AND GEORGE, 2008], most of the mesh generation procedures incorporate as a point insertion algorithm using either the Bowyer-Watson's algorithm [BOWYER, 1981 AND WATSON, 1981] or the Green-Sibson's algorithm [GREEN AND SIBSON, 1980]. As a final process mesh is optimized by considering the mesh element quality criteria.

4.2 GPU-DT Algorithm Overview

4.2.1 Discrete Voronoi Diagram

The discrete Voronoi diagram is a similar representation of Voronoi diagram, in which integer grid points (discrete space) are used instead of continuous space coordinates. As shown in figure 4.11 (a), the given domain is defined by grids of square cells. Point-set are given by integer grid coordinate values. If a grid point A is located inside a Voronoi region E_i of point x_i, then A is colored by x_i. In the case that a grid point A lies in an equal distance from two points x_i and x_j and $i < j$ then A is colored by x_i. In this way each and every grid points represent a color of their Voronoi regions. These colored regions are called discrete Voronoi diagram $D(S)$ of S (as shown in figure 4.11 (b)) [RONG ET AL., 2008].

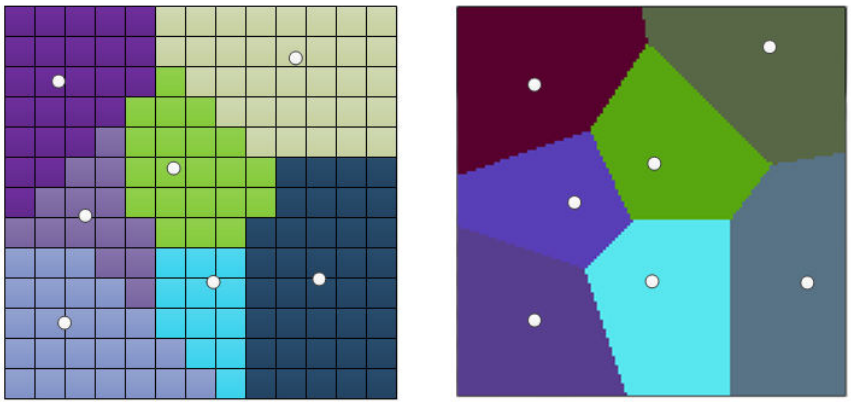

Figure 4-10 (a) Euclidean coloring of a given point-set, **(b)** Discrete Voronoi diagram

4.2.2 Parallel Banding Algorithm (PBA)

For a given set of sites S on a; $N = n\,x\,n$ grid pixels domain, the distance detail of each pixels to the nearest site is called distance transform. The distance transform relation is closely related to discrete Voronoi diagram of that point-set. When each site is given different color value, then coloring the closest region with respective color of that site produces the discrete Voronoi diagram. The Euclidean Distance Transform (EDT) is the 2D representation of binary image distance transform. The computing from distance transform is straight forward to discrete Voronoi diagram.

The parallel banding algorithm is a GPU based efficient algorithm to compute the EDT. This is the first GPU based parallel algorithm to compute the discrete Voronoi diagram [CAO ET AL., 2010]. This algorithm utilizes the GPU parallel capability by partitioning the rectangular input pixels information into horizontal and vertical bands for concurrent computation. At the end the results from different bands are merged into final discrete Voronoi diagram. GPU implementation of this algorithm is done using CUDA. According to the [CAO ET AL., 2010] report, this algorithm realizes better performance than other GPU based approximate EDT implementations.

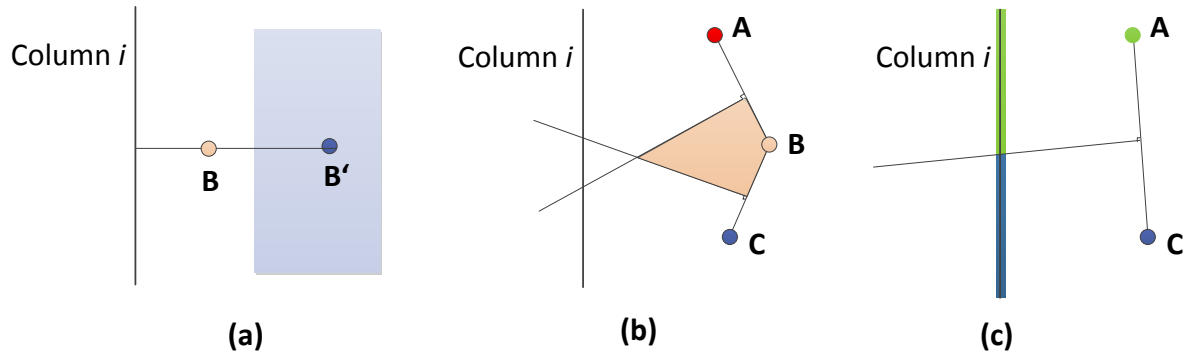

Figure 4-11 (a) Nearest Voronoi region (fact 1), **(b)** Dominating Voronoi regions (fact 2), **(c)** Deciding the proximate sites (fact 3)

In computation of parallel banding algorithm to determine the EDT, there are three straightforward facts utilized, which improve the computation efficiency. To compute the discrete Voronoi diagram of column i of a 2D grid, the intersection of that column with the Voronoi diagram is computed. As shown in figure 4.12 (a), for the two sites B and B' in a row, the first fact states that if B is closer to the column i, then Voronoi region of B' cannot intersect the column i. This implies that to compute the discrete Voronoi diagram of column i, only one site (closest to that column) per row have to be considered.

As shown in figure 4.12 (b), for the three considered sites A, B, C from top to bottom, if the perpendicular line of AB intersects the column i below than the perpendicular of BC, according to the second fact, the Voronoi region of B cannot intersect the column i. This fact implies that the Voronoi region of A and C dominates the Voronoi region of B. That means, the Voronoi region of B is dominated by A and C. According to the third fact as shown in figure 4.12 (c) if A is on top of C then the Voronoi region of A is also on top of the Voronoi region of C.

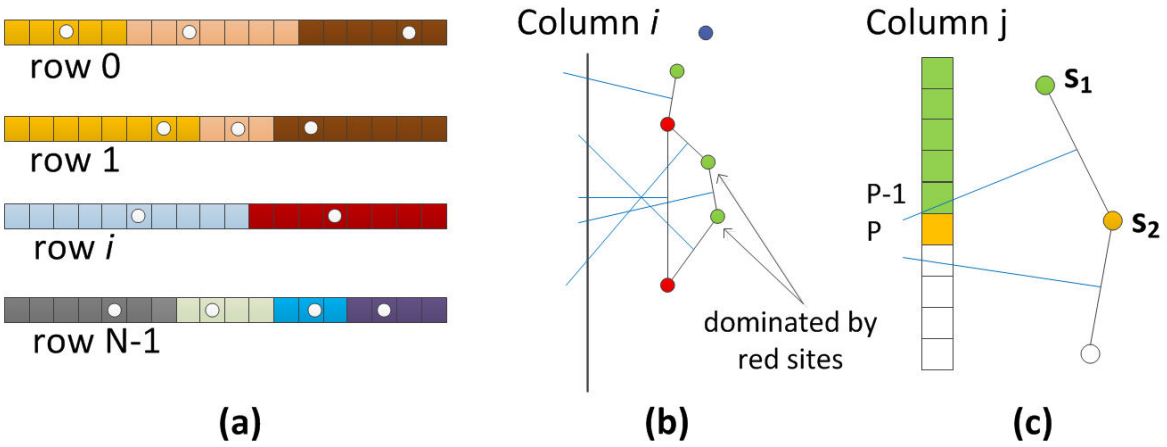

Figure 4-12 (a) Find the nearest site of each pixel in a row (phase 1), **(b)** Determine the proximate sites of each column (phase 2), **(c)** Color the columns using proximate sites (phase 3)

In the PBA algorithm to determine the discrete Voronoi diagram, [CAO ET AL., 2010] uses three phases of computation using the above three facts. As shown in figure 4.13 (a) using the fact 1, the phase 1 computes for the nearest site of each pixel along the row. To determine the nearest sites, sweeping is done along the row from left to right and then right to left. In phase 2, by using the fact 2 and the result of phase 1, the set of proximate sites of each column can be computed. The proximate sites of a particular column are the sites, which decide the Voronoi diagram of that column. As shown in figure 4.13 (b) the proximate sites of each column is computed from top to bottom and stored in a stack. As an example, the marked green color sites are dominated by the red color sites and they are not included in the proximate sites of column i. In the final phase (phase 3), each column is colored according to the proximate sites information as shown in figure 4.13 (c). At the end of phase 3, by merging the color information of all columns the discrete Voronoi diagram of that point-set is produced.

It should be noted that the computation in phase 1 is independent between different rows. This process can be parallelized by computing each row independently. Similarly in the computation of phases 2 and 3, each column can be independently computed in parallel. This parallelization possibility enables to parallelize this PBA algorithm using CUDA parallel programming. In the CUDA implementation, many vertical and horizontal bands are used to increase the number of CUDA parallel threads to exploit the maximum parallel performance of GPU. Further detail of this algorithm and implementation detail can be found in the reference [CAO ET AL., 2010].

4.2.3 Jump Flooding Algorithm (JFA)

The jump flooding algorithm is an improved flooding method than standard flooding and used in the applications of distance transform and digital Voronoi diagram. Parallel implementation using GPUs is available. Flooding can be defined in this context as a process of populating information of a given grid point to other neighbor grids. Since the JFA algorithm uses the approximation in the calculation of distance, there is a possibility for errors in Voronoi diagrams. Different variants of JFA were introduced by the same authors to reduce this type of approximation errors [RONG AND TAN, 2007].

The basic idea of flooding the content (position information) of seed points (given points) (s) is to identify closet neighbor for each grid point. In computing a Voronoi diagram of 2D point-set in $N\ x\ N$ size of grid, given seeds points are located on few grid points. The contents of s are populated to each grid points. Then each grid point can determine which seed point is its closest one. As shown in figure 4.14, in a standard flooding the content of a seed s is flooded outward in increasing distance. In the first round of flooding, site information is passed to the neighbor grid points (maximum eight). From the second round onwards, the direct neighbor grids get the values from the previously populated neighbors. These flooding longs for number of N rounds till all the grids received the seed information.

The main idea of JFA is that the content of seed points can be populated in logarithmic steps instead of the constant number of steps in standard flooding method. The neighboring seed point contents are kept in grid points. That means each grid point contains the distance detail of closest seed point. At the start of the algorithm all grid points and also grid points which belong to seed points are initialized with information. The algorithm is run in multiple rounds ($log_2(N)$) to flood a given grid information to all other grids.

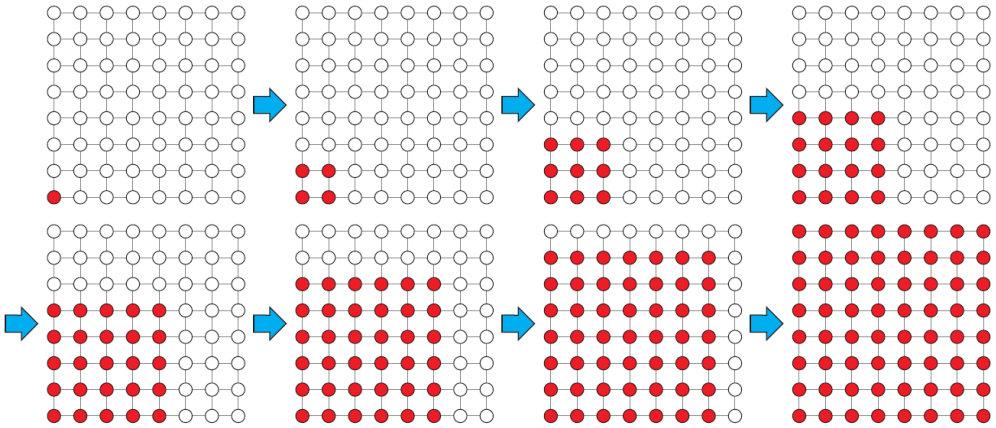

Figure 4-13 The standard flooding algorithm (based on [Rong et al., 2008])

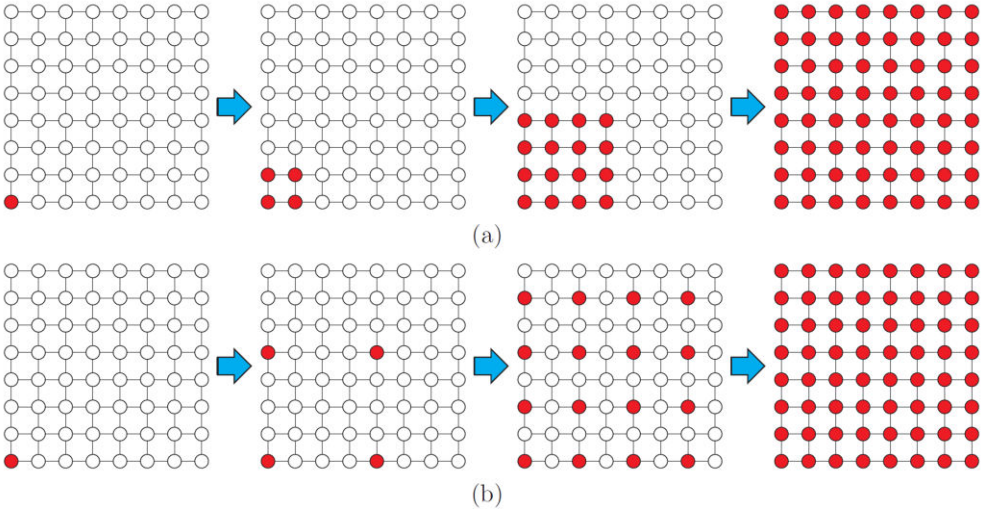

Figure 4-14 Jump flooding algorithm; **(a)** doubling-step size, **(b)** halving-step size (based on [Rong et al., 2008])

In each round, each grid point (x, y) populates its value to maximum eight neighbor grid points $(x + i, y + j)$, here $i, j \epsilon (-k, 0, k)$. The step size k is varied after each round to flood the values to different neighbor grids in each round. In this algorithm, the change of step size in each round can be performed in two different ways. The first type is doubling the step size in each round by starting from $k = 1$. The other way of step size variation is that starting from $k = N/2$ and then in each round k is divided by two. In the doubling step of JFA algorithm, the step size k in the round r is given by $\frac{N}{2^r}$. In the halving step size method is given by 2^{r-1}.

As shown in the figure 4.15 (a) information of one seed point in (0, 0) is populated to all grid points by starting from $k = 1$ then k = 2 and $k = 4$ in 3 rounds. In this example $N = 8$ and

number of rounds are $log_2(N) = 3$. In figure 4.15 (b), halving step size starts from $k = 4$, and then k takes 2 and 1.This completes the flooding of all grid points in 3 rounds.

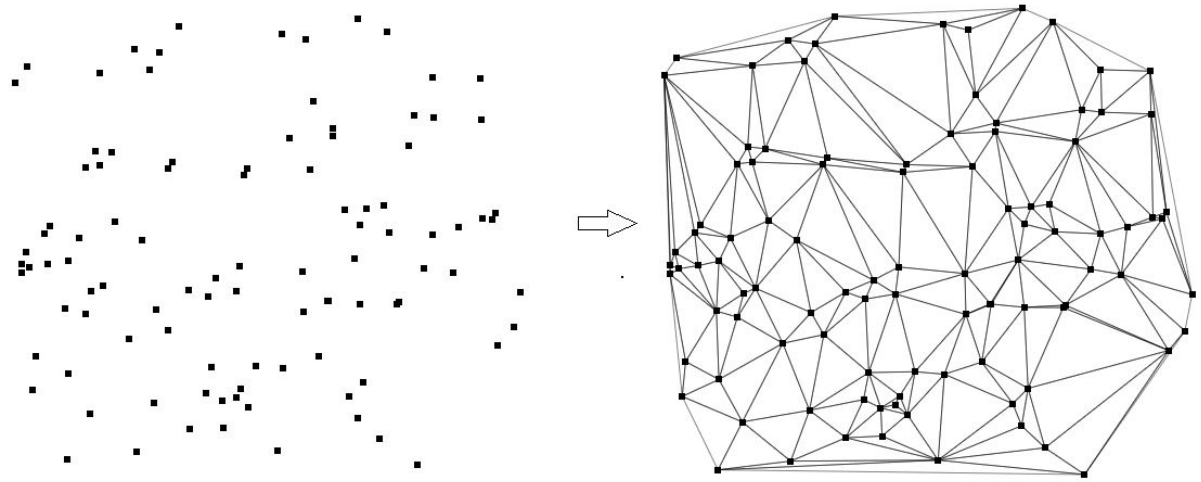

Figure 4-15 GPU-DT triangulation of random point-set

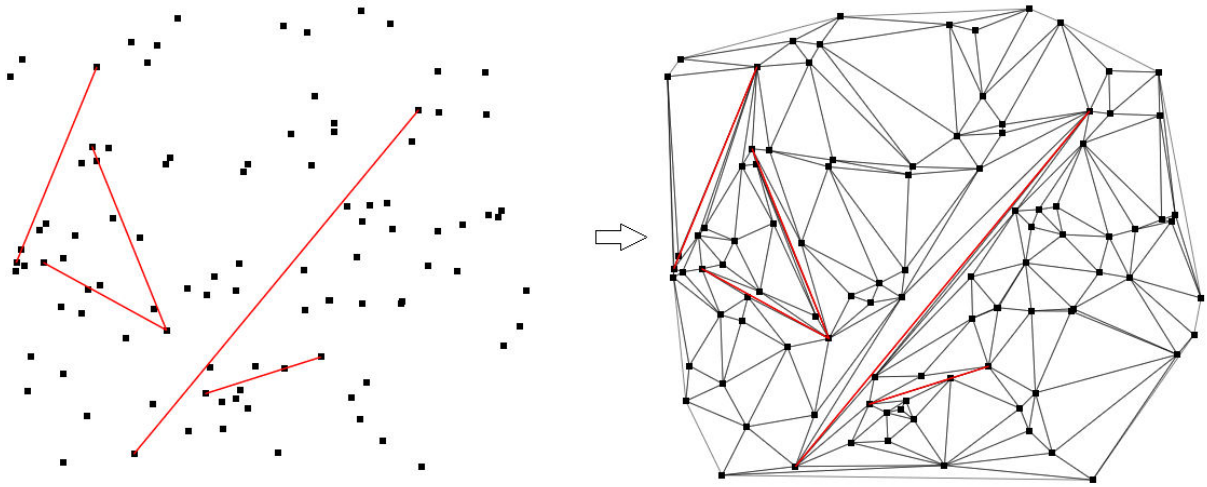

Figure 4-16 GPU-CDT triangulation of typical random point-set with randomly inserted constraint edges

4.2.4 Details of GPU-DT Algorithm

The parallel Delaunay triangulation using GPU is a very recent approach among. GPU-DT is an efficient GPU-based parallel Delaunay triangulation algorithm. There is one previous work has been presented the concept to build the Delaunay triangulation $T(S)$ from digital Voronoi diagram D (S) using graphic hardware [Hoff et al., 1999]. Any implementation detail of this

work was not reported. It is straightforward that deriving $T(S)$ directly from Voronoi diagram $V(S)$ is quite simple, but getting from $D(S)$ is realized by [RONG ET AL., 2008] as a non-trivial task. In this implementation the existing challenges are overcome to derive the Delaunay triangulation from $D(S)$ for a given two dimensional point-set S in R^2 using the computing power of GPU and CPU. According to the reported results from original authors and from our preliminary test results, the performance of GPU-DT for a large number of 2D point-sets is better than the CPU-based best sequential Delaunay triangulator (*Triangle*). There are still limitations in this method, because of the limited memory space in GPUs.

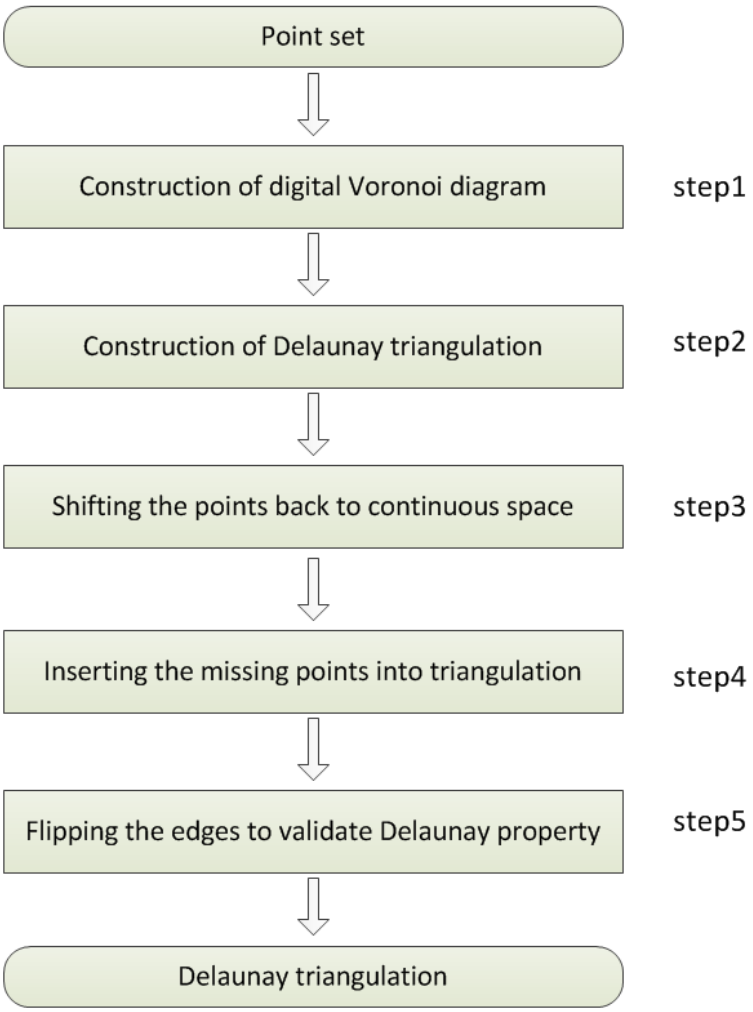

Figure 4-17 Flow chart of GPU-DT algorithm

Algorithmic details of GPU-DT are described based on [RONG ET AL., 2008, QI ET AL., 2013, CAO, 2009] in following sections. The more recent release of this implementation from research group is GPU-CDT. This handles for a Delaunay triangulation of a planar straight line graph (PSLG) which contains 2D point-sets and random constraint edges. A typical

triangulation of a 2D point-set using GPU-DT implementation appears like in figure 4.16. When a point-set is included with random constraint edges the triangulation result using the GPU-CDT appears like in figure 4.17. A detail description of GPU-DT algorithm is presented according to above mentioned references. Construction of Delaunay triangulation using GPU-DT is computed using the five main steps as shown in the flow chart in figure 4.18.

4.2.4.1 Construction of Digital Voronoi Diagram (Step1)

Continuous point space is scaled into a digital texture space in such a way that all the given points (sites) fit within the texture boundaries. A texture space is defined by 2D integer grids of $n \, x \, n$ cells. Here n is the decided texture resolution in one dimension. The maximum texture resolution (or texture size) available in currently available graphic cards can go up to 8192x8192.

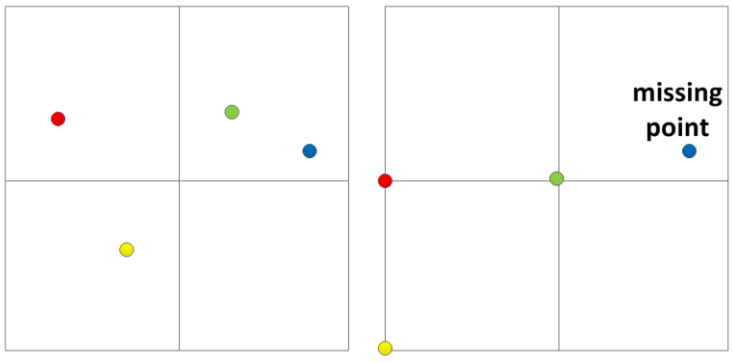

Figure 4-18 (a) Multiple points fall in a grid cell, **(b)** After mapping, missing-point remains

As shown in figure 4.19 (a) and (b) during mapping process each point is mapped into the nearest lower left grid coordinate. If there is more than one point falls within a grid cell then only one point is mapped the grid and rest of the points are considered as missing points and inserted during the process step 4. The quantity of missing points depends on the density of the point distribution and the allocated texture resolution.

There are different methods available to flood the seed points on the texture and using that to derive the digital Voronoi diagram. The standard flooding technique is preferable for its error free nature but it is slower in performance. This realizes a time complexity of $\sqrt{2}n$. The different variants of Jump flooding algorithm is relatively faster with the time complexity of $\log n$. The non-impressive issue with JFA is that sometimes it does not produce the exact digital Voronoi diagram. According to [RONG ET AL., 2008], it should be noted that the chances to affect the triangle topology in the final triangulation using JFA are less. In the

currently available GPU-DT implementation, the Parallel Banding Algorithm (PBA) [CAO ET AL., 2010] is incorporated to compute the digital Voronoi diagram. The PBA algorithm is much faster than standard flooding algorithm and producing error free Voronoi diagram.

4.2.4.2 Construction of Triangulation (step 2)

In this step, using the dual property of Voronoi and Delaunay diagram, the Delaunay triangulation is determined. The digital Voronoi diagram produced from previous step is used to determine the Voronoi vertices in each row of the texture. As shown in figure 4.20, each pixel has four grid corners. Each grid corners shared with four neighbor pixels except the corners in the texture boundaries. Each corner is incident of different colors from one to four. If a corners is surrounded by three or four different colors then this grid corner is called digital Voronoi vertex (pixels with Yes). There is an exceptional case that three different colors share a corner at the same time the diagonal pixels hold the same color. The corners shown in figure 4.20 (b) (pixels with No) are not valid Voronoi vertices.

When a Voronoi vertex is shred with three different colors which belongs to three different sites, there is one triangle is added into the triangulation by connecting these three sites. In the case that Voronoi vertex shared with four different colors then two triangles are formed by connecting those four sites. After the triangles and topology are identified, the points are re-mapped to the original coordinate values. As opposite to the step 1, point coordinates are scaled reversely back to original coordinates without affecting the triangulation topology. This remapping is relatively trivial process, but it is very numerical sensitive, and should be handled cautiously to avoid invalid triangulation.

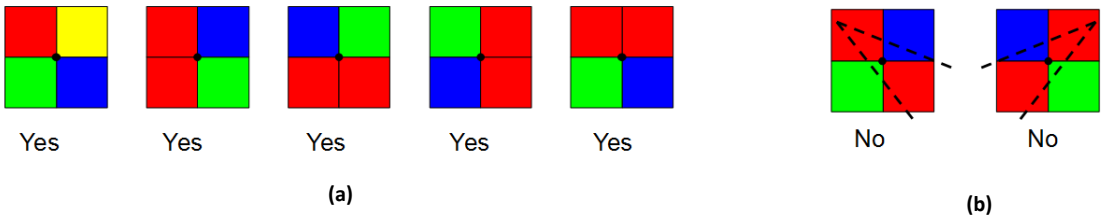

Figure 4-19 (a) Valid Voronoi vertices, **(b)** Non-valid Voronoi vertices (based on [RONG ET AL., 2008])

The drawback with the digital Voronoi diagram is that the Voronoi regions at the texture boundaries are truncated. When the triangulation is formed using dual property, the triangulation is not always covered with convex-hull boundary. This problem can be fixed by traversing along the texture boundary to identify the triangles which contains the Voronoi

vertices outside the texture boundary. This boundary traversing process is similar to Graham's boundary scan algorithm [GRAHAM, 1972]. This computation is performed on CPU level in parallel without affecting the GPU computation of identifying the triangles.

This triangulation is not included all the given points because of the missing points to be inserted at later step. The missing points which are inserted into the triangulation may locate on the convex-hull or outside the convex-hull. There might be frequent requirements to fix the convex boundary again and again during the insertion points into triangulation. This can be a troublesome process. To avoid this problem, the given point-set is bounded by a large enough convex polygon boundary (fake boundary) at the beginning. That means to make sure that any inserting and deleting points locates within the triangulation.

The fake boundary is established by deciding few additional points far enough from the given point-set S, to make sure to all the given points locate within the big boundary. As shown in figure 4.21 (a) a fake boundary is established using three additional fake points. Fake boundaries with less fake points have few disadvantages. This type is producing lot of sharp triangles which are not preferred because of causing topology conflict (bad cases) during the shifting. Let the number of points lie on the convex-hull is k. Then each of the three additional boundary points form triangles with averagely $k/3$ points. This makes the mesh manipulation slower due to the large number of triangles are walked over for each operation.

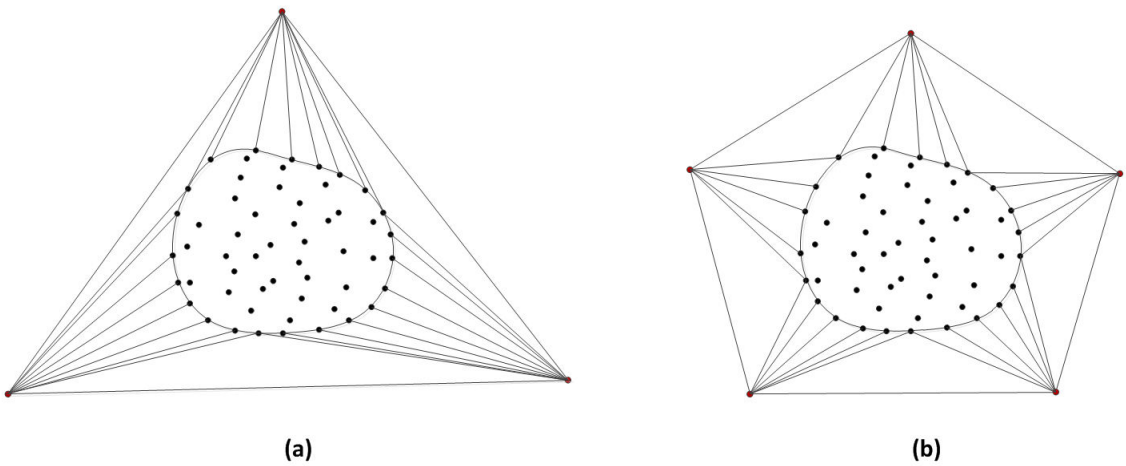

Figure 4-20 Fake boundary: **(a)** With three additional points, **(b)** More additional fake

A better approach to get rid of these sharp triangles incorporated in the GPU-DT implementation [RONG ET AL., 2008]. The number of additional boundary point is increased

to $\sqrt{k}$ as shown in figure 4.21 (b). This approach reduces the number of triangle belongs to each additional points. Also produces the relatively big triangles which are preferred to avoid the bad cases. After decide the triangulation, fake points are removed and convex-hull boundary is established.

4.2.4.3 Shifting (step 3)

In this step, the point-set mapped (step 1) into the texture is remapped to their original coordinate values. That means the point-set S' in digital space is moved to original continuous space point-set S. This process is called as shifting in the GPU-DT implementation. When approximate coordinate positions in the digital space are replaced by exact coordinate positions, there are possibilities that the triangle topology to be affected by forming intersecting edges. As shown in figure 4.22 (a), the triangle topology with solid edges are the existing triangulation and the dotted edges form the triangle after shifting the vertex p to p'. In this case, the shifting process affects the triangle topology and forms the intersecting triangles. In this algorithm, these cases are called bad cases and treated separately during the shifting process. In figure 4.22 (b), shifting the vertex p to p' is not affecting the topology (not forming any intersecting triangles). This case is called good case. In good cases the point coordinate values are simply updated to original values without updating the triangle structure. According to the reported experience of [QI ET AL., 2013], the good case is the ideal case and it occurs in the majority of the cases.

Detecting a point whether it belongs to good case or bad case can be done in parallel. This computation is possible in multiple rounds. In each round one thread takes care of one point. Each thread walks through all the triangles in the triangle fan of that point by checking the counterclockwise test whether shifting of that point makes any crossing of neighbor triangles or not. To avoid the parallel conflict, each thread continues the process unless the index of that point is smaller than the index of neighbor points. If then this point would be valid for that round. At the end, that point is decided to belongs into bad or good case, if a good case, then point is updated with real coordinate values.

To utilize the parallel performance of GPU in shifting step, good cases and bad cases are handled separately. First of all the good case point-sets are identified and shifted to original coordinates. Shifting of good case points is done in multiple rounds until all points are shifted. In each round of iteration, points are shifted in parallel. At the start, all the good case points are identified and marked as unchecked. In each round, every parallel thread takes the

responsible for an unchecked point S_i. Each thread confirms that the selected site is shiftable without affecting the neighbor points. When a point S_i is neighbor to point S_j and sharing a common edge, shifting these two points in a single round using two parallel threads is not possible. Unless there is no unchecked neighbor point S_j with smaller indices ($j < i$) then only S_i can be shifted in this round. Otherwise shifting S_i is postponed to next iteration. The processed points are immediately marked as checked.

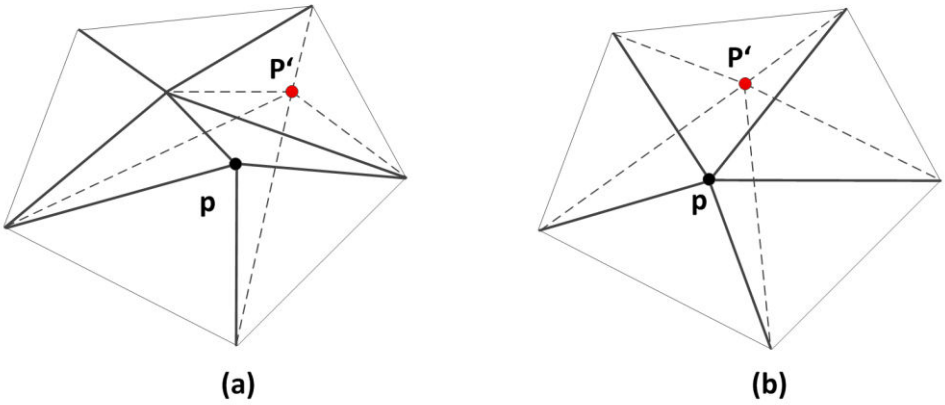

Figure 4-21 main two shifting cases: **(a)** bad case, **(b)** good case

4.2.4.4 Inserting Missing Sites (step 4)

Inserting a point into the triangulation is relatively simple when compared to computation of shifting process. Because of the effect of inserting a point in the triangulation is small. As shown in figure 4.23 there are three possibilities to locate a missing point into the triangulation. The simplest situation is that missing point falls within a triangle as shown in figure 4.23 (a). This insertion simply introduces three more triangles into the triangulation and the neighbor relations can be updated without much effort. The second simplest case is that missing point falls on an edge which share with two triangles. In this case as shown in figure 4.23 (b) the two triangles are subdivided into four triangles. The last inserting case is as shown in figure 4.23 (c) that missing point falls outside the triangulation. This will be a more tedious process to fix the convex-hull of triangulation. This is a time consuming process and affect the large number of triangles. As discussed earlier in step 2, this situation is somehow avoided by introducing fake boundary points which still cover a big region outside the actual triangulation. This will make sure that all the missing points are located within the triangulation.

The computing process of this step by considering first two cases (missing points are located within a triangle and on an edge) affects only two triangles for each insertion of point. To insert a point, first the point should be located on which triangle or on which edge that point is located. Searching the triangle location for each point is a non-trivial process. This situation is efficiently handled by keeping the record of neighboring vertices to those missing points. For the missing points from step 1 (mapping the points), each missing point is recorded the corresponding grid point where it should be located. Using this grid point from digital Voronoi diagram corresponding vertex p_i and a corresponding triangle instance can be identified where that point is located. For the set of missing points from step 3 (bad case points), before deleting that point a neighbor point is recorded. From this point a triangle instance can be identified and the search can be started from this triangle to quickly locate the triangle.

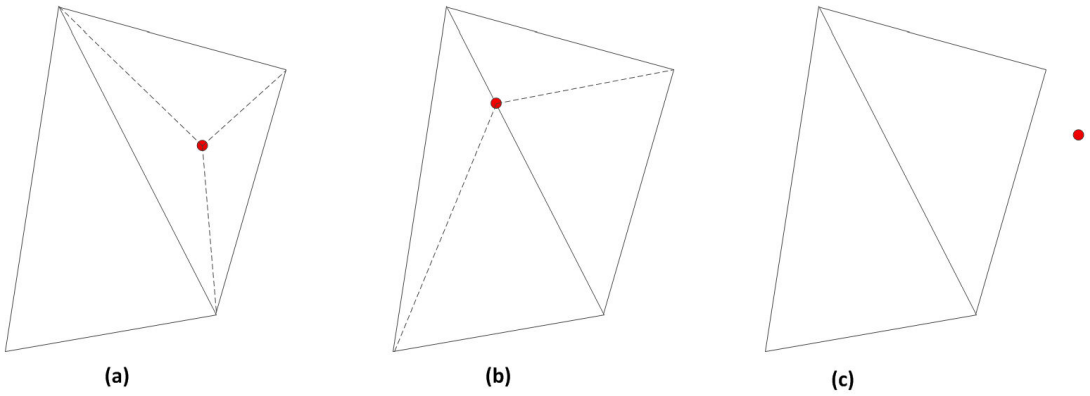

Figure 4-22 Inserting a missing point within a triangulation: **(a)** Within a triangle, **(b)** On an edge, **(c)** Outside the triangulation

A parallel implementation of inserting missing points can be performed in multiple rounds. In processing using parallel threads, each thread deals with one missing point to insert without any conflict with other parallel threads. The main intention of parallel insertion is that each iteration inserts as much as number of points. When two parallel threads try to inset two different points within a same triangle, this situation is avoided by giving the priority to the point with smaller index and other threads have to wait for the next rounds. When two missing points fall on the same edge which are shared by two triangles, then the minimum index gets the chance to proceed in that round. Except the first round, the location search of missing points is done in subsequent rounds in such a way that, it starts from the triangle on which a point was inserted previously.

In the parallel implementation using CUDA several parallel threads can compete between them, but only one thread gets success at the end. In the case when two threads try to insert points on the same edge there is a necessity of thread synchronization between parallel threads to allow the thread with minimum index. There is an atomic function $min()$ is available in CUDA, this will introduce the threads synchronization implicitly. The thread synchronization is not an attractive choice in CUDA by considering the parallel performance. Since the missing point falling on an edge is a very rare case, this synchronization will not cost much in the GPU-DT performance. According to Cao's experimental results [CAO, 2009], for a uniformly distributed point-set, the insertion of missing point step is relatively faster and ends in small number of iterations.

4.2.4.5 Flipping Triangle Edge (step 5)

In this step the existing triangulation is transformed into a valid Delaunay triangulation. To confirm the Delaunay property all over the triangulation, every edge is checked for in-circle property as shown in figure 4.24. A parallel implementation is possible in multiple iterations, because flipping one edge can lead to another edge being flipped. Each thread takes responsible for one triangle and it checks with three neighbor triangles whether they satisfy the in-circle property or not. If not the two neighbor triangle are identified for flipping. To avoid the flipping of a same edge by two parallel threads, only the thread with smaller thread *id* will take care of flipping and other thread will leave the flipping effort. During each edge flipping, triangle's new vertex detail and neighbor triangle detail should be updated.

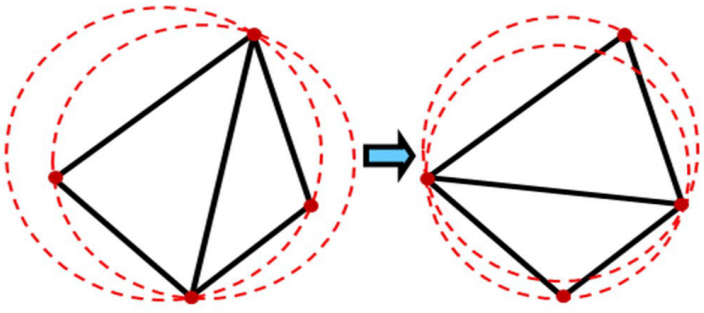

Figure 4-23 Flipping an edge to validate in-circle Delaunay property

To speed up the flipping process, in subsequent rounds of flipping only the triangles which are not flipped in the previous round are considered. The in-circle test is very numerically sensitive. For a robust in-circle test, Shewchuk's algorithm [SHEWCHUK, 1996] is preferred.

But in the CUDA implementation of GPU-DT, adopting this algorithm is more complicated. Instead of this robust algorithm, a simple in-circle test function (less precise) is implemented in CUDA. In this implementation, during the in-circle test any four vertices which are almost on a co-circular then those triangles are separately checked by recursive flipping routine using robust algorithm in the CPU level for the robustness of this algorithm.

4.3 Mesh Decomposition

Parallel computing of finite element analysis problem is mainly based on the concept of domain decomposition technique. In the sense of finite element method, domain decomposition is the technique dividing a large and computationally time consuming problem into manageably smaller sub-problems and they are solved in an efficient way in parallel [TOPPING AND KHAN, 1996].

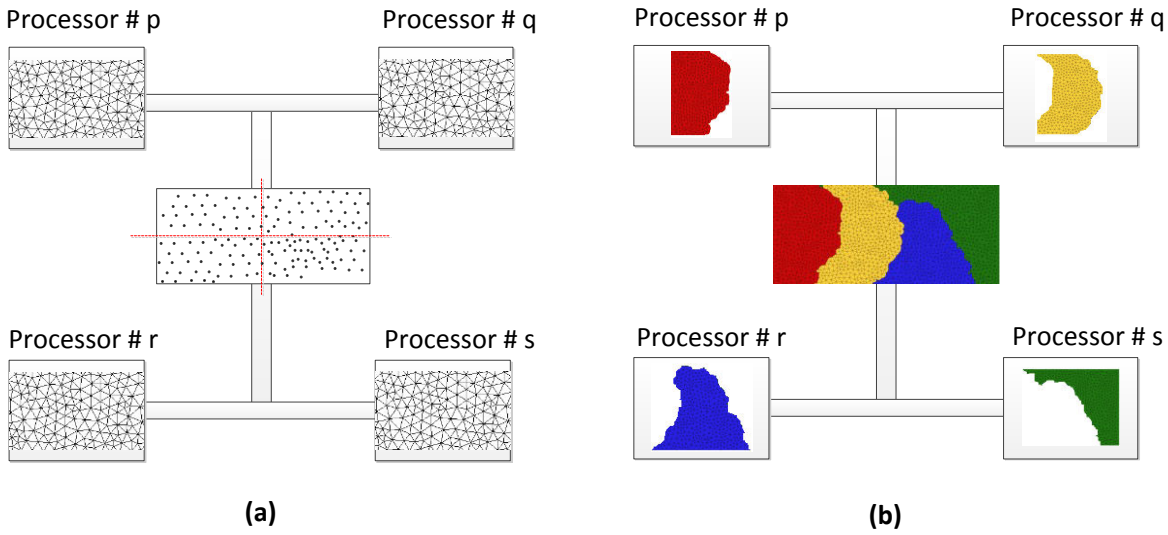

Figure 4-24 (a) A priori partitioning, **(b)** A posteriori partitioning

This section is mainly focused on mesh decomposition techniques used in FEM. To design an efficient mesh decomposing algorithm, there are few important considerations considered in general. First an efficient partitioning is done to introduce the number of balanced sub meshes which are depending on the availability of the number of processors (processing elements-PE). Balanced sub meshes are defined depending on the work-load of different sub domains rather than the geometry size of sub domains. In general for an unstructured meshing problem, the work load is not uniformly distributed throughout the mesh domain. In this case partitioning into balanced sub meshes can produce irregular and different size of sub domains. Second consideration is an efficient merging process of sub meshes into a final complete

mesh. Fast mesh decomposing algorithms are optimized for reduced size of interface boundaries between subdomains to reduce the communication overheads between processors.

In decomposing a regular shape of mesh domain (square, rectangle,) partitioning technique is applicable to produce regular shape of sub domains (box, stripes). However partitioning a complex domain is a non-trivial process. This partitioning produces the irregular shape sub domains. In a design of general purpose automated mesh decomposer, it is taken into account of the algorithmic, numerical and hardware issues to deliver an automated and robust algorithm [TOPPING AND IVANYI, 2010].

There are main two aspects, which are considered in the application of domain decomposition in FE problems. They are explicit and implicit approaches [TOPPING AND KHAN, 1996]. In the explicit approach finite element mesh is physically partitioned into subdomains to process in parallel. The implicit approach deals with partitioning of system of assembled equations to perform parallel analysis. In this section, explicit partitioning methods for parallel mesh generation are discussed.

The main focus in this section is to give an overview of mesh partitioning technique and available methods. The main two categories of mesh partitioning are posteriori and priori partitioning techniques. In posteriori methods a large size mesh is split into number of small size sub-meshes and given to different processors for the further analysis. In a priori technique, subdomain mesh partitions are directly constructed from domain geometry or coarse mesh without producing the complete large size fine mesh (as shown in figure 4.25 (a)). As shown in figure 4.25 (b), one of the posteriori mesh partitioning is used to split the mesh into four sub-meshes. In contrast to the posteriori method, priori methods are easy to implement and perform faster. The memory requirement to compute the meshing problem is less than posteriori method.

4.3.1 Priori Partitioning Method

In this type of mesh partitioning, the requirement of initial mesh at the beginning is not necessary. Hence the main drawback of a large memory requirement is avoided not like in posteriori partitioning method. Partitioning of domain is considered to start from an empty mesh domain or a starting mesh which is relatively a coarse mesh. Another approach in this type of partitioning is extracting partitions from the boundary or surface of the entire mesh domain. In partitioning using both approaches the balancing between subdomains and interface smoothness are considered. The difficulties exist in the priori approach are load

balancing between subdomains and properly managing the domain interfaces. Load balancing is deduced from coarse geometry or the domain boundary. Direct partitioning or pre-partitioning is introduced by [GALTIER AND GEORGE, 1997] which is kind of priori partitioning technique. In this method subdomains are produced at the beginning by defining the set of separators of domain boundary facets.

4.3.2 Posteriori Partitioning Method

The detail of this section is based on the literature [FREY AND GEORGE, 2008]. This type of partitioning technique is possible with different methods such as greedy method, spectral method, inertial axis partitioning, K-means method and so on. The main drawback in the posteriori partitioning technique is the requirement of large memory. The required memory size to handle each sub mesh is the sum of the memory resources to store the complete mesh plus the memory required to store that particular sub mesh. The important classical issues related with the mesh partitioning have to be considered in mesh partitioning. Few of these significant issues are the smoothness or regularity issue, required interface size, required number of connections between subdomains and keeping the balance between sub-meshes.

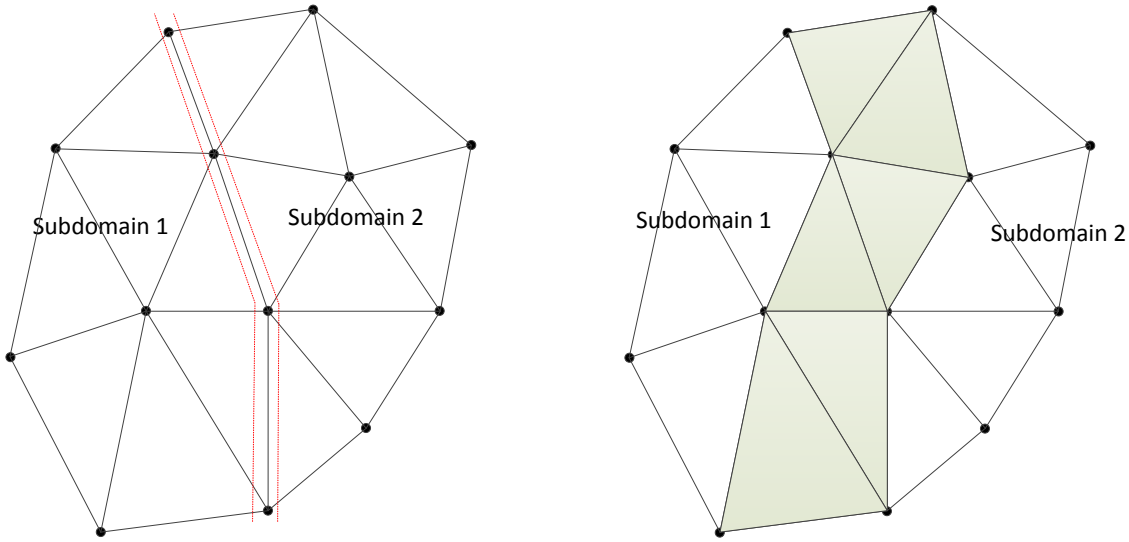

Figure 4-25 (a) Element based decomposition; **(b)** Nodal based decomposition

The main two entities based on which decomposition is applied are nodes and elements. In the element based decomposition (also called element oriented decomposition (EOD)), partitions are formed by distributing the elements among the subdomains as shown in figure 4.26 (a). That mean, the logical association of one element belongs to only one subdomain. This element based criteria is most frequently applied in mesh decomposition algorithms. In this

decomposition method, the interface between subdomains is constructed based on vertices and edges.

In the nodal based decomposition algorithms (also called vertex oriented decomposition VOD), vertices are distributed among the subdomains. That means one vertex is logically associated with only one subdomain as shown in figure 4.26 (b). The interface between subdomains is formed by elements which are shared with both neighboring subdomains. The common problem in the posteriori decomposition techniques is the difficulties in mesh parallelism. For a large size of mesh problem, producing the initial mesh (finer mesh) is sometime not possible. Due to this reason parallel processing is compromised with respect to size of the mesh. In the following sections few of the posteriori and priori decomposition algorithms are discussed.

4.3.3 Recursive Coordinate Bisection (RCB)

This partitioning algorithm works by splitting the mesh domain along the cutting plane which is perpendicular to the coordinate directions (x, y and z). By creating axis-aligned bounding box (AABB) of the mesh domain, cutting axis is decided. As shown in figure 4.27 (a) and (b), a 2D point-set is partitioned by bisecting the domain as vertical and horizontal partitions respectively. First cutting axis is decided to be perpendicular to the largest dimension of mesh domain. Then points are sorted according to the coordinate values in the direction perpendicular to the cutting axis. As last step, partition plane is determined as median of sorted coordinate values and point cloud is divided into two sub domains. This process is repeated until the required numbers of partitions are produced.

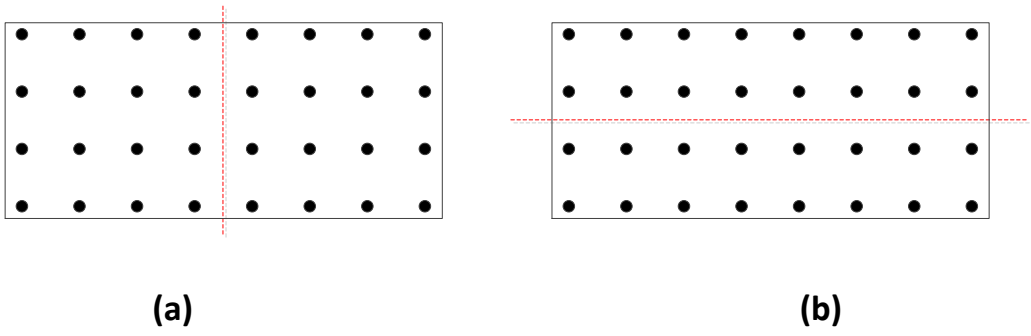

Figure 4-26 (a) vertical cutting plane, **(b)** horizontal cutting plane

4.3.4 Inertial Axis Recursive Partitioning

In this method an oriented bounding box (OBB) is used to decide the partition planes instead of AABB in RCB method. The direction of the OBB can be found by computing the eigenvectors of the co-variance matrix (CV) of the coordinates of the points as shown in equation 4.8. The center-point of the domain is $(\overline{x}, \overline{y})$. As shown in figure 4.28, mesh domain geometry and its inertial axis partitions of an example problem are given.

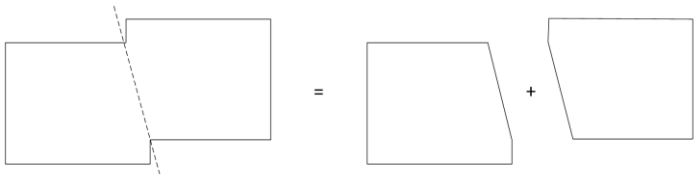

Figure 4-27 Inertial axis cutting of mesh geometry

$$CV = \begin{bmatrix} \sum (x-\overline{x})^2 & \sum (x-\overline{x})(y-\overline{y}) \\ \sum (x-\overline{x})(y-\overline{y}) & \sum (y-\overline{y})^2 \end{bmatrix} \qquad 4.8$$

4.3.5 Spectral Partitioning Method

This is one of the best performing graph partitioning methods using recursive bisections. This gives the best performance for unstructured mesh partitioning. This method is proposed by Simon and is also called Simon's method. In this method eigen vector information is used to determine the finite element mesh partitions by satisfying the conditions to have an equal number of elements in both side of bisected graph and to have a reduced interface boundary [TOPPING AND IVANYI, 2010]. However the computational cost increases non-linearly with the increasing size of the mesh to be partitioned.

In recursive spectral bisection method a laplacian matrix $L = [l_{ij}]$is formed as shown below,

$$l_{ij} = \begin{cases} 1 & \text{if elements i, j share and interface } i \neq j \\ -D_i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \qquad 4.9$$

Here the degree of an element D_i of a finite element mesh is taken as the number of elements sharing an interface with the element being considered.

4.3.6 Greedy Method

Greedy algorithm is one of the finite element mesh partitioning algorithm and performs relatively faster. This method produces subdomains from an overall finite element mesh sequentially. The number of subdomains created is equal to the number of available processor. The number of elements per subdomains is decided by the total number of elements, number of different type of elements and the number of available processors. In a mesh of single element type, the number of elements per subdomain is equal to the ratio between the total number of elements and number of processors. Each node is assigned with a weight factor which is equal to the number of elements connecting to that particular node. The algorithmic detail of this algorithm is based on the Farhat's domain decomposition method [TOPPING AND IVANYI, 2010].

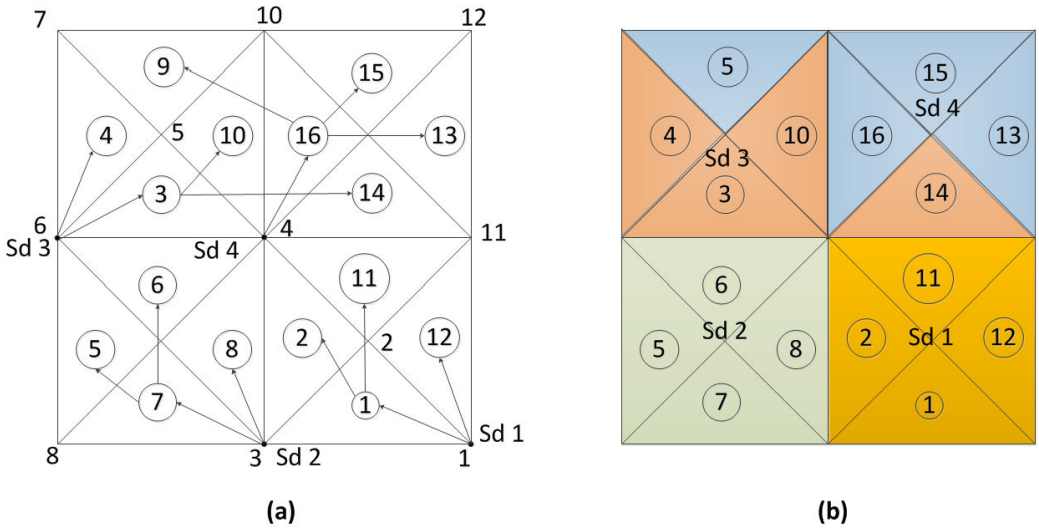

Figure 4-28 (a) Example of greedy algorithm using 4 subdomains to partition 16 triangle elements **(b)** produced mesh partitions using greedy method (based on [TOPPING AND KHAN, 1996])

As shown in figure 4.29, a simple example is used to explain the greedy algorithm based on Farhat's method [FARHAT AND ROUX, 1991]. The nodes 1, 8, 7 and 12 can be the starting nodes, because they have the lowest weight-factor (number of elements) as 2 elements. Since the number processor is decided as 4, the number of elements per subdomain is 4 (16/4). As shown in figure 4.29 (a) to the first subdomain (sd1) node 1 is selected as the starting node and first four elements (1, 2, 11and 12) are decided to subdomain 1. To the second subdomain (sd2) elements 7, 8, 5 and 6 are selected from the starting node 3. In this way subdomain 3 and 4 are decided for their four elements and the elements details of different subdomains are shown in figure 4.29 (b). From the partition results, one can notice that the subdomains one and two are well formed, but three and four are not in the expected form. For the subdomain

three, element 14 was selected instead of 9. This shows the vulnerability in this method in the selection of optimal elements.

In this technique, there is always no guarantee to form an optimum subdomain interfaces, even though it performs a relatively fast. In this method, formation of mesh partitions is sensitive to the element size and node numbering of the finite element mesh. That mean different partitions solutions are possible for a given mesh topology by changing the numbering of elements and nodes.

4.3.7 Multi-Block Method

In the parallel mesh generation of structured mesh, multi-block method is widely used. Mesh domain is divided into elementary geometry entities as confirming or non-conforming divisions as shown in figure 4.30.

It is suitable and easy to implement this partition technique for automated parallel mesh generators. The main idea of partitioning is dividing the domain into small convex blocks. By populating the mesh vertices within each block, convexity is established through domain. Final mesh is obtained by merging every mesh-block. In unstructured mesh generation, meshes within blocks are structured mesh but globally it is not always a structured mesh [FREY AND GEORGE, 2008].

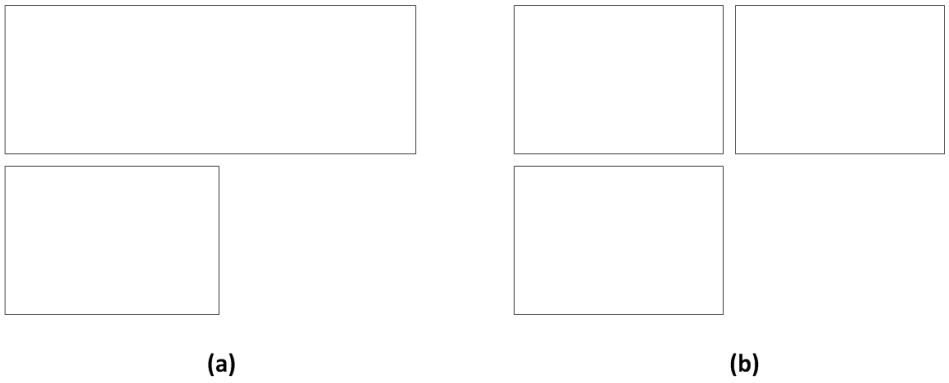

Figure 4-29 Multi-block divisions **(a)** non-conforming division, **(b)** conforming division

4.3.8 Geometric Domain Decomposition (GDD)

In decomposing a meshing problem a geometric domain decomposition technique can be used. In this method sub domains are created by inserting internal separators into the meshing domain [LINARDAKIS AND CHRISOCHOIDES, 2008]. There are many parallel mesh generation implementations using this approach which requires very low [CHEW ET AL., 1997] or no

communication [LINARDAKIS AND CHRISOCHOIDES, 2006, SAID ET AL., 1999, GALTIER AND GEORGE, 1997] requirement during parallel processing. This supports highly for an efficient parallel process. The importance attention in applying this decomposition technique that the separators should be small and satisfy the quality criteria like angle between separators should be satisfactory.

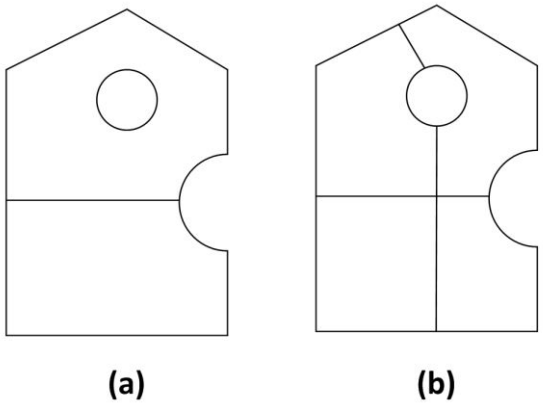

Figure 4-30 N-way partitioning process of a domain into (N = 2, 4) partitions using MADD

As shown in figure 4.31, an example geometry decomposition of a domain using medial axis domain decomposition (MADD) is given. This method was introduced in [LINARDAKIS AND CHRISOCHOIDES, 2006]. In this algorithm the medial axis concept is used to determine the domain separators in such a way that those make good angles between them [LINARDAKIS AND CHRISOCHOIDES, 2008].

Chapter 5

5 Multi GPU-DT Concept

5.1 Concept Overview

This research work focuses mainly on parallel Delaunay triangulation and unstructured mesh generation using a computing system accelerated by multiple GPUs. The GPU-DT [QI ET AL., 2013, CAO, 2009] is a currently existing parallel Delaunay triangulation implementation using GPU. This implementation is capable of computing a Delaunay triangulation for a given point-set using a single GPU computing system. Also this leverages a significant speed up over the existing CPU-based fastest sequential Delaunay triangulation.

In this research approach, GPU-DT is used as core algorithm for triangulation and part of this algorithm is upgraded to function on multiple GPUs system. The modified multi GPU-DT algorithm is partially capable to be computed on multiple GPUs computing system. The limitations exist in graphic hardware and CUDA technology place a challenge to upgrade the other parts of this algorithm. In general, unstructured meshes consists internal and external constraint-boundaries in addition to the domain boundaries and interior point-set. The already existing GPU-DT implementation does not handle boundary constrain edges of a general mesh domain. This thesis has upgraded the existing implementation and which is now capable to separately recognize the boundary and internal constraint edges and include them into the final mesh. The multi GPU-DT mesh generator can generate unstructured Delaunay meshes of a given point-set with constraint boundary edges.

In a general construction process of unstructured meshes, as a very first step, mesh geometry is defined by boundary points and boundary edges. This initial point-set is extended by adding refined interior points and refined boundary points. There are various methods and algorithms are already available to generate required mesh point-sets. In this implementation, a complete point-set of mesh points should be ready in the first stage. For this given mesh geometry, point-set is generated from other existing mesh examples and imported as an input data into multi GPU-DT mesh implementation. The details of constraint boundary edges are also passed as another set of input data. The main focus here is to develop an unstructured mesh for a given point-set using multiple GPUs.

As shown in figure 5.1, the flow chart shows the main processes of overall multi GPU-DT mesh implementation. This implementation contains four main steps of process ($S1, S2, S3$ and $S4$), as shown in the flow chart in figure 5.1. They are:

Step1: Dividing the point-set into number of vertical partitions

Step2: Construction of the interface region mesh of each partition plane

Step3: Meshing individual sub-domains in parallel using multiple GPUs

Step4: Merging sub-domain meshes into complete final mesh

In this study an overall multi GPU-DT concept is explained by using a small meshing problem. This explanation of concept covers all the necessary processes involved in a parallel mesh generator from the preparation of mesh geometry until exporting the mesh data as final mesh output.

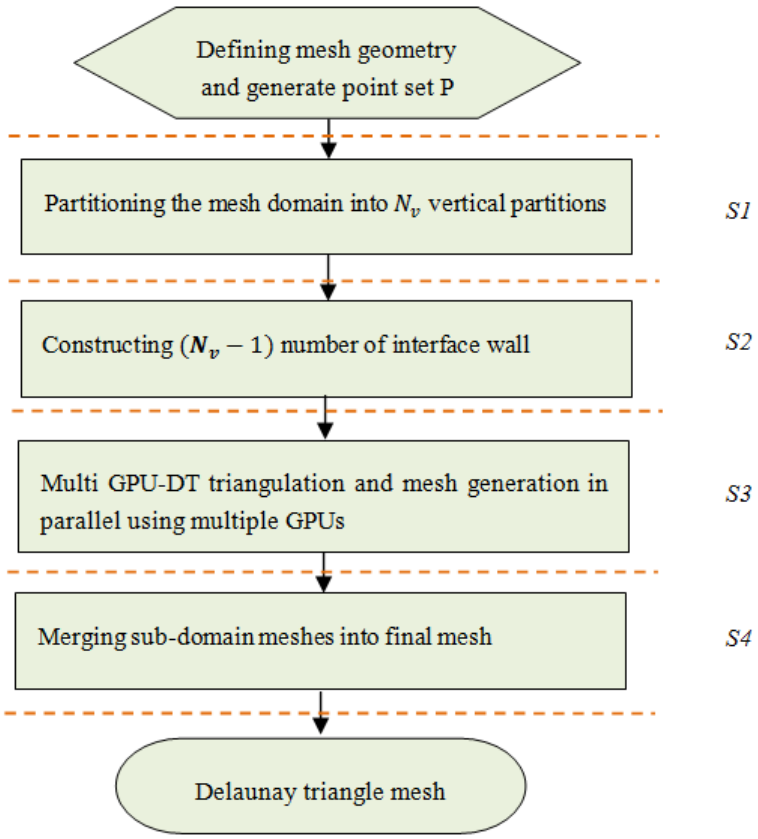

Figure 5-1 Flow chart of process in multi GPU-DT mesh

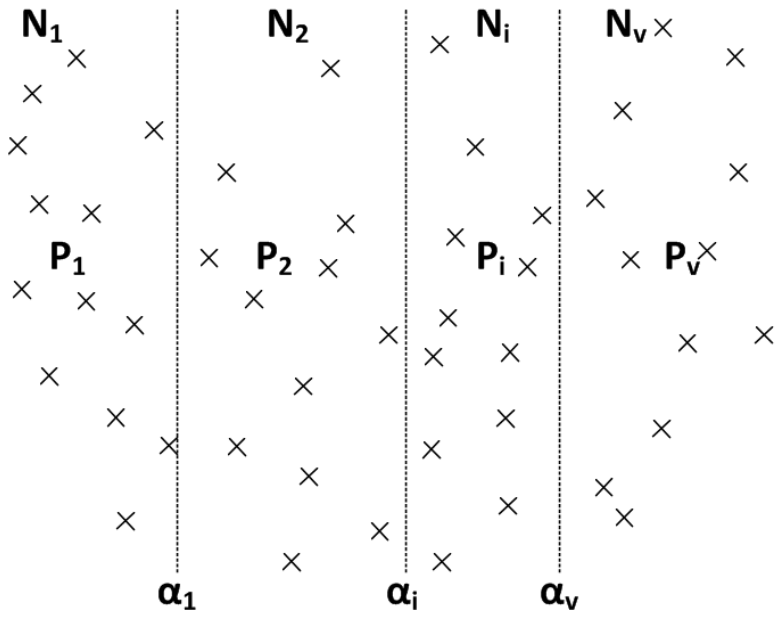

Figure 5-2 Partitioning of 2D point-set into number of N_V sub domains

5.2 Defining Mesh Geometry and Partitioning the Point-set

In a general meshing problem, mesh geometry is defined by boundary points. After that internal points or refining boundary points are introduced during the meshing process. There are meshing problem that defined with the meshing points in the first stage. These points can be produced from an output of existing mesh geometry or computed from any computational simulations. In this example, only point-set of random points within a given size of rectangular domain is used to explain the concept. This point-set is generated using uniform random generator. Constrained boundaries are not considered in this example and only focused to describe the Delaunay triangulation algorithm using multiple GPUs.

Partitioning the input point data into number of partitions is decided by the available number of GPU devices and size of the point-set. According to the experience the point-set contains less than 10 millions points is manageable within a single GPU for GPU-DT computing. The limited size of memory available in current graphics cards is a main restriction to incorporate large size of points. Over allocating points within one device leads to increase the number of missing-points, therefore it will ultimately affect the overall GPU-DT performance. In deciding the number of partitions, partition numbers is equal to available number of GPU is preferable. For a large size of mesh points with limited number of GPUs, the alternative solution would be a partitioning of domain into more number of sub domains than the

available number of devices (over partitioning). This requires multiple rounds of GPU-DT operation in sequential way and compromises the overall performance.

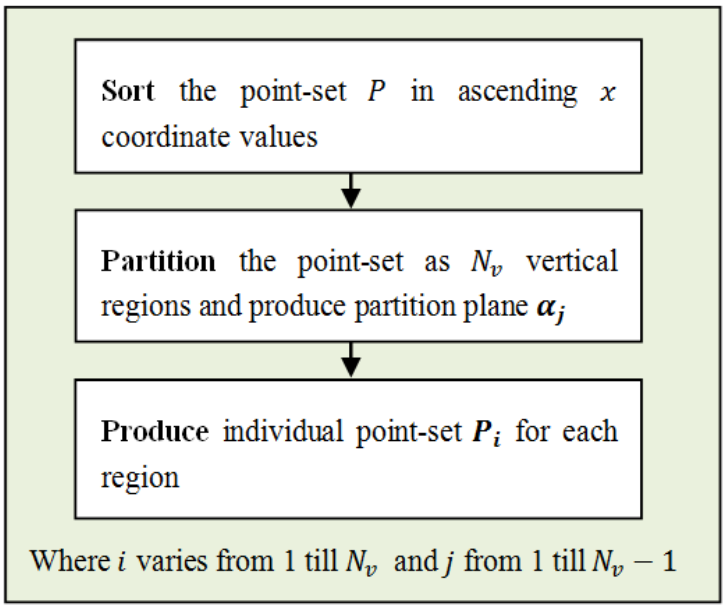

Figure 5-3 Flow chart of process in partitioning point-set

Partitioning the input point data into number of partitions is decided by the available number of GPU devices and size of the point-set. According to the experience the point-set contains less than 10 millions points is manageable within a single GPU for GPU-DT computing. The limited size of memory available in current graphics cards is a main restriction to incorporate large size of points. Over allocating points within one device leads to increase the number of missing-points, therefore it will ultimately affect the overall GPU-DT performance. In deciding the number of partitions, partition numbers is equal to available number of GPU is preferable. For a large size of mesh points with limited number of GPUs, the alternative solution would be a partitioning of domain into more number of sub domains than the available number of devices (over partitioning). This requires multiple rounds of GPU-DT operation in sequential way and compromises the overall performance.

In this step, as shown in figure 5.2, point-set is decomposed as sub point-sets according to the number of available GPUs. Number of partitions (N_v) is decided by available number of GPUs. For a meshing problem of a given point set and when the required numbers of partitions are very few, partitioning can be done using vertical partition planes. This partitioning can be done by deciding vertical partition planes $\alpha_i$to make sure an almost equal number of points locate in each partition. For each sub-domain individual point-set data is generated.

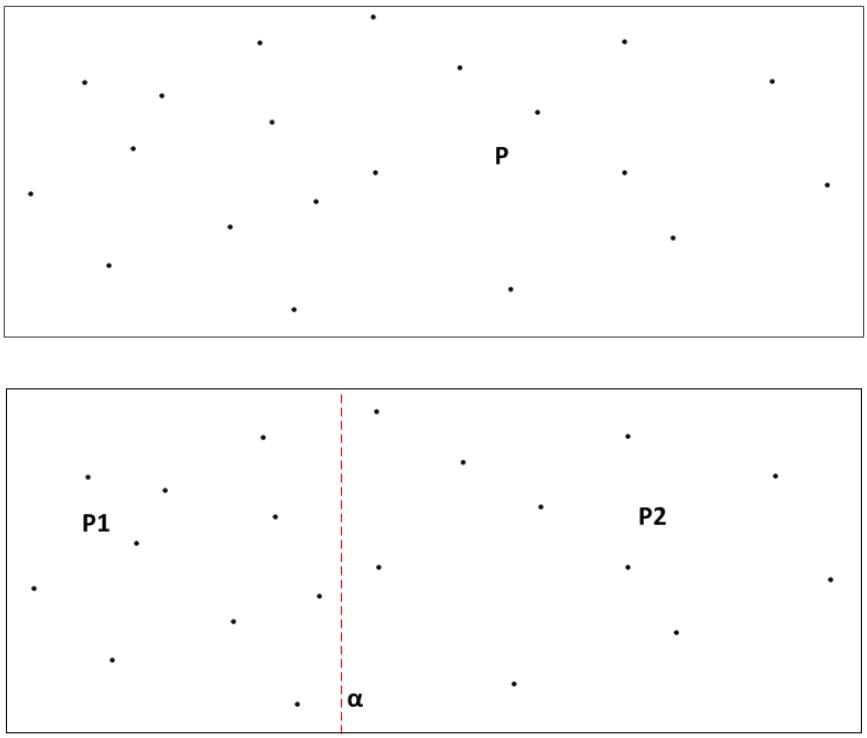

Figure 5-4 (a) Initial point-set P; (b) Partitioned point-sets P_1 and P_2

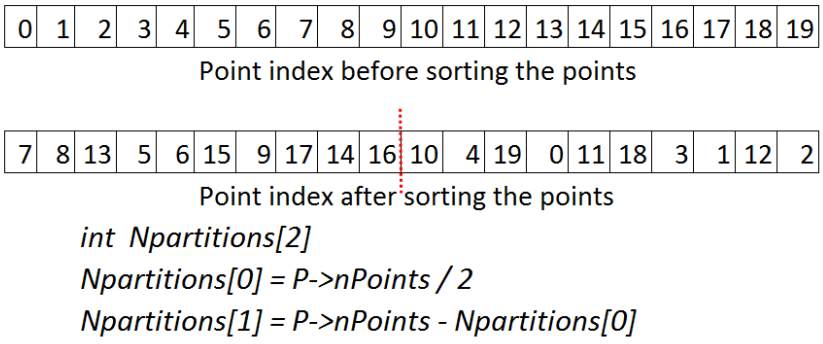

Figure 5-5 Index of points before and after sorted the coordinates

The flow chart in figure 5.3 describes the main three steps involved in partitioning process of the point-set. To partition the point-set into vertical partitions, first of all the coordinate values are sorted with respect to ascending x values. The sorted indices of points are recorded into an integer array to reference the original coordinate values of points. Actual partitioning is only done on this integer array by making sure of almost equal number of points are grouped into all partitions. If the numbers of points are not divided evenly into all partitions, the rest of the points are included in the last partitions. The partition plane is calculated by middle x value of the adjacent two partitions. Point-set is partitioned into N_v number of vertical partitions. After

partitioning, number of point-sets P_i are produced and stored as independent point-set to further process on different GPUs.

Table 5.1 Point-set P of example mesh problem and the two point-sets $P_1$and P_2 which are partitioned from the original point-set

Pointset P		
Index	X	Y
0	110	10
1	140	30
2	165	50
3	130	55
4	90	115
5	35	70
6	40	90
7	10	55
8	25	95
9	65	110
10	90	55
11	115	78
12	160	90
13	30	25
14	72	8
15	60	40
16	76	50
17	68	80
18	130	105
19	100	95

Pointset P_1		
Index	X	Y
0	10	55
1	25	95
2	30	25
3	35	70
4	40	90
5	60	40
6	65	110
7	68	80
8	72	8
9	76	50

Pointset P_2		
Index	X	Y
0	90	55
1	90	115
2	100	95
3	110	10
4	115	78
5	130	105
6	130	55
7	140	30
8	160	90
9	165	50

To explain the details of different steps of multi GPU-DT, a mesh problem is used by using a point-set as shown in figure 5.4. The selected example problem contains a point-set of 20 random points in a rectangular domain as shown in figure 5.4 (a). In this example two partitions are preferred to fully utilize both of the GPU devices available in our computing system. As shown in the figure 5.4 (b), point-set P is partitioned into two vertical partitions of point-sets $P_1$and P_2 by the partition line α. Point lists P_1 and P_2 are produced by keeping the track of original point ID from the point list P. As shown in the figure 5.5, the points in each partition are referenced by the global point index which is stored in the integer vector. This detail is necessary to produce the final mesh data as a standard mesh output using the original point ID information (global ID of vertex). As shown in table 5.1, the coordinate detail of initial point-set P and the partitioned point-sets P_1 and P_2 are given in different table.

5.3 Interface Wall Triangulation

After dividing the point-set, the interface region is triangulated using the incremental construction-InCoDe algorithm [CIGNONI, 1994]. To construct the simplexes in interface region, the incremental construction algorithms are used similar to Cignoni's Dewall algorithm. This approach is also used in the work of independent parallel Delaunay algorithm-IPDT from [CHEN ET AL., 2001]. The original implementation of incremental construction algorithm was proposed by [MCLAIN, 1976] by using empty circle property to construct Delaunay triangles. This algorithm starts from an initial triangle (simplex $s \in DT(P)$) and incrementally builds the successive triangles without modifying the existing triangles.

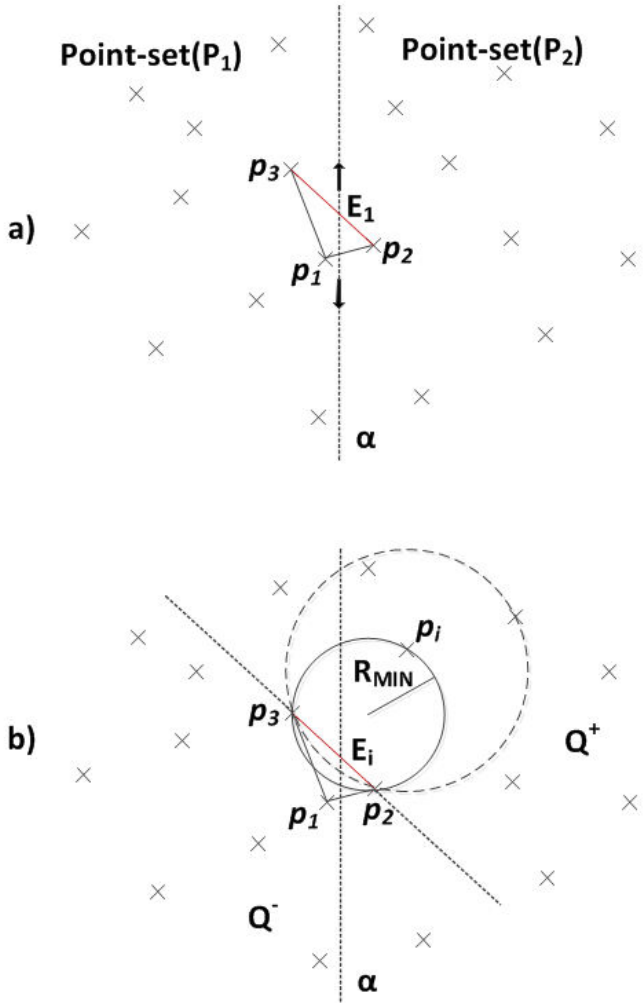

Figure 5-6 Interface wall triangulation: (a) forming initial simplex, (b) constructing generic simplex

The efficiency of triangulation of interface region using this InCoDe algorithm can badly influences the overall performance of triangulation. Since the triangulation is only focused to

find a small number of triangles which are intersecting the partition line, it will not significantly affect the overall performance of triangulation.

The construction of triangles in the interface wall region between adjacent partitions consists of three main steps. As described in the original implementation in InCoDe algorithm, the main steps are generation of initial simplex, generation of generic simplex and termination of interface wall construction. The algorithmic and implementation details of these steps are presented in the following sections. To explain the process that determine the triangles in the interface wall region as shown in figure 5.8, same algorithmic steps are used like presented in the original version of this algorithm.

5.3.1 Make Initial Simplex

As shown in figure 5.6 (a) the construction of initial simplex in interface wall region is described in this section. In this process, first point (p_1) is decided in such a way that which should be close to partition line and located on the left side of it (point-set P_1 side). In the example problem (figure 5.7 vertex 9 (A) in the point-set P_1 is selected as p_1. Second point p_2 should be located in the right side of the partition line and it is selected to satisfy the minimum distance between p_1 and p_2. In this example p_2 is decided as vertex 0 (B) in the point-set P_2.

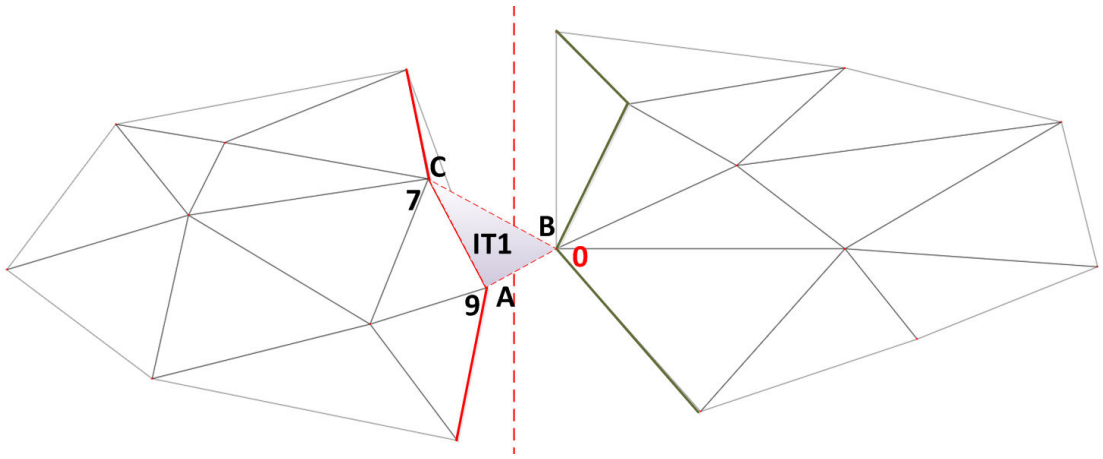

Figure 5-7 Construction of initial simplex of interface wall

Third step determines p_3 from either point-set P_1 or P_2 by satisfying the condition in such a way that the circumcircle-radius of the triangle $p_1p_2p_3$ should be the minimum. The Point p_3 can be located either in the left or in the right side of the partition line. In the example problem, p_3 has the vertex number 7 (C) which is located in the left side (point-set P_1). After determining the initial interface triangle (initial simplex) interface wall data structures have to

be updated. Interface wall data contains the four data structures which are one list of interface triangles and three lists of edges (active edges). The very first interface triangle is the initial simplex and which is inserted into the interface triangle list (IT) as shown in the table 5.2. To store the interface triangle detail, the first vertex v_1 represent the vertex in the P_1 point-set, the second vertex v_2 represent the vertex in P_2 and the third vertex v_3 can be either side 1 or side 2. To specify the third vertex detail, another integer is used as v3Side to represent the side of it. After extracting the edge details from the initial simplex, the three edge lists (for the left side (AEL1) and the right side (AEL2) of the partition line and the edges intersecting the partition line (AEL12)) are updated as shown in table 5.3.

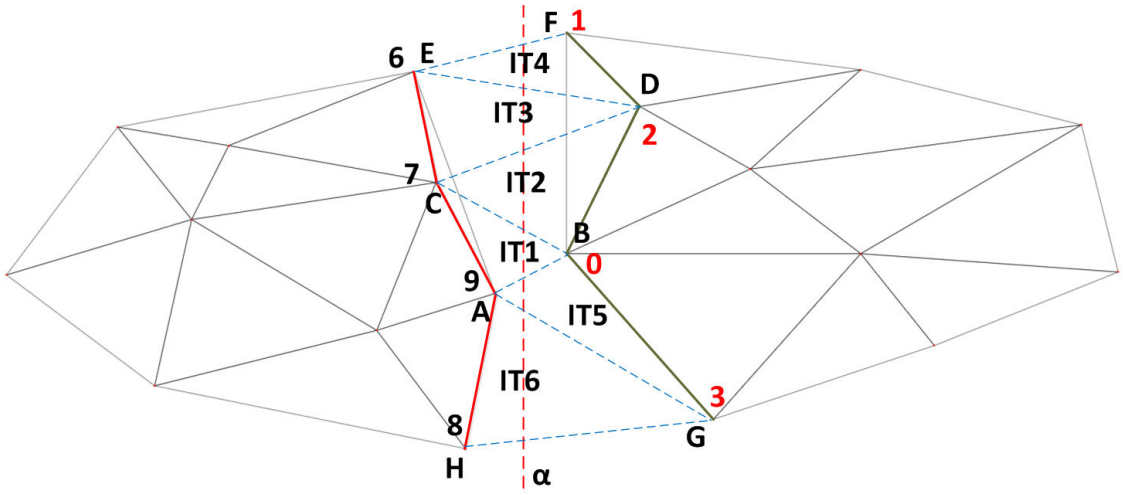

Figure 5-8 Construction interface triangles in interface wall

From the initial simplex ABC, the edges AB and BC intersect the partition line and they are inserted into the edge list of AEL12. The construction of the further Delaunay triangles from the initial simplex is progressed as two parts (see figure 5.8 (a) and (b)). As first, the upper edge BC is used to construct the interface triangles on the upper side of the initial simplex. As second, the lower intersecting edge AB is used to construct the triangles in the lower side of the initial simplex. As shown in the table 5.3, the intersecting edge list AEL12 is stored as two different lists (AEL12UP and AEL12DOWN) for the purpose of easy implementation. The third edge AC is inserted into the AEL1 list, since which is fully belong to the side 1. As shown in the data structures, the edges are recorded with the local vertex ID and the side details. The intersecting edges are recorded with the side value of 12.

Table 5.2 Initial simplex (Triangle ABC) in the example meshing problem

InterfaceTriangle	v1	v2	v3	v3Side
0	9	0	7	1

Table 5.3 Active edge list details after determining the initial simplex

Active Edge List side 1			
nEdges	1		
	Edge		
AEL1	v1	v2	side
0	9	7	1

Active Edge List side 12			
nEdges	2		
	Edge		
AEL12UP	v1	v2	side
0	7	0	12

Active Edge List side 2			
nEdges	0		
	Edge		
AEL2	v1	v2	side

	Edge		
AEL12DOWN	v1	v2	side
0	9	0	12

5.3.2 Make Simplex

In this step, the construction of further triangles is continued from the initial simplex. For a given edge (called active edge) another vertex is searched from the point-set which should satisfy the minimum radius condition as shown in figure 5.6 (b). As shown in figure 5.7, the initial simplex of the given example mesh problem is triangle ABC. Further interface triangles are generated along the partition line (see figure 5.8). The edge BC is the active edge to construct simplexes in the upper part of the initial simplex. By using the edge BC as an initial edge, the next generic simplex BCD is identified. In this way other simplexes CDE and DFE are determined in the upper part. Similarly in the lower part of the initial simplex by using the edge AB as the initial edge (active edge), the other interface triangles are determined. They are triangles AGB and AHG. The determined interface simplexes are from IT1 till IT6 are inserted into the interface wall triangle list (IWT).

While each simplex is identified, the edge list is immediately updated with new edges. In this example the active edge list of a point-set P_1 (AEL1) contains the edges AC, CE and AH. The Active edge list of point-set P_2 (AEL2) contains the edges BD, DF and BG. The edges which are intersecting the partition line (AEL12) are AB, BC, CD, DE, EF, AG and HG in the

identified order. The searching of simplex terminates when the active edge does not find a third point on its searching half-space. To speed up the searching process of a point out of complete point-set, only points which are on the outer half side of the active edge is considered. The other side is not a valid side and which contains the already constructed triangles. At the end of this process, the interface triangles details and the active edge list details appear as shown in table 5.4 and table 5.5 respectively.

Table 5.4 Final triangle list of interface wall

List of interface triangles (IW)				
InterfaceTriangle	**v1**	**v2**	**v3**	**v3Side**
0	9	0	7	1
1	7	0	2	2
2	7	2	6	1
3	6	2	1	2
4	9	0	3	2
5	9	3	8	1

Table 5.5 Final active edge list of interface wall

Active Edge List side 1			
nEdges	3		
	Edge		
AEL1	**v1**	**v2**	**side**
0	9	7	1
1	7	6	1
2	9	8	1

Active Edge List side 2			
nEdges	3		
	Edge		
AEL2	**v1**	**v2**	**side**
0	0	2	2
1	2	1	2
2	0	3	2

Active Edge List side 12			
nEdges	2		
	Edge		
AEL12UP	**v1**	**v2**	**side**
0	7	0	12
1	7	2	12
2	6	2	12
3	6	1	12

	Edge		
AEL12DOWN	**v1**	**v2**	**side**
0	9	0	12
1	9	3	12
2	8	3	12

5.3.3 Termination of Interface-wall Construction

During the construction of interface wall triangles, the construction process terminates when the active edge reaches to outer edge which belong to the convex hull of the complete point-set. In this example problem, the upper triangles construction ends when active edge reaches

to EF as shown in figure 5.8. Similarly when the active edge in the lower side reaches to HG, the construction of interface triangle finishes.

5.4 Delaunay Meshing Using GPU-DT

In this step the Delaunay unstructured mesh is generated using GPU-DT algorithm on multiple GPUs for different individual points set. The existing GPU-DT implementation is upgraded to execute on multiple GPUs concurrently. Actually this algorithm is utilized only for Delaunay triangulations as a part of mesh generation process. The basic functionality of the GPU-DT algorithm is the construction of Delaunay triangulation of a convex-hull of a given point-set. The additional tasks which generally involved in a triangular mesh generation are integrated to function as an automated mesh generator. Depending on the meshing task, a convex boundaries and internal holes are handled as constraint boundaries and they are fixed at the end of meshing process.

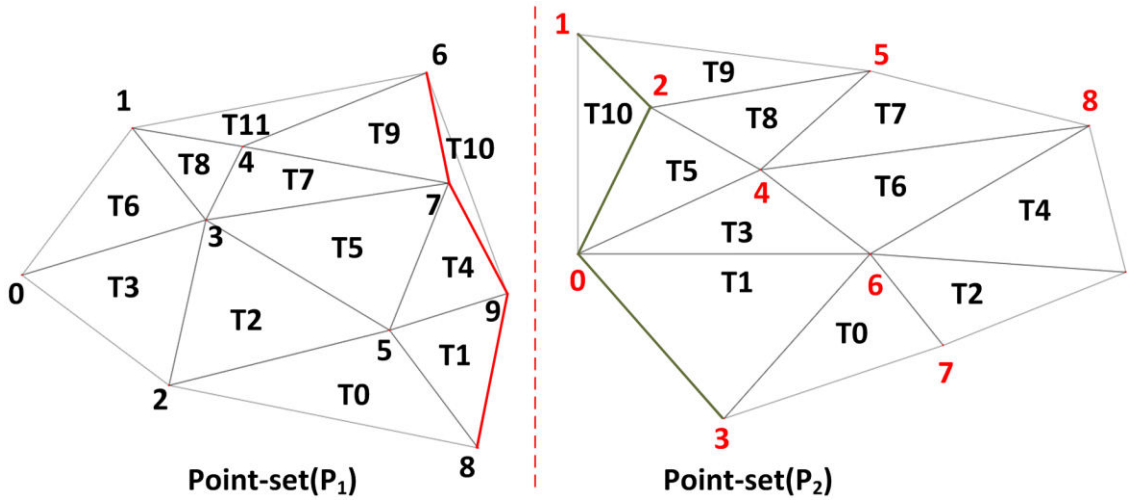

Figure 5-9 Triangle meshes of point-sets〖 P〗_1 and P_2 before merging

In the example mesh problem considered in this section to explain the multi GPU-DT concept, the triangulation results look like in figure 5.9. The details of triangulation output of two sub meshing problem are given in table 5.6 and table 5.7 respectively for point-set P_1 and P_2. Two sets of individual point data is used to compute the Delaunay mesh using GPUs in parallel. The algorithmic details of GPU-DT are discussed in section 6.1 and CUDA implementation details are discussed in section 6.2.

Table 5.6 Output of GPU-DT triangulation of point-set P_1

GPU-DT output			pOutput1			
nTris			12			
Index	Vertices int[3]			Neighbor Triangles int[3]		
	v1	**v2**	**v3**	**t1**	**t2**	**t3**
0	5	2	8	-1	1	2
1	5	8	9	-1	4	0
2	5	3	2	3	0	5
3	2	3	0	6	-1	2
4	9	7	5	5	1	10
5	3	5	7	4	7	2
6	3	1	0	-1	3	8
7	3	7	4	9	8	5
8	4	1	3	6	7	11
9	6	4	7	7	10	11
10	7	9	6	-1	9	4
11	4	6	1	-1	8	9

Table 5.7 Output of GPU-DT triangulation of point-set P_2

GPU-DT output			pOutput2			
nTris			11			
Index	Vertices int[3]			Neighbor Triangles int[3]		
	v1	**v2**	**v3**	**t1**	**t2**	**t3**
0	7	6	3	1	-1	2
1	0	3	6	0	3	-1
2	7	9	6	4	0	-1
3	6	4	0	5	1	6
4	8	6	9	2	-1	6
5	4	2	0	10	3	8
6	4	6	8	4	7	3
7	4	8	5	-1	8	6
8	2	4	5	7	9	5
9	2	5	1	-1	10	8
10	2	1	0	-1	5	9

5.5 Merging Sub-domain Meshes

As a final process, all sub-domain meshes are merged to form complete Delaunay mesh. Merging is a relatively trivial task, because interface wall and adjacent neighbor sub-domain meshes are already computed in the previous steps. Only triangle neighbor relations are updated between neighbor domain meshes.

As shown in the flow chart in figure 5.10, merging step is performed in three main steps. They are determining the convex-hull edges, updating the triangle list by identifying and deleting the non-Delaunay triangles from sub-block triangulations and finally combining the triangle list into final mesh. As the first step in merging process the convex-hull boundaries of left and right sides of the partition line are identified. One of the useful features of the GPU-DT algorithm is that it has already the details of the convex-hull information of the given points in addition to triangulation. That means the results of the GPU-DT triangulation has always convex outer boundaries.

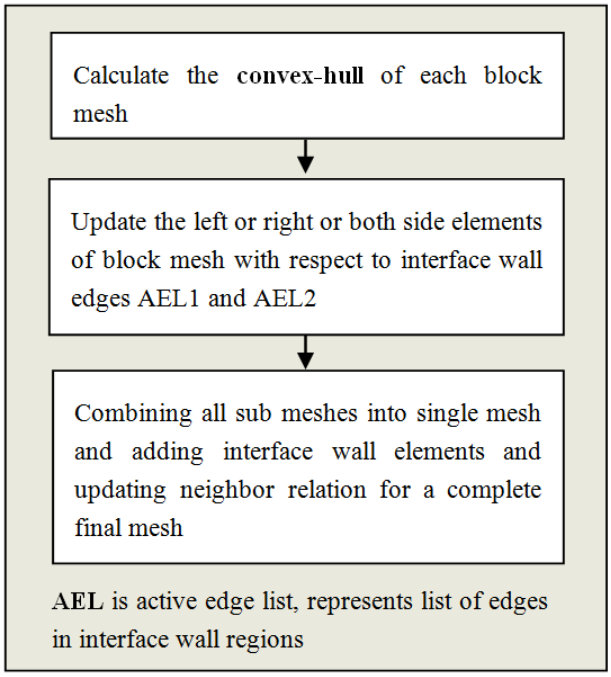

Figure 5-10 Flow chart of processes used in merging the sub-domain meshes

Since all the active edge lists are recorded from previous step (AEL1, AEL2 and AEL12) it is a trivial task to determine the upper and lower most boundary vertices (E, H in left side and F, G in the right side) (see figure 5.11 (a)). To determine the convex boundary edges in the left side (EA and AH), started from vertex E and traversed along the boundary triangles using the oriented triangle data structure used in [SHEWCHUK, 1996] algorithm. Similarly the convex boundary edges of right side triangulation (FB and BG) are determined. The detail of the convex edges related with the interface wall is listed in the table 5.8 and shown in figure 5.11(a).

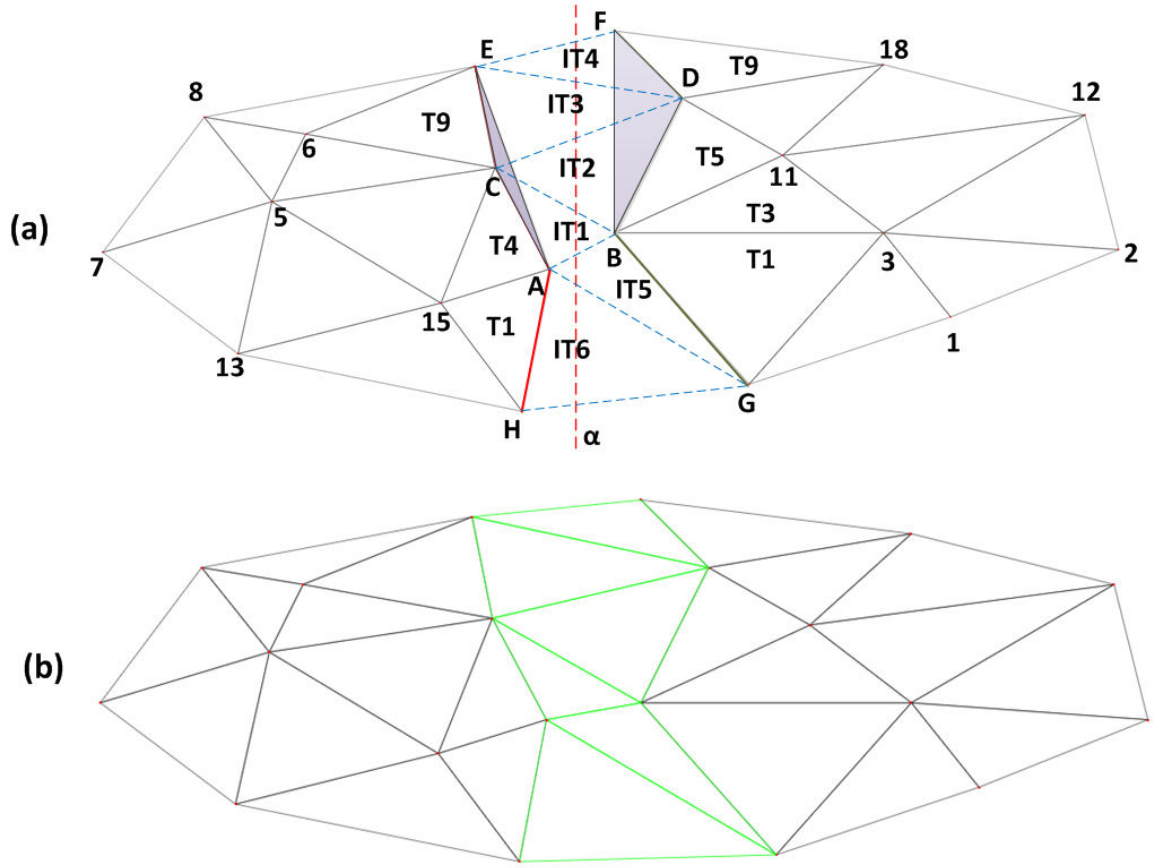

Figure 5-11 (a) merging of subdomain meshes with interface wall elements, (b) final mesh after merging

Table 5.8 List of convex edges of both side point-sets neighboring with partition line

List of covex edges of interfacewall of point set P1			
nEdges	2		
ConvexEdgeP1	Edge		
	v1	v2	side
0	6	9	1
1	9	8	1

List of covex edges of interfacewall of point set P2			
nEdges	2		
ConvexEdgeP2	Edge		
	v1	v2	side
0	1	0	2
1	0	3	2

Table 5.9 List of deleted triangles during the merging process of triangles from both sides of partition line

List of deleted triangles point set P1	
nDelTris	1
Idx	Triangle No
0	10

List of deleted triangles point set P2	
nDelTris	1
Idx	Triangle No
0	10

Table 5.10 Interface triangles in the GPU-DT triangle structure format

List of interface triangles (IW) in GPU-DT structure						
nTris			6			
Index	Vertices int[3]			Neighbor Triangles int[3]		
	v1	**v2**	**v3**	**t1**	**t2**	**t3**
0	9	19	4	21	-1	
1	17	19	9	23	9	25
2	17	10	19	17	24	26
3	16	10	17	25	4	27
4	10	16	0	28	13	26
5	0	16	14	1	-1	27

In merging both side sub meshes, interface triangles are used. The triangulations of both sides are locally Delaunay. When two locally Delaunay triangulation are combined into one triangulation, the global Delaunay property might not be valid among all triangles. It is because; during the triangulation of one side points the points of neighbor side are not considered. As shown in figure 5.11 (a) the colored triangles (AEC, BDF) in both sides of interface regions are non-Delaunay, even though they are locally Delaunay. To merge the both sides of sub meshes with interface triangles, these non-Delaunay triangles should be deleted first from the triangle list.

To identify these deleted triangles, edge list of convex edges and AELs are used. By taking into account of non-duplicate edges from the both edge lists, triangles sharing those edges are marked to be deleted. This is done in multiple rounds until the list of non-duplicate edges becomes empty. As an example in the considered mesh problem (figure 5.11 (a)) to identify

the triangle AEC, the non-duplicate edge AE is used to identify the triangle AEC. The similar process is carried out to the right side of the partition line to identify the triangle BDF. The list of deleted triangles as shown in table 5.9 is recorded to be used in updating the final mesh table in the next step.

Table 5.11 Output of Multi GPU-DT triangulation of point-set P_1 and P_2

Multi GPU-DT output					pOutputFinal	
nTris	12					
Index	Vertices int[3]			Neighbor Triangles int[3]		
	v1	**v2**	**v3**	**t1**	**t2**	**t3**
0	15	13	14	-1	1	2
1	15	14	16	-1	4	0
2	15	5	13	3	0	5
3	13	5	7	6	-1	2
4	16	17	15	5	1	26
5	5	15	17	4	7	2
6	5	8	7	-1	3	8
7	5	17	6	9	8	5
8	6	8	5	6	7	11
9	9	6	17	7	24	11
10	0	16	14	1	-1	22
11	6	9	8	-1	8	9
12	1	3	0	13	-1	14
13	10	0	3	12	15	22
14	1	2	3	16	12	-1
15	3	11	10	17	13	18
16	12	3	2	14	-1	18
17	11	19	10	25	15	20
18	11	3	12	16	19	15
19	11	12	18	-1	20	18
20	19	11	18	19	21	17
21	19	18	4	-1	23	20
22	10	16	0	10	13	26
23	9	19	4	21	-1	
24	17	19	9	23	9	25
25	17	10	19	17	24	26
26	16	10	17	25	4	27
27	10	16	0	28	13	26
28	0	16	14	1	-1	27

The merging of sub domain meshes is basically a process of combining the triangle list of both subdomains and interface triangle list into a single triangle table. The final mesh output is kept in the similar GPU-DT triangle data structure. The format of both side mesh output is already in the same triangle format (table 5.6 and table 5.7). But the triangles introduced in the interface wall region are not recorded in triangle format. The details of these triangles are recorded with local vertex index of respective subdomains as shown in table 5.4. To combine these interface triangles into the final mesh, these triangles are transferred into the global vertex indices representation and recorded in the same triangle format as shown in table 5.10. It should be noted that the GPU-DT mesh output from point-set is recorded with the local vertex indices (table 5.6 and table 5.7). They are easily replaced with the global vertex indices while recording the final mesh table.

As shown in table 5.11, the final mesh table is filled with the triangle list of point-set P_1 first, then point-set P_2 and at the end the interface triangles are inserted. The size of the final mesh table is considered as the number of triangles from point-set P_1, P_2 and interface triangles. Since few triangles are deleted from the both point-set, those records are used to store the few interface triangles from the bottom. In the given example as shown in the table 5.11, the triangle record 10 and 22 are deleted and those triangle records are replaced by interface triangles 27 and 28. At the end, the records 27 and 28 will be deleted from the table. The actual number of triangle in the final mesh will be the summation of number of triangle of point-set P_1, P_2 and interface triangles minus deleted triangles. In this example (12+11+6-2) 27 triangles are in the final mesh. During the combination process of triangles, updating the new neighbor relation should be done. Finally for the combined final mesh as shown in figure 5.11 (b), the mesh table looks like in table 5.11.

6 Implementation Details of Multi GPU-DT

In this section the algorithmic and implementation details of the multi GPU-DT algorithm is presented. The implementation details are described for a general meshing problem which is divided into almost equal N_v number of sub mesh problems as mentioned above (see figure 5.2). The number of sub-problems depends on the number of available GPUs. The sub-problems are processed on different GPUs in parallel, and then the resulting sub meshes are merged into a single mesh at the end. As shown in the figure 6.1, a simple meshing problem is partitioned as two sub mesh problems. The final mesh of this example problem is the combination of the sub meshes S_i , S_{i+1} and the interface region mesh S_{α_i}.

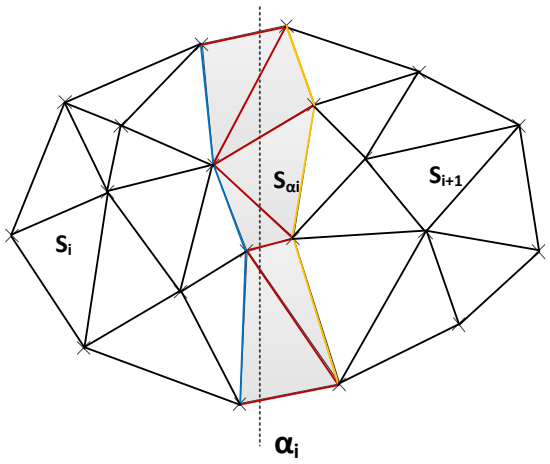

Figure 6-1 A mesh problem is partitioned as two sub problems and the sub meshes are merged into a single final mesh

The computation of this algorithm is a combination of both CPU and GPU computing. The meshing part is computed on GPUs and CPUs as a hybrid parallel computing task. The partitioning of the point-set and the merging of the sub-meshes are computed on CPU as a sequential task. In this approach, the partitioning into sub problems and the merging into the final mesh are implemented similar to the concept of the two-dimensional Dewall algorithm [CIGNONI, 1994].

6.1 Algorithmic Detail of Multi GPUDT-Mesh

The pseudocode of algorithm 6.1 explains the details of the four main steps used in multi GPU-DT mesh. This algorithm shows the general version of a meshing problem with N_v number of partitions. First the given point-set P is partitioned into N_v number of point-sets $(P_1, P_2, .. P_{N_v})$. After that the interface triangles (S_{α_i}) between P_i and $P_{i+1}$are computed for every partition line α_i. Then the meshing of point-set P_i is done with the GPU-DT-Mesh function using different GPUs in parallel. The resulting meshes are S_i and $S_{i+1}$from both sides of sub problems of a partition line α_i. Finally, all three meshes (S_i, S_{i+1} and S_{α_i}) of each partition line α_i are merged from left to right along the meshing domain. The resulting final mesh S is the combination of all subdomain meshes S_i's and S_{α_i}'s.

Algorithm 6.1: *Multiple GPU Delaunay mesh generator*

Function: Multi-GPU-DT-Mesh *(P: Point set) S: Mesh*

Input: *Point set (P)*

Output: *Triangle mesh (S)*

Local variables: *number of partitions (N_v), list of partition line (α_list), list of point set (p_list)*

```
begin
    Partition (P, Nv, α_list, p_list)
    for all αi
        Sαi:= INTTRI(Pi,Pi+1)
    end for
    for i = 0 till Nv-1
        Si:=GPU-DT-Mesh(Pi)
    end for
    for all αi
        S:= MERGE(Si, Sαi, Si+1)
    end for
end
```

Algorithm 6.2: *Partitioning the mesh domain*

Function: Partition *(P: Point set, N_v: number of partitions) α_list: list of partition line, p_list: list of point set*

Input: *P: Point set, N_v: number of partitions*

Output: *α_list: list of partition line, p_list: list of point set*

begin
P_{sorted} = ***Sort*** *(P(x))*
*P_list =****Generate point set*** $P_1, P_2 \ldots P_{Nv}$
for $i=0: N_v-1$
$\alpha[i] = (P_1.Xmax + P_2.Xmin)/2$
end for
end

6.1.1 Partitioning of Point-set

As a first process, point-set data is decomposed into sub point-sets according to the number of available GPUs. As shown in algorithm 6.2 the partition function has parameters point-set P, the number of partitions (N_v), the list of partition line (α_list) and the list of sub-domain point-set (p_list). The sorted point list P_{sorted} is produced by sorting the point-set with respect to the x coordinate values. According to the N_v, the sub point lists ($P_1, P_2, .. P_{N_v}$) are generated. The partition line $\alpha[i]$ for every partition plane is calculated. This is a middle vertical line of two closest points from both sides of that partition line.

6.1.2 Interface Wall Triangulation

As described in the original application work of Cignoni's algorithm, the interface wall simplexes $S_{\alpha_i}$are constructed in three main steps which are, (a) generation of initial simplex (*make_initial_simplex()*), (b) generation of genetic simplexes (*make_simplex()*) and (c) termination of the simplex finding algorithm to S_{α_i}. The detailed procedure is described as shown in the figure 5.8 and the algorithm 6.3 by using the same notations as in the work of Cignoni.

During the interface triangulation four different data are maintained. Three edge lists for the left and right side boundaries of the interface wall and the edges intersecting partition line and a fourth list for the triangle list of interface triangles. Figure 6.2 shows the data structures used in this implementation. An *Edge* structure in the interface wall region is represented by two integer indices (v1 and v2) of both vertices of an edge and another integer variable (side) to represent the side of the edge with respect to the partition line. The left side of the edge is marked with 1, the right side with 2 and the intersecting edge is marked with 12. The vertex indices v1 and v2 are recorded with the local index of the respective point-set.

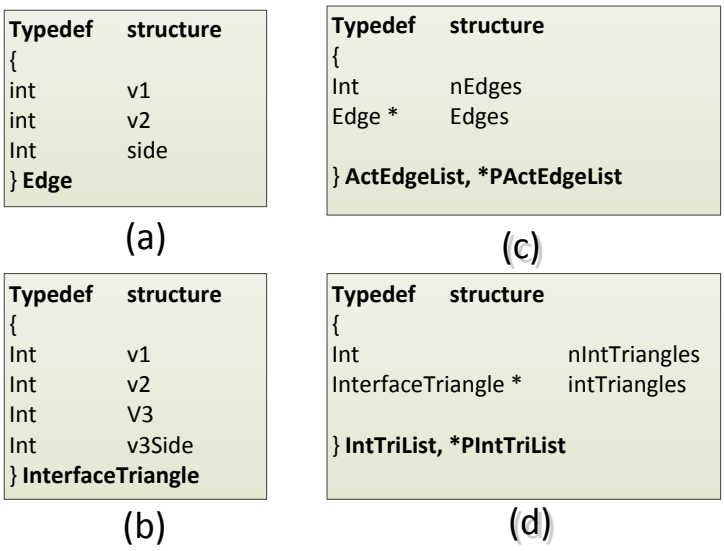

Figure 6-2 Data structures used in the interface wall triangulation and merging of mesh subdomains, representation of **(a)** an edge, **(b)** an interface triangle, **(c)** a list of active edges **(d)** a list of interface triangles

The three active edge lists (AEL1, AEL2 and AEL12) are recorded using the data structure *ActEdgeList* as shown in figure 6.2 (c). This data structure contains an array of edges and the number of available edges in the list. The interface triangle details are recorded using the data structures called *InterfaceTriangle* and *IntTriList* as shown in figure 6.2 (b) and (d).

Algorithm 6.3: *meshing of interface region between subdomains*

Function: Interface_Wall *(**P_1**: Point set1, **P_2**: Point set2, **α**: partition line)* ***AEL_α****: active edge list of interface,* ***AEL_1****: active edge list of point set 1,* ***AEL_2****: active edge list of point set 2,* ***TLIW****: triangle list of interface wall*

Input: *P_1: Point set1, P_2: Point set2, α: partition line*

Output: *AEL_α: active edge list of interface, AEL_1: active edge list of point set 1, AEL_2: active edge list of point set 2, TLIW: triangle list of interface wall*

Local variables: *number of partitions (N_v), list of partition line (α_list), list of point set (p_list)*

begin

if *(AEL =ϕ)* ***then***

t = ***make_initial_simplex****(P_1,P_2)*

AEL = edges(t); insert(t, TLIW)

end if

for *each $AE_i \in$ AEL do*

if *intersect(AE_i, α) $\neq \phi$* ***then*** *insert (AE_i, AEL_α)*

```
            else if AE_i  P_1 then insert(AE_i, AEL_1)
            else if AE_i  P_2 then insert (AE_i, AEL_2)
            end if
        end for
    while AFL_α ≠ ϕ do
        AE_j= extract(AFL_α)
        t = make _simplex(P_1,P_2, AE_j)
        if t ≠ null then insert(t,TLIW)
        end if
        end while
    for each edge AE_k ϵ edges(t) AND AE_k ≠ AE_j do
            if intersect(AE_k, α) ≠ ϕ then update(AE_k, AEL_α)
            else if AE_k  P_1 then update(AE_k, AEL_1)
            else if AE_k  P_2 then update(AE_k, AEL_2)
            end if
        end for
end
```

The computation of a given interface wall is performed by the function *Interface_Wall()* as shown in the algorithm 6.3. This function takes both side point-sets P_1 and P_2 as input parameters, computes and updates the other parameters (the active edge lists AEL1, AEL2, AEL12 and the interface triangle list TLIW).

6.1.2.1 Generation of Initial Simplex

The function *make_initial_simplex()* generates the first Delaunay triangle which intersects the partition line α. The successive construction of simplexes starts from this triangle along the partition line. As shown in the algorithm 6.4, the function *make_initial_simplex()* selects the first point p_1 which is nearest to the partition line α and located in the point-set P_1. The second point p_2 is selected which should be nearest to point p_1 and located on the other side of line α. Then the point p_3 is searched among point-set P ($P_1 \cup P_2$), by satisfying the condition that the circumcircle around the edge p_1p_2 and the point $p_3$has the minimum radius value. This function process terminates with a valid initial simplex ($p_1\ p_2\ p_3$) after the search of all points. The first simplex produces the first two edges ($p_1p_2,\ p_2p_3$) which intersect the line α as shown in figure 5.7.

Algorithm 6.4: *finding the initial simplex during interface wall triangulation*

Function: make_initial_simplex *(**P**$_1$: Point set1,* ***P**$_2$: Point set2)* ***s**: initial_simplex*

Input: *P_1: Point set1, P_2: Point set2*

Output: *s: initial_simplex*

begin

$p_1 = P_1(X_{max})$

$p_2 = \{p \in P_2; min(dist(p,p_1))\}$

$p_3 = \{p \in (p_1 Up_2); R_{min}(circle(p,p_1,p_2))\}$

initial_simplex = Triangle(p_1,p_2,p_3)

return intial_simplex

end

6.1.2.2 Generation of a Generic Simplex

The function *make_simplex()* constructs the adjacent Delaunay triangle of a given edge E_i (active edge AE) by satisfying Delaunay property. This function searches for a point p_i among all the points $p_i \in$ P which has the minimum radius of circumcircle passes through the edge AE and point p_i as shown in figure 5.8 (b).

The *make_simplex()* function selects the point p_i which minimizes the function dd (Delaunay distance) such that,

$$dd(\text{AE,p}) = \begin{cases} Rmin & if\ c \subset OuterHalfspace\ (AE) \\ -Rmin & Otherwise \end{cases} \qquad 6.1$$

The search for the points in function *make_simplex()* can be limited to only outer halfspace of AE Q^+ with respect to edge E_i as shown in figure 5.8 (b). The other halfspace (Q^-) contains the previously generated triangle and is not required to be searched for points.

For an active edge during the search for points p_i by the *make_simplex()* function, if there are no points on outer halfspace of that edge, that edge should belong to the convex-hull of point-set P. In this case, function *make_simplex()* returns no triangle simplex.

The generation of new triangle simplexes is performed from the two active edges of the initial simplex, one after the other. The algorithm searches for triangles on the upper side of the initial simplex first. After the upper side has reached the outer convex-hull edge, the lower side is started and continued to find the outer edge which is also on the convex-hull (as shown in the figure 5.8 (c)).

Algorithm 6.5: *finding generic simplex during interface wall triangulation*

Function: make_simplex *(**P_1**: Point set1, **P_2**: Point set2, **AE_i**: active edge)* ***s****: simplex*

Input: *P_1: Point set1, P_2: Point set2, AE_i: active edge*

Output: *s: simplex*

begin

$P_i = \{p \epsilon (P_1 U P_2)\ \&\&\ p \epsilon\ Q^+(AE_i);\ R_{min}(dd(AE_i, p))\}$

$simplex = Triangle(AE_i, p_i)$

return simplex

end

6.1.2.3 Termination of the Interface-wall Construction

When the *make_simplex()* function returns no more triangles on both sides of the initial simplex along the partition line, the interface wall construction is completed. The algorithm for interface wall construction returns the edge lists AEL (active edge list), such as;

AEL_α : The edges intersected by the partition line α,

AEL_1 : The edge list of interface wall boundary located in point-set P_1,

AEL_2 : The edge list of interface wall boundary located in point-set P_2.

6.1.3 GPU-DT Triangulation and Meshing on Multiple GPUs

The existing GPU-DT algorithm is implemented only on a single GPU device. In this research approach, the existing triangulation implementation is intended to be upgraded to exploit multiple GPU devices. The basic concept is that triangulation task is divided into many similar triangulation problems then they are solved using GPU-DT implementation on multiple GPU devices. Since the GPU-DT computation process is not a simple SIMD computing pattern, an independent computation on each device is not possible without intermediate calculations on the CPU. There are many limitations to implement the GPU-DT tasks running simultaneously on multiple GPUs. The communication bandwidth is a hardware related limitation, which influences the concurrent communication between different GPU devices and the CPU. The software level limitation is the CUDA API, which is a continuously developing technology. A MIMD computing paradigm like the GPU-DT process is a challenging task to use the combined parallelism of GPUs and CPU. In this implementation part of the GPU-DT implementation is modified to parallelize all the CUDA functions (kernels) and to be executed concurrently on multiple devices.

As shown in the figure 6.3 (a) for an example meshing problem, the existing implementation is capable of building Delaunay triangles without considering the internal or external geometry boundaries.

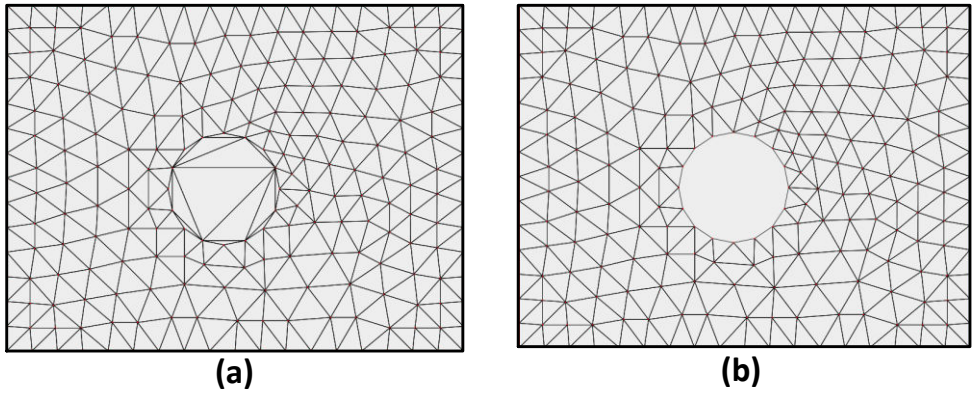

Figure 6-3 (a) GPU-DT triangulation of an example problem and **(b)** triangle FE mesh of that problem

Algorithm 6.6: *Fixing the constraint boundary and internal edges*

Function: fixConstraintBoundaries *(**mesh_in**: mesh, **c**: list of constraint edges) mesh_out: updated mesh*

Input: *P_1: mesh_in: mesh, c: list of constraint edges*

Output: *s: mesh_out: updated mesh*

begin
    repeat
        repeat
            v_j = select start vertex in constraint c_i
            tri0 = locate one triangle contains vertex v_j
            tri = Triangle (v_j v_{j+1} v_k)
            removeTriList (tri)
        until *(all the inner triangle which share the constraint edges to be removed)*
    until *(all the constraint boundaries be fixed)*
    mesh_out = update the mesh using removeTriList
end

The GPU-CDT is another version of GPU-DT algorithm which is capable of integrating the constrained edges in addition to the points-set (constraint Delaunay triangulation) (QI ET AL., 2013). This algorithm removes the intersecting triangles of a constraint edge. The constraint edge list detail is passed to the program as an input parameter. The algorithm runs through each given constraint edge by identifying the intersecting triangles and records them within

the removing list. The removing list of triangles can be identified in multiple rounds to avoid parallel threads conflicts between neighbor triangles.

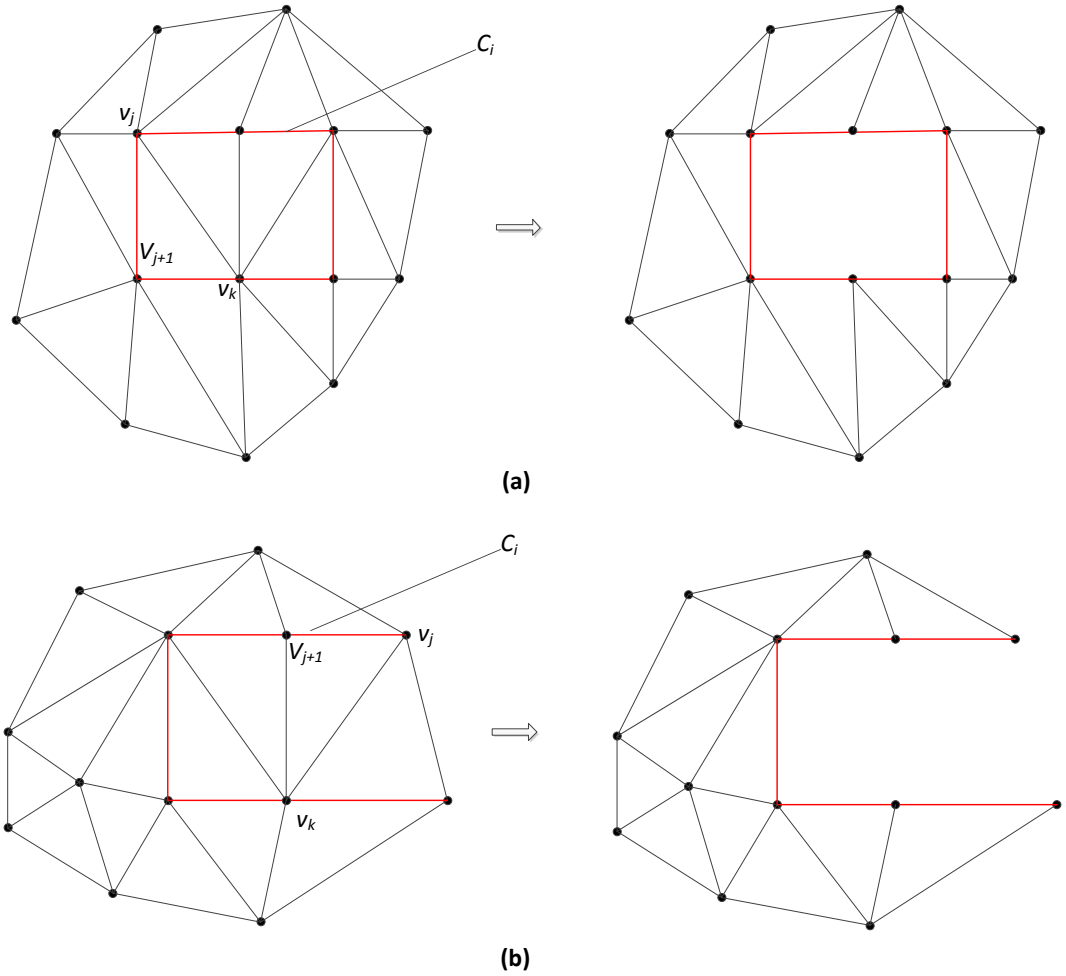

Figure 6-4 (a) Inner boundary constraint edges are fixed by removing internal triangles within the hole; **(b)** External boundary constraint edges are fixed by removing the external triangles neighboring with constraint edges

For the task of implementing meshing using GPU-DT, the existing functionality of a constraint edge removal in the GPU-CDT algorithm does not fully fulfill the requirement. As shown in figure 6.4 (a) and (b), constraint boundaries have to be defined with a number of finer edges, and these edges are part of the triangulation. Figure 6.5 gives data structure details which are used in the constraint edge implementation to store the constraint edge details. The EDGE structure is used to store the single constraint list which contains the number of constraint edges and edge details as an integer array containing adjacent vertex indices. The BND structure can store all the edge list details. As shown in the algorithm 6.6, triangles are sharing these constraint edges and those located outside of the boundaries are

identified and removed from the triangulation. The algorithm first locates the initial vertex v_j of a constraint edge list c_i. For that vertex v_j one triangle is identified which shares that vertex. From this starting triangle, next triangle $v_j v_{j+1} v_k$ is identified using the oriented triangle records (see figure 6.4). This process continues until all the constraint edges in that list are traversed and until all the constraint boundary lists are covered. As shown in figure 6.4, the external triangles shared with constraint edges are removed and the resulting meshes are formed as shown in the right side of the meshes in that figure.

```
Typedef     structure
{
Int *       node
Int         nNode

} EDGE, *PEDGE
```

(a)

```
Typedef     structure
{
EDGE *      inBnd
Int         nInBnd

} BND, *PBND
```

(b)

Figure 6-5 Used data structures in fixing of constraint boundary: **(a)** Edge is defined by list of nodal indices, **(b)** List of constraint boundaries

6.1.4 Merging the Sub-domain Meshes

In merging process the interface wall elements are simply added with the neighbor region elements list to form a complete mesh. To merge the interface wall triangles TLIW with TLP1 and TLP2 few of the triangles from the sub region triangulation have to be deleted. As shown in figure 6.6, the deleted triangles are marked in green in this example mesh problem. These triangles are introduced during the triangulation of sub regions. Since each sub-region is independently triangulated, the GPU-DT algorithm produces the triangulation of the convex-hull of that point-set. Due to this, finding the convex-hull of the sub-region is easy. In the work of Chen [CHEN ET AL., 2001], an efficient algorithm is introduced to remove these superfluous triangles as shown in the algorithm 6.7.

This algorithm is modified to include the task of updating the new neighbor relations between newly introduced interface wall triangles and sub-region triangles. The new interface wall elements are combined into the triangle list of the final mesh. For an efficient memory usage, the deleted triangle memory spaces are used to insert new elements.

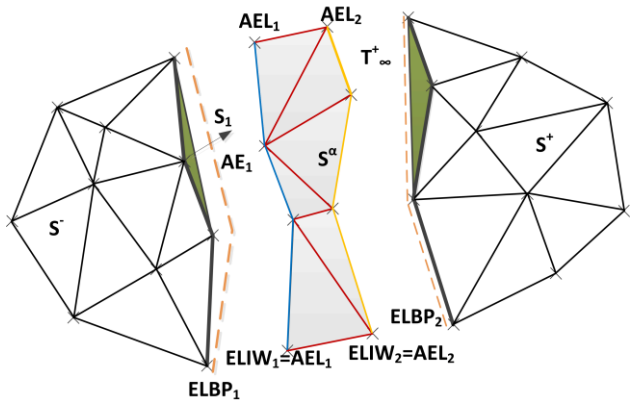

Figure 6-6 Merging subdomain meshes using interface triangulation

Delaunay triangulation using incremental construction techniques is slower and has a low efficiency compared to other techniques [CIGNONI, 1994]. The worst case complexities of this technique are $O(n^2)$ and $O(n^3)$ for 2D and 3D problems, respectively. The time-consuming part of these algorithms are the *make_simplex()* function and active edge list management. For each active edge, *make_simplex()* searches through all the points to calculate the minimum Delaunay distance $dd(e,p)$. In the active edge list management, the operation on the list requires a time which is proportional to the number of edges currently stored in the list. To overcome this bottleneck, two speeding techniques are proposed by both of the authors Cignoni and Chen [CIGNONI, 1994, CHEN ET AL., 2001]. They are the uniform gridding technique and the hash table usage for active edge list operation.

Algorithm 6.7: *merging subdomain meshes with interface wall meshes*

Function: mergeMesh(list_s*: list of submeshes,* ***list_IW****: list of interface wall mesh)* ***s****: final mesh*

Input: *list_s: list of submeshes, list_IW: list of interface wall mesh*

Output: *s: final mesh*

begin

for *each interface wall Iwi*

Get the triangle list of point set 1 (P1) (left side of partition line) TLP1 and point set 2(P2) (right side of partition line) TLP2. From the active edge list (AEL1 and AEL2) create the edge list of interface wall ELIW1 = AEL1 and ELIW2 = AEL2. Get the triangle list of interface wall TLIW

repeat
Create the edge list of sub region boundary by traversing along the convex-hull of point set. ELBP1 and ELBP2 are the edge list of boundary points on interface region in point set one and two respectively

Delete the duplicated edges in ELIW1 and ELBP1
repeat
Mark the last edge in ELBP1 as active edge AE1

Mark the triangle S1 shared with AE1 as deleted (convex-hull edge)

For each edge of triangle S1, search for the duplication in ELIW1,
if it is duplicated delete that edge

If that edge is not found in the ELIW1, then update the ELBP1 with that edge

until *ELIW1 or the TLP1becomes empty*

until *to be fixed other side of interface wall (ELIW2 and ELBP2)*

Combine the triangle list into TLP = TLP1 +TLP2

Delete the marked triangles from the TLP and insert the equal number of triangles from TLIW into list TLP. Update the neighbor element relations. Add the rest of the triangles in TLIW at the end of the list

end for

end

6.2 CUDA Implementation Details of Multi GPU-DT

6.2.1 Necessary CUDA Computing Concepts Related to Implementation

In this section few important parallel computing concepts, related to the implementation of the GPU-DT algorithm on multiple GPUs are discussed. The CUDA API is a continuously developing API technique. Currently available modern graphic hardware allows multiple device parallelism. To improve the parallel performance there are optimized memory access concepts in the CUDA technology, which are also discussed in following sections.

6.2.1.1 Coalesced Memory Access

Modern DRAMs (global memories) on GPUs and the current CUDA technology allow programmers to achieve highly efficient global memory access. This is done by organizing the memory access of threads using a favorable pattern [KIRK AND HWU, 2010]. By organizing all the threads in a warp to access consecutive memory locations in the global memory an optimum memory access can be achieved. This type memory access pattern is called coalescing memory access. DRAM-bursting is a hardware feature in which each address space is partitioned into burst-size of sections. That means whenever a memory location is accessed, all other locations in the same burst section are also delivered to the processor. As shown in figure 6.7 (a) if memory access is partitioned by burst section sizes, this is called coalesced memory access. If memory loading accesses different burst sections partially, then this access is called un-coalesced memory access (As shown in figure 6.7 (b)). In CUDA programming, for efficient memory handling, programmers should use coalesced global memory access.

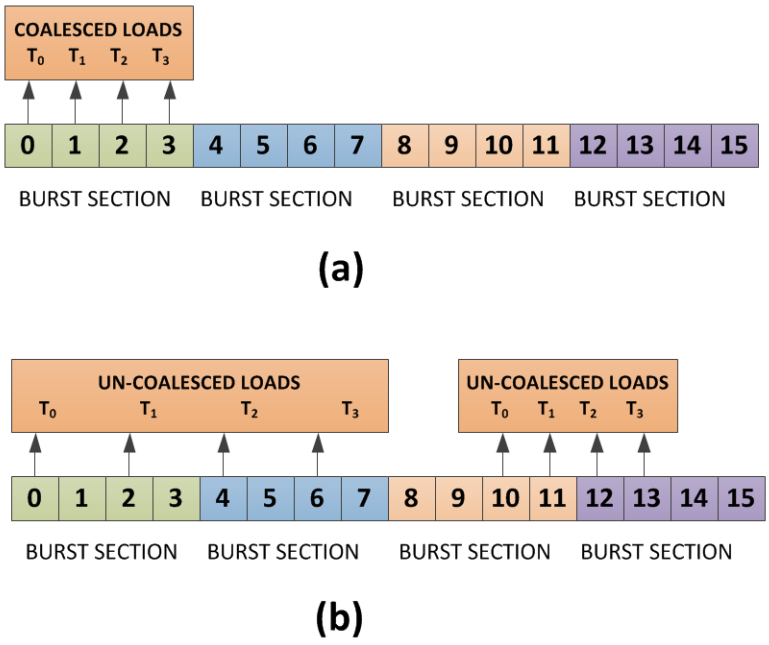

Figure 6-7 (a) Coalesced memory access, **(b)** Non-coalesced memory access

6.2.1.2 Usage of Pinned Memory Allocation

By default, virtual memory management in modern computers uses pageable memory resulting in paging in and paging out [KIRK AND HWU, 2010]. Since physical memory is limited, the paging-in and the paging-out of physical addresses are automatically established by the CPU. In modern graphic cards, direct memory access (DMA) hardware facilitates the

memory transfer. DMA uses physical addresses which are translated from the real source address with the help of the operating system. As shown in the figure 6.8 (a), a two-step process by maintaining a pinned memory buffer is compulsory to avoid the overwriting risk into the pageable memory. For a secure memory transfer, first data addresses are translated into a pinned memory buffer (page-locked memory) then the data is copied by the DMA into the device memory.

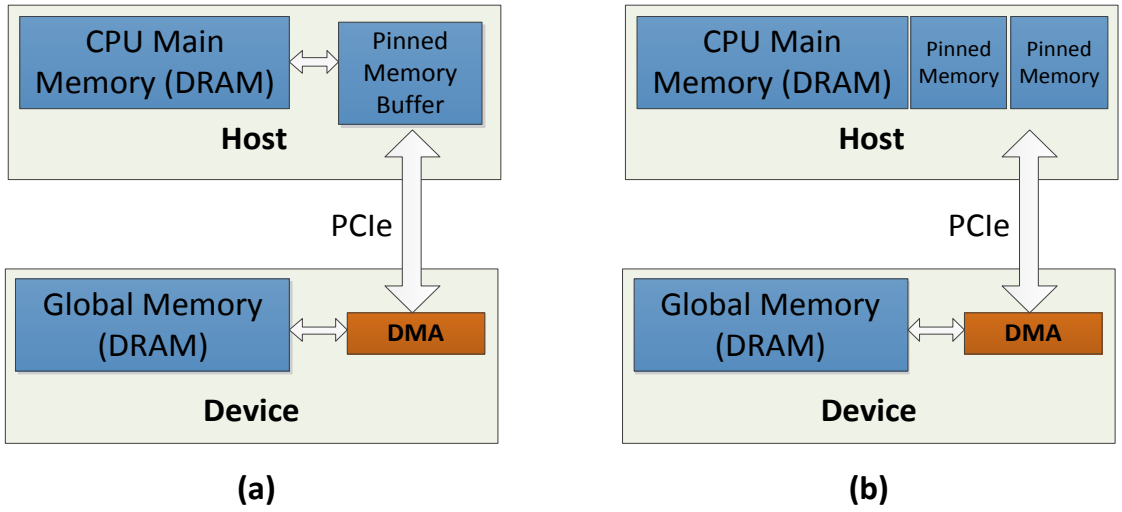

Figure 6-8 (a) Pageable data transfer, **(b)** Pinned memory data transfer

In the pinned memory data transfer model, as shown in figure 6.8 (b), page-locked memory space is explicitly allocated in the host at the start of the computation. In this case the DMA can directly access the data source in pinned memory space without CPU intervention for any synchronization. This transfer is asynchronous and faster than the pageable data transfer.

In the pageable data transfer, since the PCIe memory bandwidth is limited, the two steps of memory copying and synchronization make the overall data transfer slower. But the pinned data transfer model, due to direct copying of data, is around two times faster than the previous approach. This process allows asynchronous CUDA computation. At the same time, the programmer has to compromise with large memory size requirements. That means over-allocating pinned memory will reduce the overall performance of programs because pinned memory consumes space during the CUDA runtime so that host memory will be limited for other operating system activities [RUETSCH, 2012].

6.2.1.3 Concurrent Kernel Execution

Besides taking advantage of the data parallel capability of GPUs using multiple threads, computing using concurrent kernels is another advanced stage of parallelism using GPUs. The

current development in graphic card processors and the supporting API technology allows an advanced level of parallelism using the GPUs. The CUDA hardware architecture called Fermi (or later than this technology) devices support multiple kernel execution concurrently. The CUDA API releases with compute capability 2.0 or more support the concurrent execution of up to 16 CUDA kernels on GPUs [KIRK AND HWU, 2010]. A simultaneous computation on the CPU during kernel execution is possible. For data transfer between the device and the host, simultaneous memory copying is possible using the function *cudaMemcpyAsync()* instead of *cudaMemcpy()*, but it currently supports only different directional memory copying. That means two concurrent memory copying actions are possible in opposite directions from the device to the host and from the host to the device.

6.2.1.4 CUDA Streams

The streams technology available in the latest CUDA architecture supports the parallel execution of multiple kernels and parallel *cudaMemcpy()*. This was made possible because of the asynchronous data transfer and the kernel execution capability of modern hardware. A stream in CUDA programming means the queue of operations which are aggregations of kernel calls and *cudaMemcpy()* (as shown in figure 6.9 (a)). This is analogous to task parallelism [KIRK AND HWU, 2010].

The sequence of operations in different tasks can be called using different streams. The effective usage of stream techniques in concurrent CUDA programming can be used either by running different streams concurrently or by letting operations from different streams to be interleaved [NVIDIA, 2012]. In reality the kernel and memory copy calls are not called in different stream queues as shown in figure 6.9 (b). The kernel engine and copy engine separately handle all the kernel and copy instructions. In general, the kernel waits until necessary data to be transferred and copy operations from device to host wait until the kernel operation ends.

As shown in figure 6.10 an example problem is compared for overlapping concurrent kernel execution and serial execution. In a serial run, single stream takes responsibility for both *cudaMemcpyAsync()* operations in both directions from the host to the device (H2D) and from the device to the host (D2H) and kernel execution. In the example of 2-way concurrency two kernel executions are divided into 4 kernels (K1, K2, K3 and K4) and copying D2H is divided into 4 parts. Four streams take the responsibility to run simultaneously. According to Nvidia's report, this variant of concurrent implementation can give 1.33 times improved performance

compared to serial one. The three-way concurrency example utilizes the possible three parallel operations such as memory copying H2D, D2H and kernel execution. All three processes are divided into four parts and executed using four different streams. This variant gives even more performance. There are more possible variants of concurrent kernel executions as long as it does not exceed the limitations of hardware and API.

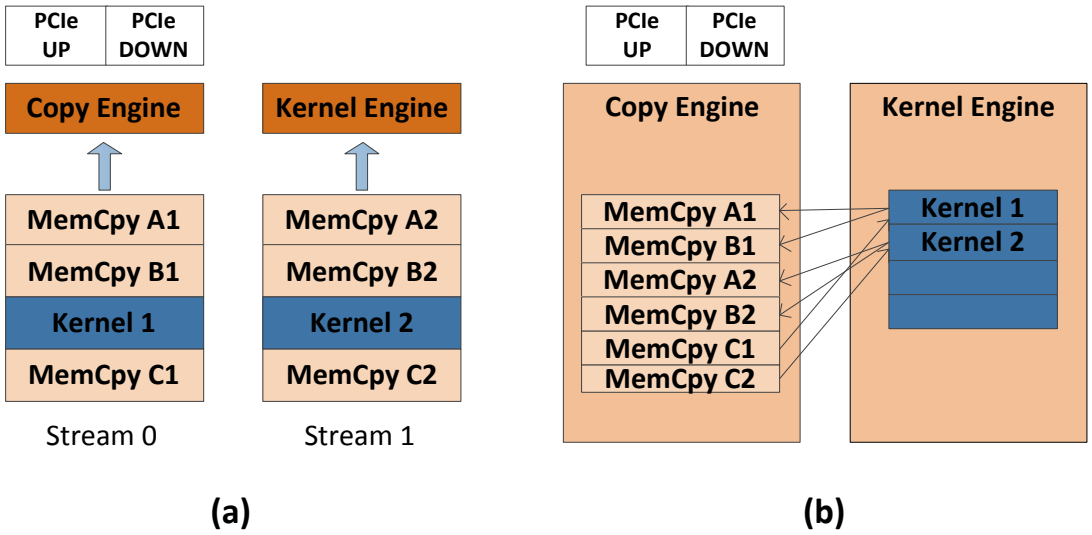

Figure 6-9 (a) Conceptual view of data flows in streams, **(b)** Real nature of streams

Note about this type of concurrent CUDA executions using multiple streams, that to experience a real performance improvement, the program should be possible to partition without interdependencies. That means data should also be able to be divided into independent partitions and kernel operations should not depend on other kernel operations or there is no need for intermediate communication requirements between device and host.

In general CUDA programming, if there are no streams defined, default stream 0 is used to queue up all the operations. With the default stream by defining *cudaDeviceSynchronize()* before and after every CUDA operation, a program is completely synchronous with respect to the host and the device.

There are a few conditions which can be summarized for concurrency in CUDA computing:

1. CUDA operations must be queued in different streams.
2. For concurrent memory copy using *cudaMemcpyAsync()* “pinned” memory is used to allocate memory in host. This is a page-locked memory allocated using *cudaMallocHost()* or *cudaHostAlloc().* This is also called non-blocking data transfer.
3. Enough device resources (memory units) and many independent data units are available.

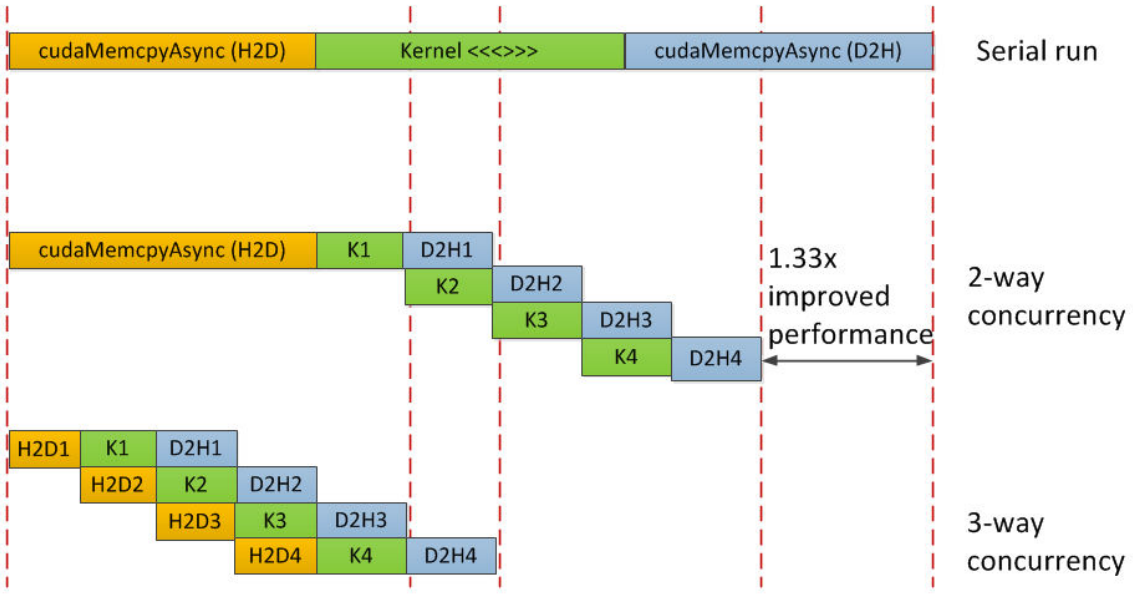

Figure 6-10 Different variants of CUDA concurrency (based on [NVIDIA2, 2013])

6.2.2 Multi GPU Version of PBA Algorithm

The original GPU-DT algorithm employed the parallel banding algorithm to generate the digital Voronoi diagram. Construction of Voronoi diagram is an important step in computation of Delaunay triangulation. Computation of digital Voronoi diagram is performed completely on GPU. Therefore, as a first step of multi GPU-DT parallelization approach, the PBA algorithm is extended to support on multiple GPUs system.

Algorithm 6.8: standard parallel banding algorithm using single GPU

Function: PBA_singleGPU(**P**: point set) **v**: discrete Voronoi diagram

Input: P: point set

Output: v: discrete Voronoi diagram

begin

```
AllocateCUDAMem(P)
CUDAMemCopy(P,HostToDevice)
Phase1-kernel ()   // find the nearest site of a pixel in a row
Phase2-kernel()    // determine the proximate sites of each column
Phase3-kernel().   // color the columns using proximate sites
CUDAMemCopy(P,DeviceToHost)
```

end

As shown in the algorithm 6.8 the single GPU parallel implementation of the PBA algorithm contains these main steps (as discussed in section 4.2.2). At the start, the input data (here given point-set *P*) is transferred to the GPU memory from the host memory. To transfer data from the host to the device memory space is allocated in the device (*cudaMemalloc()*) then data is copied to the device (*cudaMemcpy()*). After data is transferred to the device, computation of the digital Voronoi diagram is computed on the device using the kernel functions *phase1-kernel(), phase2-kernel()* and *phase3-kernel()*. At the end of the computation, results are transferred back to the host using *cudaMemcpy()*.

The multi GPU version of this algorithm is modified as shown in the algorithm 6.9. According to the number of available GPUs, data is partitioned into a number of partitions. In this implementation two partitions are used to fully utilize the computing system available with two GPUs. At the start, the two partitions of input data are transferred into the device memory of each GPU. By setting the device using the CUDA function *cudaSetDevice()* to a targeted GPU, memory allocation and memory copy functions are called within a for-loop. After the data transfer, computations are performed on each device in parallel. Each kernel function is called within a for-loop which runs a number of GPUs times. At the end of the computation in devices, the results are transferred back into the host memory. Figure 6.11 shows the CUDA runtime profile of a single GPU execution of the PBA algorithm. It is noticeable that the major part of total GPU runtime is consumed for data transfer between the host and the device.

Algorithm 6.9: *upgraded implementation of PBA algorithm for multiple GPUs*

Function: PBA_multiGPU (list_P *: list of point sets****) list_v*** *: list of Voronoi diagrams*

Input: list_P *: list of point sets*

Output: list_v *: list of Voronoi diagrams*

```
begin
    for each device (dev)
        AllocateCUDAMem(P_dev)
        CUDAMemCopy(P_dev,H2D)
    end for
    for each device(dev)
        Phase1-kernel (dev)
    end for
    for each device(dev)
        Phase2-kernel(dev)
```

```
    end for
    for each device(dev)
        Phase3-kernel(dev)
    end for
    for each device (dev)
        CUDAMemCopy(P_dev,D2H)
    end for
end
```

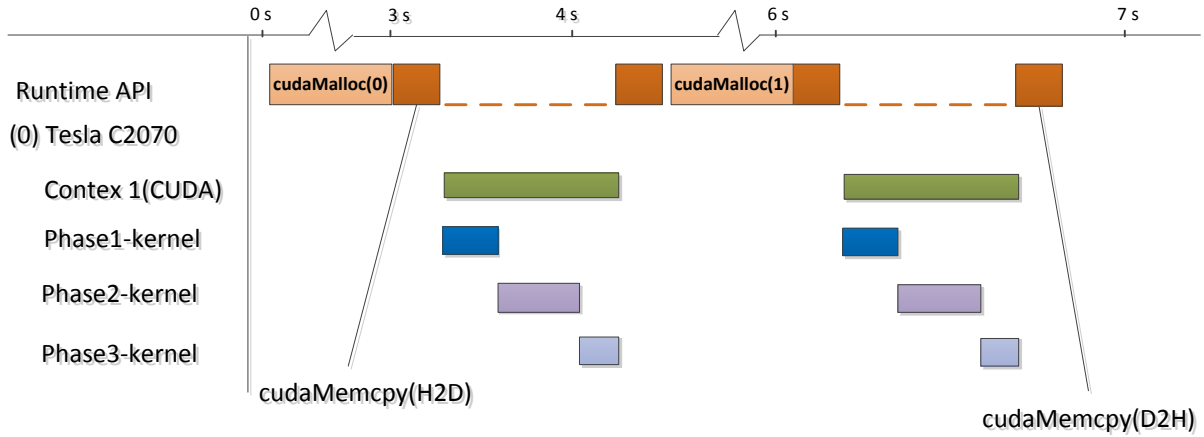

Figure 6-11 CUDA runtime profile of single GPU PBA implementation

As shown in the figure 6.12, the runtime CUDA computation profile shows the parallel execution of kernel functions. The parallelization technique used in this implementation distributes the computation task into different GPUs and lets them execute in parallel. Distributed tasks have an almost equal work load of computation. As shown in the multi GPU runtime profile, kernel function executions are performed in parallel using two default CUDA streams to handle two different GPUs individually. In this type of parallelization using multiple GPUs, memory transfer between the host and the device is only a serial execution. The CUDA functions involved in data transfer are *cudaMemalloc()* and *cudaMemcpy()*. They are called by the CUDA runtime API as a serial operation calling one device after other. Due to this inefficient data transfer, the overall performance of the multi-GPU PBA algorithm is compromised.

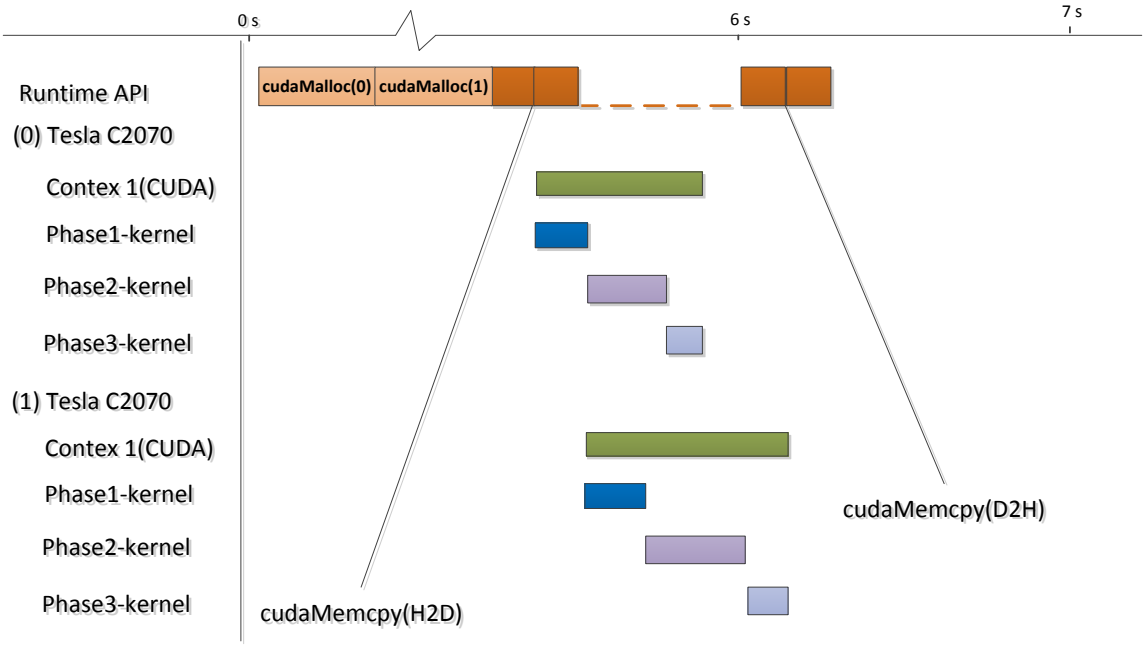

Figure 6-12 CUDA runtime profile of multi-GPU PBA implementation

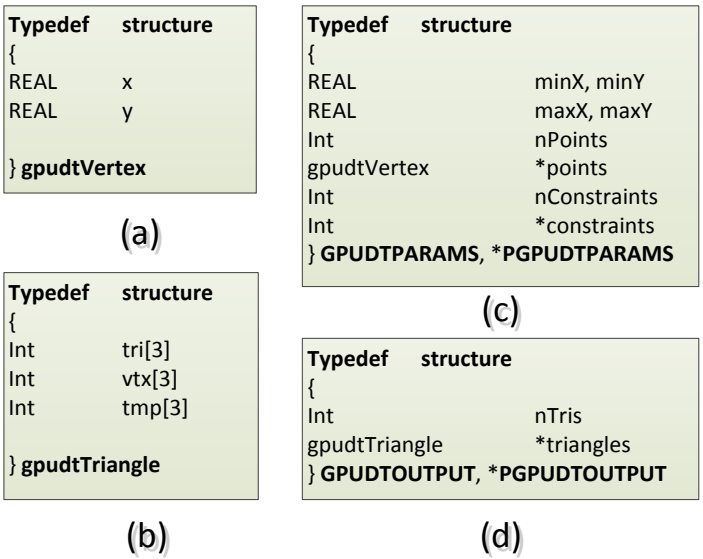

Figure 6-13 Main data structures used in the GPU-DT algorithm, representation of **(a)** Vertex, **(b)** Triangle, **(c)** All input parameters of compute GPU-DT routine, **(d)** Out parameters of GPU-DT routine

6.2.3 CUDA implementation of GPU-DT

At the beginning of a GPU-DT execution, input data is generated or imported into the allocated host memory. The main two input and output parameters are *GPUDTPARAMS* and *GPUDTOUTPUT*, respectively. In figure 6.13 the details of these data structures are shown. The basic data structures *gpudtVertex* and *gpudtTriangle* are used to represent the vertex and

triangle data, respectively. After the host data initialization, the CUDA implementation part begins. The computation flow of the CUDA implementation is given in the abstract pseudocode of algorithm 6.10.

Algorithm 6.10: *CUDA implementation steps of GPU-DT*

Function: GPU-DT *(**Pin**: GPUDTPARAMS)* ***Pout**: GPUDTOUTPUT*

Input: *Pin: GPUDTPARAMS*

Output: *Pout: GPUDTOUTPUT*

begin

```
    cudaAllocation()
    cudaInitialization()
    cudaDiscreteVoronoiDiagram()
    cudaReconstruction()
    cudaShifting()
    cudaMissing()
    cudaFixBoundary()
    EdgeFlipping()
    return Pout
```

end

In the functions *cudaAllocation()* and *cudaInitialization()*, the input and output data is allocated into the device memory and copied there. The necessary runtime data variables are also allocated into the device memory. The implementation details of *cudaDiscreteVoronoiDiagram()* have already been discussed in section 6.2.1. The function *cudaReconstruction()* computes the triangles from the discrete Voronoi diagram. Each CUDA thread takes care of one row of texture memory and calculates the Voronoi vertices along that row. Then another kernel function identifies the triangles corresponding to those Voronoi vertices.

In the *cudaShifting()* function, there are two main process steps as shown in algorithm 6.11. First, check through all the points in parallel and identify the good-case points (shiftable without any conflict). The good-case points are shifted immediately to the original coordinates. During the CUDA kernel execution, when a particular thread tries to shift a point, the point index should be lower than the other neighbor points to be shifted in this round. Otherwise, shifting of this point is delayed for the next round.

Algorithm 6.11: *CUDA implementation of shifting process*

```
Function: cudaShifting()
begin
    active_points = npoints
    repeat
        kernel_shift_good case points()
        kernel_mark_bad case points()
    until (active_points = 0)

    active_points = number of bad case points

    repeat
        kernel-mark points to be deleted in current round
        kernel-mark the triangles in the fan of deleting point
        kernel-patch the star-hole of deleted triangles
        kernel-update the neighbor relations of new triangles
    until (active_points = 0)
end
```

After that, the rest of the points marked as bad case points are deleted in multiple rounds. To avoid the conflict between threads dealing with neighbor points, deletions of points are possible in multiple rounds until all the points are deleted. In each round of deletion of bad case points, deleting triangles in the triangle fan of that particular point are marked to be deleted. Then the polygonal hole of deleting triangles is patched with new triangles. Finally, new triangles are inserted into the deleted triangle slots and neighbor relations are updated.

Algorithm 6.12: *CUDA implementation of missing points insertion*

```
Function: cudaMissing()
begin
    active_points = number of missing points
    repeat
        Kernel_locateTriangles_contain_missing_points
        KernelPreprocessTriangles
        Kernel_FixVertexArrayMissing
        Kernel_insertMissingSites
        Kernel_updateMissingTriangleLinks
    until (active_points=0)
end
```

After the shifting phase, a phase of inserting the missing points is carried out fully in parallel. As shown in algorithm 6.12, abstract details of inserting the missing point step are given. To employ the CUDA parallelization without conflicts between threads, this computation is only possible in several rounds of insertion until all the missing points are inserted. At the beginning of each round, the number of existing missing points is decided and the triangles which contain those missing points are identified in parallel. After that the possible missing points are identified to be inserted in that particular round without any conflicts. Then those triangles are marked to be deleted and the vertex array is updated with existing triangles. Next, the missing point is inserted and three new triangles are introduced. Finally, the neighbor relations are updated to each newly introduced triangle.

The *cudaFixBoundary()* is another parallel execution step in GPU. This process also performs the task in multiple rounds to avoid thread conflicts. In each round, detecting the boundary sites, the necessary boundary triangles are deleted, the boundary hole of deleted triangles are patched and finally the triangle links of boundary triangles are updated. The final phase of GPU-DT is *edgeFlipping()* which does a recursive flipping of necessary edges by going through all the edges in the complete triangulations.

6.2.4 CUDA Implementation of Multi GPU-DT

In this thesis, the GPU-DT algorithm is partially implemented to be compatible on multiple GPUs system. As a part of the algorithm, computation of digital Voronoi diagram is upgraded to support parallel computation on multiple GPUs. The details of the multi GPU PBA algorithm are discussed in the section 6.2.2 and its performance study discussed in section 7.2. In upgrading the existing GPU-DT code as compatible to multiple GPUs, there are significant challenges in modifying some of the computing steps with the currently existing CUDA technology. As an example, *cudaReconstruction(), cudaMissing()* and *cudaShifting()* computation routines do not perform in parallel on multiple GPUs due to intensive communication overhead between host and device and synchronization requirements. To see the CUDA computation flow of the GPU-DT algorithm, a runtime profile is shown in figure 6.14. It is straight forward that there are frequent communication request between the device and the host during kernel execution. These data transfers between the host and device negatively influence the overall CUDA performance.

To utilize multiple GPUs to run kernel functions concurrently in different devices, intermediate data transfer should be avoided. The parallelization concept is utilized as shown

in algorithm 6.13. Kernel runs are called within a loop by assigning a particular device index. In this way of programming, kernels are computed in parallel in different devices. When a memory transfer is required between the host and a device, then the computation is synchronized. Since these computation steps invoke frequent memory transfers during kernel execution, parallel performance is fully affected and overall performance becomes worse than the single GPU performance. These types of computations would be possible to execute on multiple devices, when CUDA hardware and API technology permits the asynchronous and bidirectional data transfer between hosts and devices in the future.

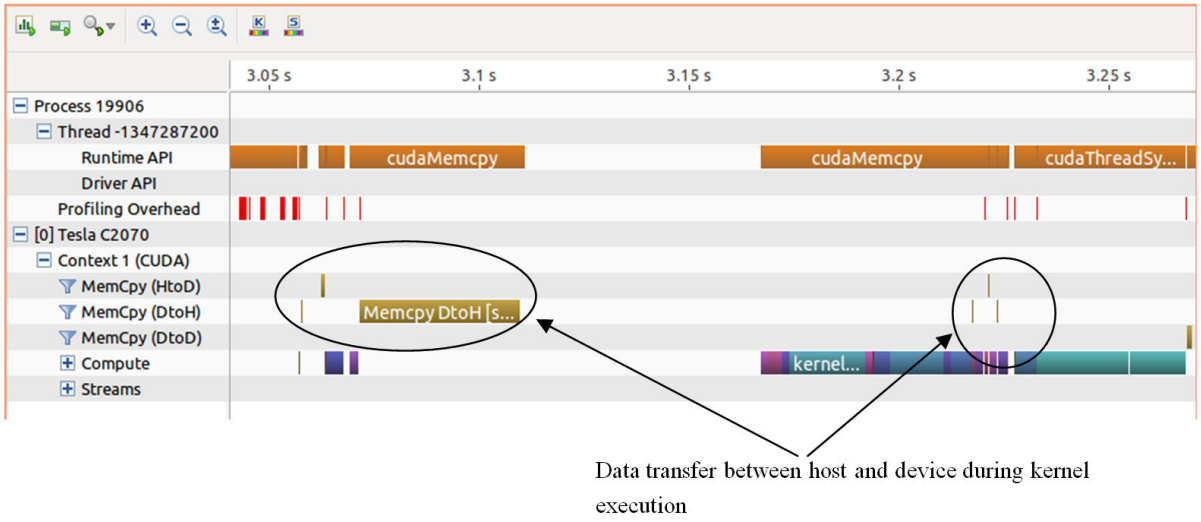

Figure 6-14 CUDA runtime profile of a typical GPU-DT computation

---

Algorithm 6.13: *Sample concurrent kernel invocation on multiple GPUs*

```
begin
    for each device (dev)
        cudaSetDevice(dev)
        kernel1 <<< >>> ()
    end for
    for each device (dev)
        cudaSetDevice(dev)
        kernel2 <<< >>> ()
    end for
end
```

Chapter 7

7 Case Studies & Results

In this chapter different case studies are presented to support the multi GPU-DT meshing concept. The computer system used for all the GPU implementations is equipped with an Intel Xeon X5650@ 2.67GHz CPU processor and two Tesla GPU devices (C2070 6GB). The RAM size of the CPU system (host memory) is 48 GB. The global memory size of each device is 6 GB. The operating system used for most of the computation is 64 bit Ubuntu 11.10 (Linux 3.0.0-25 generic x86-64).

Initially, a performance comparison of the GPU-DT implementation with the best CPU-based Delaunay triangulation implementation *Triangle* [SHEWCHUK, 1996] is studied. The results obtained with the above GPU computing system are presented in section 7.1. Section 7.2 presents the variation of the number of missing points with respect to the number of partitions and allocated texture sizes. Parallel performance of digital Voronoi diagram computation using the PBA algorithm is studied in section 7.3. In this case study the upgraded multi GPU PBA algorithm is compared with the existing single GPU PBA algorithm. Section 7.4 demonstrates the capability of the upgraded GPU-DT algorithm as a Delaunay mesh generator. A simple tool-model is used to demonstrate the functionality of the upgraded mesh generator. Finally, section 7.5 compares the computational work load of the interface wall triangulation with respect to the requirement of the overall triangulation.

7.1 Comparison of GPU-DT with CPU-DT

As a preliminary study, the existing single GPU version of a GPU-DT implementation is tested against the fastest CPU based triangulation implementation (*Triangle*). This test result shows that the required triangulation time using *Triangle* (CPU-DT) is less than the time using GPU-DT for a point-set with a small number of points (less than 10,000 points). For a large point-set (more than 10,000 points), GPU-DT is faster than the CPU-DT. As shown in the figure 7.1, computation time is compared between both triangulation implementations. With the above mentioned computing system, an improved performance of GPU-DT over CPU-DT is noticed up to 7 million points. The used texture resolution in the GPU in all test cases is kept as 2048 x 2048 texture array.

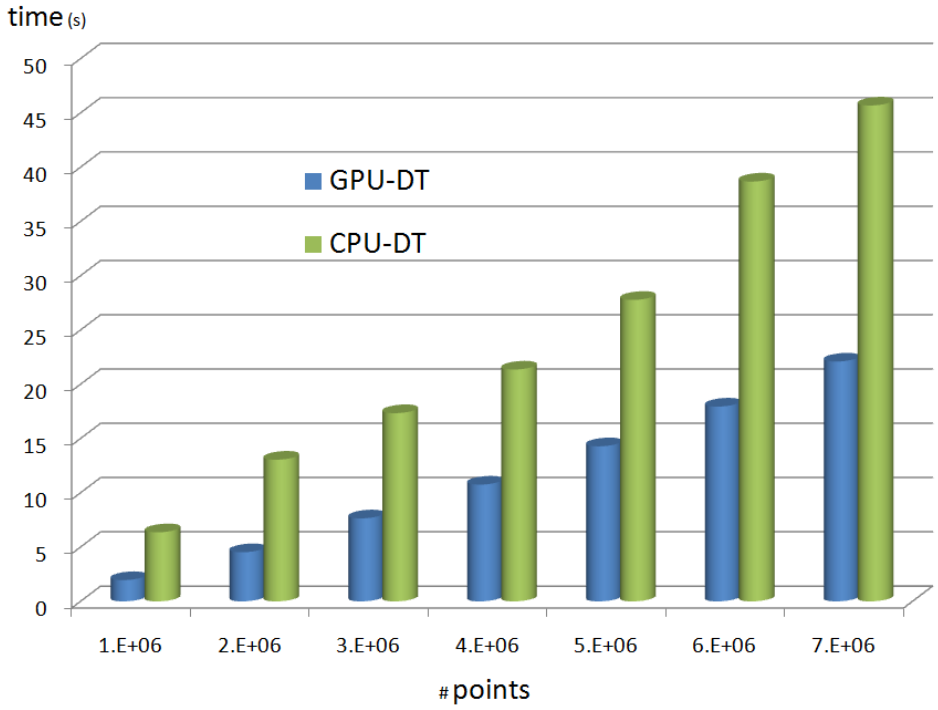

Figure 7-1 Comparison of computation time of GPU-DT and CPU-DT

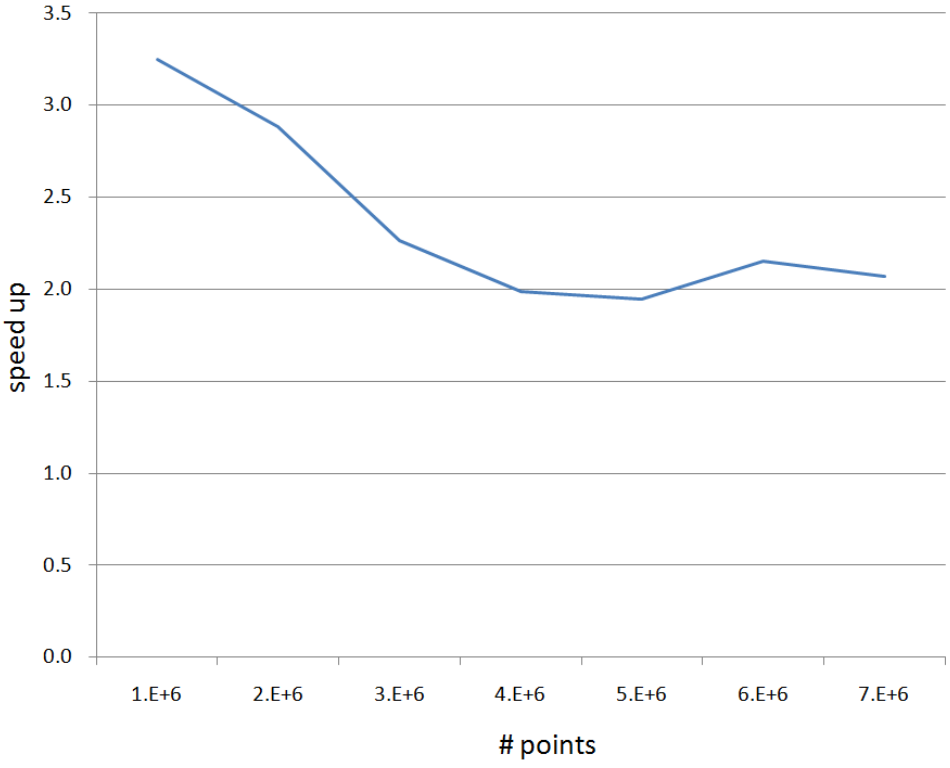

Figure 7-2 Comparison of speed-up of GPU-DT over CPU-DT

In figure 7.2, the speed-up of GPU-DT against CPU-DT is plotted. For a large point-set, a speed-up of more than two is realized by GPU-DT. It is noted that the GPU-DT implementation experiences a speed up of more than ten according to the authors of the GPU-DT implementation [QI ET AL., 2013]. This computing performance depends totally on the distribution of the used point-set and the computing power of the GPU and the CPU.

7.2 Parallel Improvement by Partitioning

The initial phase of the GPU-DT calculation is the construction of a digital Voronoi diagram. The original algorithm was implemented to suit parallel computation using only a single graphic card (GPU). This calculation contains few kernel functions, which are executed on the GPU. The existing GPU-DT parallelization concept is data parallelism achieved by executing tasks using several hundred parallel CUDA threads.

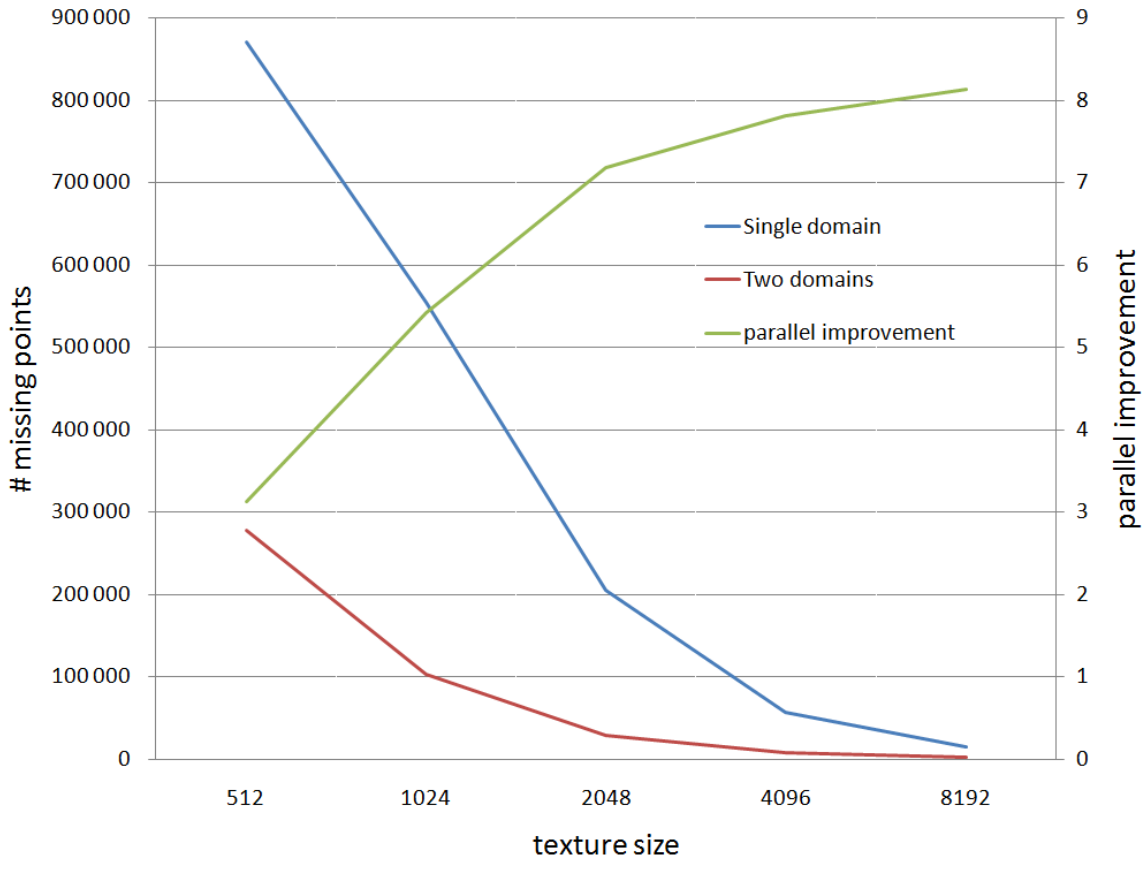

Figure 7-3 Changes in the number of missing points for different texture sizes for a point-set of one million points

Table 7.1 Number of missing points with respect to texture size and number of partitions

Texture size	#Missing points (1 million points)		
	1 GPU	2 GPUs	Parallel improvement
512 x 512	870 271	278 225	3
1024 x 1024	555 081	102 349	5
2048 x 2048	204 888	28 513	7
4096 x 4096	57 123	7 304	8
8192 x 8192	14 843	1 825	8

Texture size	#Missing points (2 million points)		
	1 GPU	2 GPUs	Parallel improvement
512 x 512	1 900 000	746 219	3
1024 x 1024	1 489 622	356 736	4
2048 x 2048	712 704	110 307	6
4096 x 4096	220 343	29 061	8
8192 x 8192	58 619	7 418	8

Texture size	#Missing points (4 million points)		
	1 GPU	2 GPUs	Parallel improvement
512 x 512	3 870 205	1 741 012	2
1024 x 1024	3 478 566	1 110 267	3
2048 x 2048	2 218 275	410 353	5
4096 x 4096	819 656	114 702	7
8192 x 8192	230 029	29 767	8

The computation of a digital Voronoi diagram for a given point-set is divided into two domains with nearly equal number of points in each domain. The data of each domain is given into two separate GPUs and computed in parallel instead of giving the entire point-set into one GPU. Understandably, increasing the number of missing points will lead to an increase in total GPU-DT triangulation time. The number of missing points depends on the allocated texture resolution (texture size), the distribution of point coordinates and the total number of

points. The point distribution used in this study is a uniform random distribution. The test is performed for a given number of points and computed comparing different texture space allocation with a single GPU and double GPUs.

As shown in the table 7.1 the number of missing points details for different PBA test cases are recorded. For a point-set size of 1, 2 and 4 million points, single GPU and double GPUs version of PBA tests are performed. The numbers of missing points for both versions of PBA with respect to different texture sizes are compared. The possible parallel improvement of the multi GPU version is calculated by considering the number of missing points. The number of missing points on multi GPU implementations is several times less than the number on the single GPU version. The ratio of number of missing points between both versions are recorded as parallel improvement of multi GPU version over single GPU one.

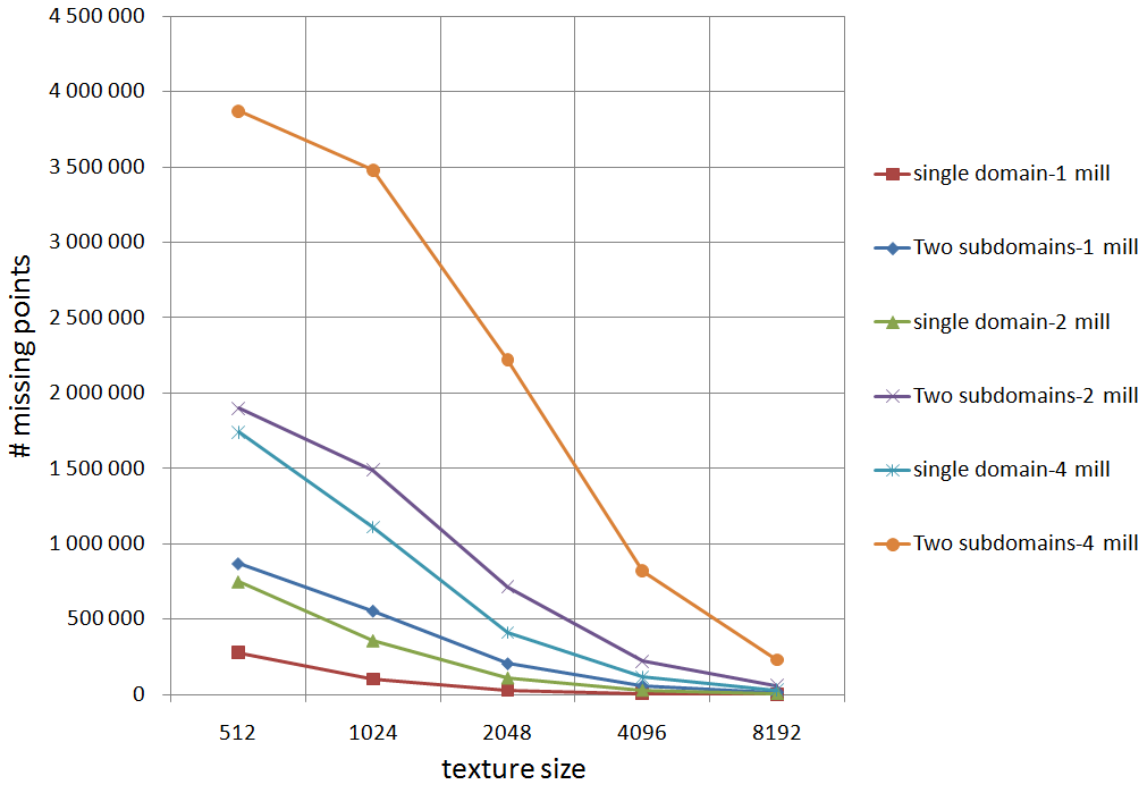

Figure 7-4 Changes in the number of missing points for different texture sizes for 1, 2 and 4 million points

In figure 7.3, changes in number of missing points due to different texture sizes are plotted for two PBA versions. The number of missing points is obviously larger for small texture memory allocations. The parallel improvement of the multi GPU PBA version against the

single GPU PBA version is also plotted. The noticed parallel improvements for smaller texture sizes are also smaller than the larger texture sizes.

Figure 7.4 demonstrates the trends in changes of the number of missing points for different point size with respect to different texture sizes. From all these test cases, one can notice that decomposing the points as multiple partitions and processing them on different GPUs can reduce the number of missing points. This will lead to a reduction in the required computation time for the insertion of missing points. Ultimately, this would positively influence the overall computational time of the GPU-DT computation on multiple GPUs.

7.3 Multi GPU PBA Results

This section presents a comparison study between the single and multiple GPU version of the PBA algorithm. Digital Voronoi diagrams are computed using both versions of the PBA algorithms for different sizes of point-sets. As shown in figure 7.5, the CUDA computation profile of the single and multi GPU PBA algorithms are recorded. The total computation time ($\Delta t_1, \Delta t_2$) of the CUDA kernels for the single GPU version and the total kernel execution time (Δt_3) in the multi GPU version are recorded for all the test cases. The required total CUDA computation time is also recorded in both computations. From the total runtimes of both versions, the improved runtime (Δt) is computed. Several test cases are considered by changing the size of the point-set and texture.

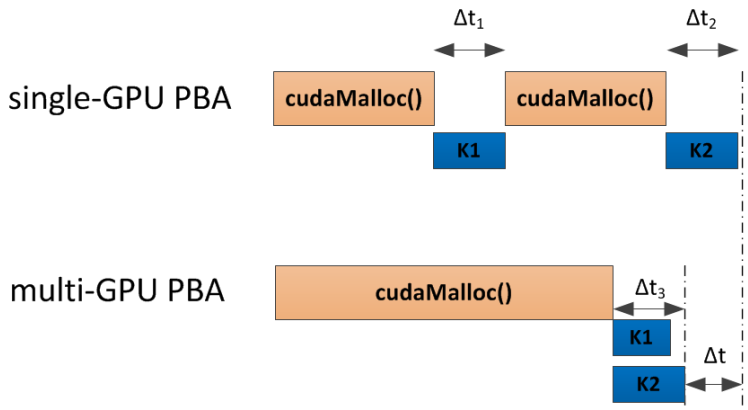

Figure 7-5 The CUDA computation time profile of single and multi GPU version of PBA algorithms

As shown in the CUDA profile, there is no significant gain in the time needed for memory allocation and memory transfer in the multi GPU processes. With the currently available CUDA technology the multi GPU PBA computation only get advantages by running kernels in parallel, since the memory transfer is still sequential. As shown in figure 7.6, for one of the

selected test cases the total CUDA computation time of single and multi GPU PBA algorithms are plotted with respect to point sizes from one to five million. One can notice that the total computation time of the single GPU version is more than the multi GPU one.

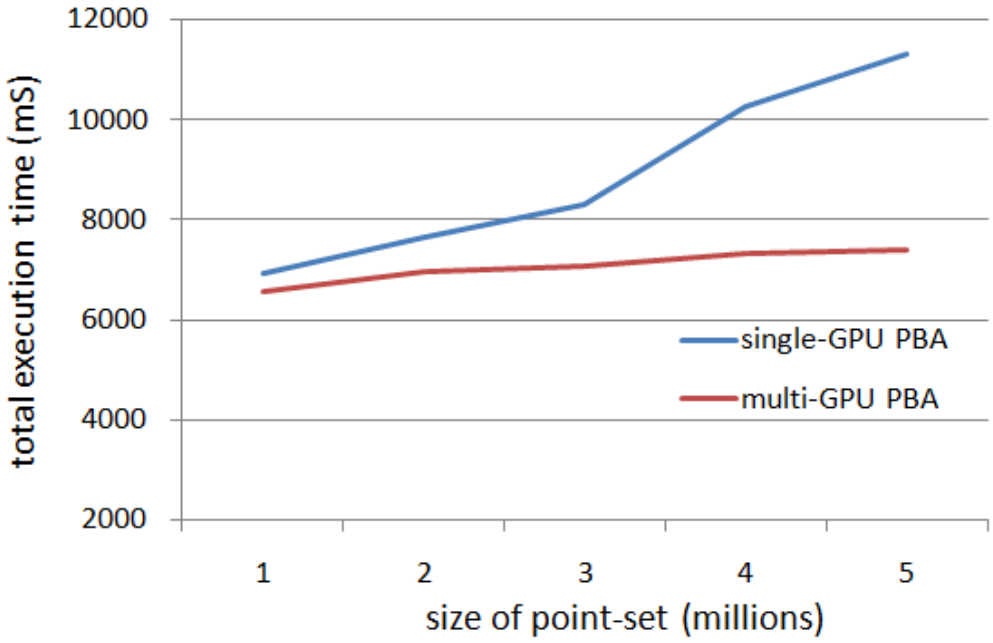

Figure 7-6 Comparison of the computation time for single and double GPU versions of PBA algorithms with respect to different point-set sizes

In figure 7.7, the speed-up of multi GPU PBA algorithms over the existing single GPU version is plotted. In calculating the parallel performance, the speed-up is computed by considering two different computation times of both versions of PBA. By considering the total runtime of PBA computation, the achieved speed-ups are less than 1.5 (blue curve). The red curve represents the speed-up by only considering the required computational time of kernel executions. These speed-ups reached around two, for a two GPUs system. From this, one can conclude that the CUDA kernel executions in the multi GPU PBA algorithm run in parallel. Based on experience in these experiments, it should be noted that these CUDA parallel performances are not always consistent. Here, the successful cases which show better parallel performances have been reported. From these results one can conclude that the PBA algorithm is highly parallelizable using multiple GPUs by porting the existing single GPU version. Still, there are improvements to be expected in the CUDA hardware and API technology to overcome the existing bottle necks in communication and data transfer between

the device and the host. When these technology limitations are overcome, the PBA algorithm will be highly scalable onto multiple GPUs.

As a concluding remark, since the PBA computation is a part of the GPU-DT algorithm, the parallel efficiency achieved in a multi GPU PBA computation will improve the overall performance of the multi GPU-DT. Another important positive remark is the parallel improvement achieved by considering the number of missing points. By partitioning the triangulation task as sub-problems and computing each one on different GPU, the required computation time to insert the missing points is dramatically reduced. This significantly improves the overall performance of a multi GPU-DT computation.

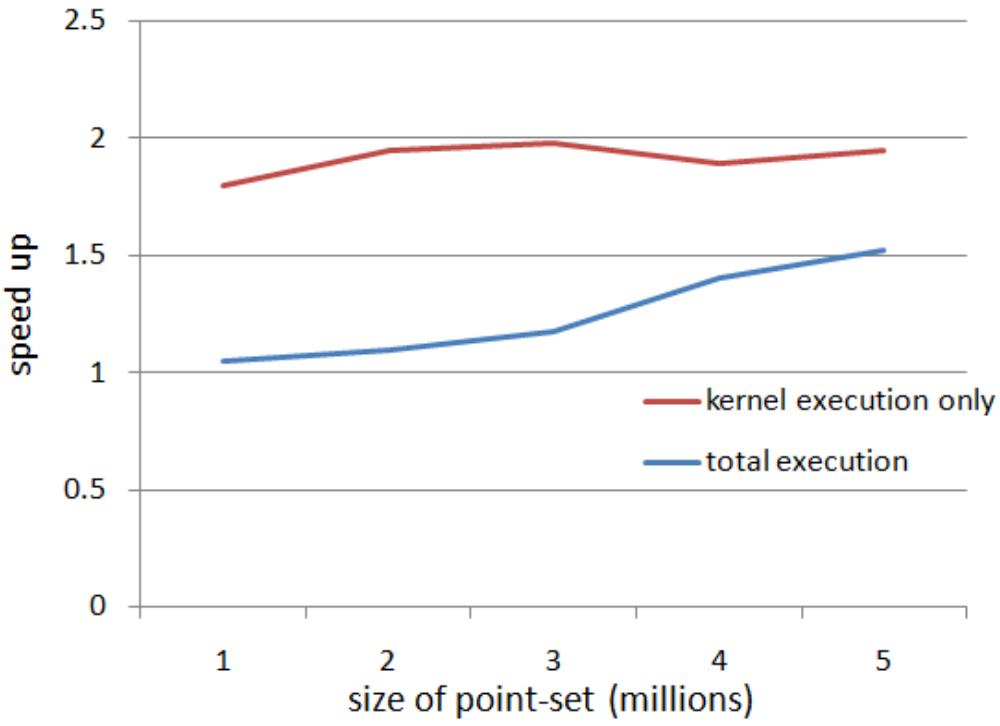

Figure 7-7 The speed-up plot of multi-GPU PBA computations over the single-GPU version by considering overall computation times and kernel execution times

7.4 Meshing Using Upgraded Delaunay Triangulation

This section is intended to explain the meshing capability of the upgraded GPU-DT implementation using a simple example. In this demonstration a geometrical model of a tool (spanner) is meshed using the existing GPU-DT and its upgraded implementations. In this model four different constraint boundaries are used. For this geometry the existing GPU-DT implementation produces a mesh as shown in the figure 7.8. This implementation obviously

considers only the convex-hull of the given point-set and generates the Delaunay triangulation correspondingly. This does not recognize the constraint boundaries and produces the triangle within and outside the internal and external boundaries respectively.

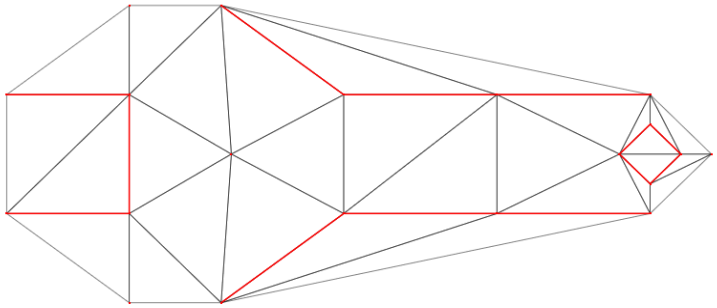

Figure 7-8 Delaunay mesh of a tool using the existing GPU-DT

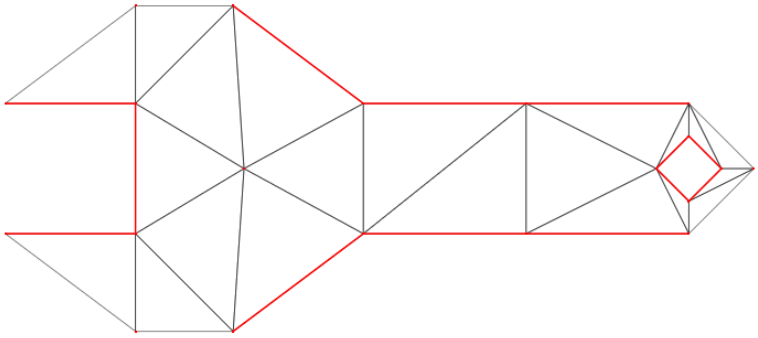

Figure 7-9 Delaunay mesh of a tool using the upgraded GPU-DT as mesh generator

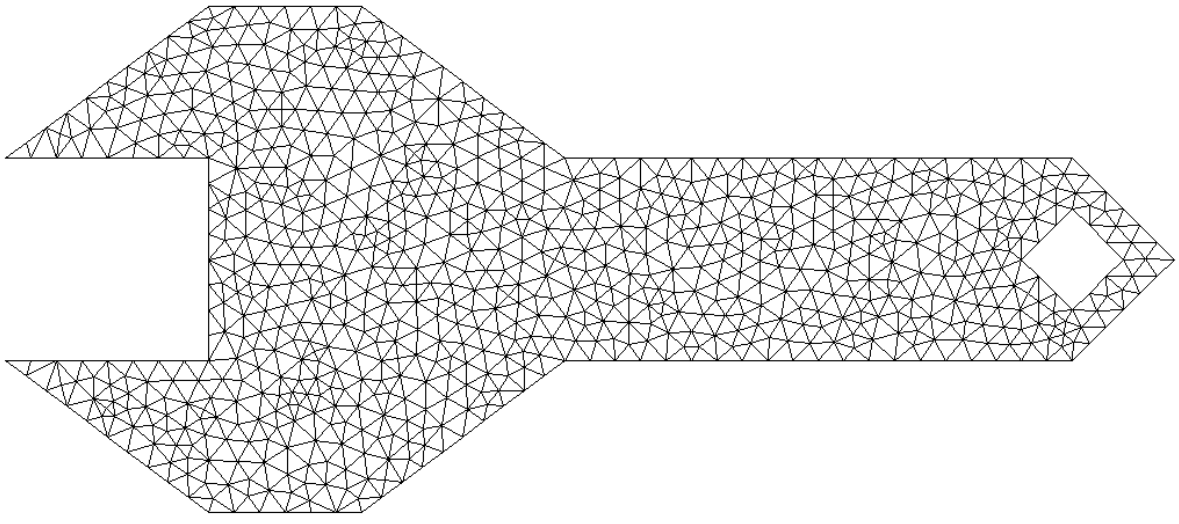

Figure 7-10 Final refined mesh of a tool using upgraded GPU-DT mesh generator

The upgraded version GPU-DT as a mesh generator is capable of recognizing constraint boundaries and thus removes those triangles which are not within the defined boundaries (see figure 7.9). In this implementation, the parallel execution of fixing the constraint boundaries is not a focus. The computation is done in the CPU as a sequential process. This process influences negatively the overall performance of the upgraded GPU-DT. It should be noted that updating this process as a further CUDA parallel computation is one possible research outlook for future versions.

A refined point-set of this geometry is produced with the help of a third party mesh generator. For this point-set the upgraded GPU-DT mesh generator can produce a mesh as shown in figure 7.10.

7.5 Computation Work-load of Interface-wall Triangulation

In this section a study of interface wall construction and its computational work load are discussed. The triangulation method incorporated in the multi GPU meshing concept is an incremental construction technique. In general, this technique is relatively slow and does not favor constructing large number of triangles. In this study, the main argument is that the computation time needed to construct the interface wall triangles is much less than the overall computation time. To illustrate this scenario, a meshing problem as shown in figure 7.11 is defined as two sub mesh problems. For this problem, the number of interface triangles is recorded for different mesh problem sizes. That means by changing the number of mesh points, the size of the meshing problem is changed.

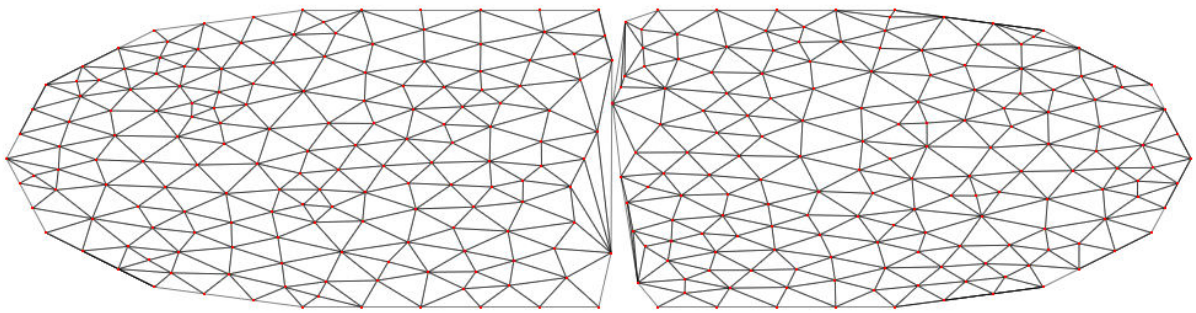

Figure 7-11 Two sub domain meshes of a multi-GPU-DT meshing

As shown in figure 7.12, the proportion of interface triangles with respect to the total number of triangles in the meshing problem is plotted. From this graph one can notice that the

percentage of interface triangles is significantly reduced when the meshing problem size becomes large. This result leads to a conclusion that the utilization of an incremental construction technique to determine interface wall triangles would not significantly affect the computation time for large meshing problems. There are techniques such as bucketing search and unit grid methods to speed up the process of searching for points in incremental construction methods [CIGNONI, 1994]. If one of these techniques would be incorporated in this implementation, then the interface wall construction process would be faster. This could be another research outlook in the future versions.

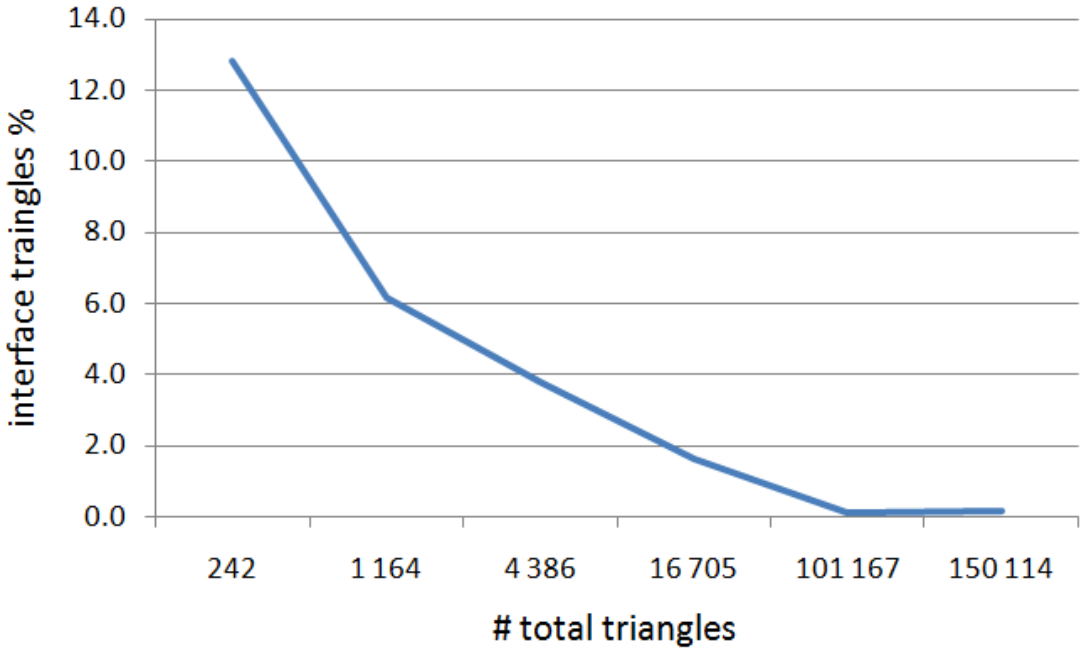

Figure 7-12 The proportion of interface triangles with respect to the total number of triangles in a meshing problem

Chapter 8

8 Conclusion & Outlook

8.1 Summary and Research Contribution

Finite element method (FEM) becomes an inevitable method in most of the current numerical simulations practices. These simulations are becoming continuously more complex, because of the increasing complexity of designed products and the expected precision in solutions. As an example, the required finite element (FE) model of an atomic submarine contains more than one million components. To produce a more reliable and accurate solution, models are refined in general to get high resolution results. This refinement requirement again increases the size of the problem. To analyze dynamic and real-time FE problems, faster and efficient mesh generation techniques are required. To satisfy these requirements in FE analysis (FEA), high performance computing resources are required. The continuous development in computer technology and support of parallel computation techniques already paves the way for large scale FE simulations. Industrial experiences signify the importance of geometric modeling, meshing, and meshing related processes in FEA. The general relation between modeling and analysis in a life cycle of FE design of a product is given as a ratio 80:20 of modeling to analysis.

Parallel computation can be applied in all stages of the FE-simulations. Various parallel techniques and algorithms were developed to exploit the existing parallel computing resources. Even though many CPU-based parallel meshing algorithms and implementations exist, the parallel computing potential of modern graphic processing units (GPUs) has not yet been fully exploited by meshing applications. Most of the modern GPUs support numerical computation other than graphic computation. They are available with many hundreds of computing cores within a small device. The computing system accelerated by graphic hardware is cheap and energy efficient compared to traditional CPU computing systems. There is a great potential to replace the CPU-based parallel computing system with a GPU-based system. To incorporate with an efficient parallel computing system, finding new parallel algorithms to exploit GPUs are in demand.

General purpose GPU (GPGPU) parallel computing is widely used in various numerical computing applications and realizes a large performance improvement over traditional CPU-based parallel computing. The application of GPGPUs in computational geometry problems, such as triangulation and mesh generation, are challenging problems and researches in this area are relatively new and highly motivated within the research community. General aspects of efficient GPU parallel computation are the availability of huge numbers of light weight

parallel threads, reduced communication requirements between CPU and GPU, and no internal synchronization requirements between parallel threads. Unfortunately, computational geometry or triangulation algorithms contain heavy-weight parallel threads. They are involved a lot of synchronization requirements and highly depended on the communication between the CPU and GPU. This communication overhead influences the parallel performance of these algorithms negatively.

The Delaunay triangulation method is a widely applied triangulation technique to generate meshes to solve partial differential equations used in FEM applications. The bounded theoretical facts of Delaunay triangulation make it a well applied triangulation technique. Common experience says that for a given set of points, Delaunay triangulation produces a better triangulation than any other technique. One of the most efficient GPU-based parallel Delaunay triangulation implementation is GPU-DT. However, it supports parallel computation only for a single GPU device. This implementation was programmed using CUDA (compute unified device architecture) programming environment.

This thesis is focused to develop a Delaunay based mesh generator for FE meshing applications. Proposed mesh generator is expected to be computed in parallel using any computing system which accelerated with multiple GPUs. For the Delaunay triangulation the existing GPU-DT implementation is planned to upgrade to support on multi GPU system. The CUDA programming environment is considered to develop this parallel mesh generator.

According to this thesis objective, a Delaunay-based parallel mesh generator using multiple GPUs computing system is presented. In this research concept, four main steps are implemented to contribute to the presented multi GPU mesh generator. They are (1) partitioning of a point-set, (2) meshing of interface regions between subdomains, (3) meshing of subdomains, and (4) merging the subdomain meshes into a final mesh. A coordinate-bisection partitioning technique is used to partition the given set of mesh points into different vertical partitions. The number of partitions depends on the available number of GPUs of the applied computing system. The interface between subdomains is constructed as a preprocess using a Delaunay triangulation technique. In this implementation, an incremental construction algorithm is used to identify the Delaunay triangles in the interface region.

The parallel banding algorithm (PBA) is a GPU-based algorithm which is used as one part of the GPU-DT algorithm to construct the digital Voronoi diagram. The computation of digital Voronoi diagrams is parallelized by executing multiple kernels concurrently on multiple GPUs. After the parallel execution of triangulation, a merging process combines subdomain meshes together by considering interface region meshes. Implementation of the partitioning, the meshing of the interface regions and the merging are implemented as CPU-level computations. The existing Delaunay triangulation does not consider non-convex internal and external constraint boundaries. To upgrade this as a mesh generator to function on any kind of geometry domain, these boundary constraints are considered during triangulation. The

existing triangulation version is thus upgraded according to these constraint-boundary requirements.

The Delaunay triangulation performance of GPU-DT is compared with another CPU-based sequential Delaunay triangulator (*Triangle*). This preliminary performance test shows a speed up of more than two in most test cases using the earlier mentioned computing system. According to the reported results of the original developers of the GPU-DT implementation, it realizes a speed up of more than ten over the *Triangle*. This initial test results show confidently that this work is an efficient way to perform a Delaunay triangulation using GPUs to develop a mesh generator.

To study the effect of partitioning a mesh problem and processing it on different GPUs, the quantity of missing points are compared between a single GPU and a multiple GPU version of triangulations. The test results show that the quantity of missing points in a multiple GPUs computation is reduced several times compared to a single GPU version. Because of the limited size of texture memory space in a single GPU device, accommodating a number of points on a single device is limited. Since the process of inserting these missing points back to the triangulation is an expensive process, partitioning the triangulation process and computing on many devices promises to improve the overall performance of triangulation. Another test is used which compares the quantity of triangles required in an interface region with respect to the overall triangles in triangulation. This test result show that the required work load to produce the triangles in an interface wall region is relatively small compared to the overall work load.

The upgraded PBA algorithm, which works on multiple GPU system, is tested separately. The multi GPU version of the PBA algorithm is tested against the original single GPU version to construct digital Voronoi diagrams. Among the different test cases using a computing system with two GPUs, a speed up of around 1.7 is experienced by the multi GPU version. In this test, the total runtime of PBA computation is taken in to account. Still, the scalability of this algorithm into multiple devices is not fully achieved due to the synchronization requirements for memory transfer between the host and device. Because the sequential memory management consumes a large part of the overall computation time, the overall parallel performance of this algorithm is affected. These types of algorithms are still waiting for the development of CUDA technology to support asynchronous memory transfer between host and device in both directions. If the future CUDA technology solves the limitation of concurrent data transfer of multiple streams in any direction, the overall performance of this PBA algorithm will improve even further. Apart from this PBA algorithm, the rest of the triangulation algorithm depends highly on the interaction between the host and device during parallel execution. Due to this nature, the overall algorithm executes sequentially and affects the parallel performance. When the future CUDA technology overcomes these issues, the concurrent execution of these algorithms on different GPUs will not be an issue any longer.

Then the multiple GPU-DT meshing can be computed on a multiple-GPU-cluster system with good scalability.

8.2 Research Outlook

The triangulation using GPU-DT is based on the dual property of Delaunay and Voronoi tessellation. Therefore, to construct the digital Voronoi diagram, mesh points should be ready at the beginning. In the current research objective, it is not a focus to include this point generation task within this implementation. Defining mesh geometry and generating mesh points still depends on third party mesh generators. One of the research outlooks is to incorporate this point generation task as a GPU-based computation and include it into this mesh generator. For the mesh decomposition task, only a given point-set is considered to split the meshing domain into vertical partitions. Mesh geometry and initial coarse mesh partitions are not taken into account. This would be another research outlook, namely to incorporate different mesh partitioning techniques into this GPU-based meshing implementation.

In the current implementation, mesh partitioning and merging subdomain meshes are completely CPU-based computations. Extending these tasks as a GPU-based parallel computation is another future goal. In the computation of an interface wall between subdomains, an incremental construction Delaunay triangulation technique is used. This is quite an expensive computation method with high complexity. In this triangulation process of an interface region, the requirement of finding the quantity of triangles is limited. Compared with the total number of triangles in the final mesh, the quantity of triangles in an interface region is very low. Implementing this construction method as a GPU-based parallel version is a highly complex task due to the requirement of sequential computing steps. But incorporating other merging techniques is another future goal to improve this mesh generator.

The final research goal of this study is to be used this Delaunay mesh generator for FE applications. To bring this triangulation into FE meshes, certain post-processes are required (e.g., improving the quality of elements). In this implementation, the internal and external constraint boundaries are fixed according to the given mesh geometry as a CPU-based computation. Implementing this task as a GPU-based parallel computation is another future scope.

9 References

AGGARWAL. A., CHAZELLE B, GUIBAS L, O'DUNLAIG. C, AND YAP. Parallel computational geometry. Algorithmica 3(3):293–327, 1988

ANTONOPOULOS. C., X. DING, A. CHERNIKOV, F. FLAGOJEVIC, D. NIKOLOPOULOS, AND N. CHRISOCOIDES. Multigrain parallel Delaunay mesh generation: challenges and opportunities for multi-threaded architectures. In ICS '05 proceedings, ACM, pages 367–376, New York, NY, USA, 2005.

ASANOVIC. K., R. BODIK, J. DEMMEL, T. KEAVENY, K. KEUTZER, J. KUBATOWICZ, N. MORGAN, D. PATTERSON, K. SEN, J. WAWRZYNEK, D. WESSEL, K. YELICK, A view of the parallel computing landscape, Communications of the ACM, Vol. 52 No. 10, Pages 56-67, 2009.

ASANOVIC. K., R. BODIK, B. C. CATANZARO, J. J. GEBIS, P. HUSANDS, K. KEUTZER, D. PATTERSON, W. L. PLISHKER, JOHN. SHALF, S. W. WILLIAMS, AND K. YELICK, The landscape of parallel computing research: A view from Berkeley, Technical Report 2006.

http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

BEICHL. I AND F. SULLIVAN. Parallelizing computational geometry: First steps. SIAM News, 24(6):1–17,1991.

BERG. M. D., O. CHEONG, M. V. KREVELD, AND M. OVERMARS, Computational Geometry, Algorithms and Applications, 3rd edition, 2008.

BLANDFORD. D. K., G. E. BLELLOCH, AND C. KADOW. Engineering a compact parallel Delaunay algorithm in 3D. In Proc. SCG'06, Arizona, USA ,pages 292-300, 2006.

BLELLOCH. G. E., J.C. HARDWICK, G. L. MILLER, AND D. TALMOR. Design and implementation of a practical parallel Delaunay algorithm. Algorithmica, 24:243–269, 1999.

BLELLOCH. G. E., G. L. MILLER, AND D. TALMOR. Developing a Practical Projection-Based Parallel Delaunay Algorithm. In 1996 ACM Symp on Computational Geometry, pp. 186–195, Philadelphia, May 1996.

BOWYER. A, Computing Dirichlet tessellations, The Comp. J., 24(2), 162-167, 1981

CAO. T. T, Computing 2D Delaunay Triangulation using GPU, B.Comp. Dissertation, 2009.

CAO. T-T., K. TANG, A. MOHAMED, AND T-S. TAN, Parallel banding algorithm to compute exact distance transform with the GPU, Proceeding of the symposium on interactive 3D Graphics and Game (pp 83-90), ACM press, 2010.

CERVENANSK´Y. M., TOTH. Z, J. STARINSK´Y, A. FERKO, AND M. SR´AMEK, Parallel gpu-based data-dependent triangulations. Computers & Graphics, 34(2):125–135, 2010.

CHANDRASEKHAR. N., AND W. R. FRANKLIN. A fast practical parallel convex hull algorithm. Technical report, Electrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY, Nov, 1989.

CHEW. L. P, Guaranteed-Quality Triangular Meshes, Department of Computer science, Cornell University, 1989

CHEW. L. P, Guaranteed-Quality Mesh Generation for Curved Surfaces, In Proceedings of the ninth annual symposium on Computational geometry, pp. 274-280, 1993.

CHEN. M. B., CHUNG. T. R, AND WU. J. J, Parallel divide-and-conquer scheme for 2D Delaunay triangulation. Concurrency Computation: Practice and Experience. 2006; 18: 1595-1612, 2006

CHEN. M. B., CHUNG. T. R., AND WU. J. J, Parallel 2D Delaunay Triangulation in HPF and MPI, 15th International parallel and distributed processing symposium, IEEE Computer Society, 2001.

CHEN. M. B., CHUNG. T. R., AND WU. J. J, Efficient parallel implementations of 2D Delaunay triangulation with High Performance Fortran. In Proceedings of 10th SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, VA, USA, Philadelphia, PA: SIAM Press, 11pp, March 2001.

CHEW. L. P, N. CHRISOCHOIDES, AND F. SUKUP, Parallel Constrained Delaunay Meshing. In ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation, pages 89–96, Northwestern University, Evanston, IL, 1997.

CHRISOCHOIDES. N, A survey of parallel mesh generation methods. Technical Report Brown SC-2005-09, Brown University, 2005. Also appears as a chapter in Numerical Solution of Partial Differential Equations on Parallel Computers (eds. Are Magnus Bruaset and Aslak Tveito), Springer, 2006.

CHRISOCHOIDES. N., A. CHERNIKOV, A. FEDOROV, A. KOT, L. LINARDAKIS, AND P. FOTEINOS, Towards Exascale Parallel Delaunay Mesh Generation. Pages 319-336. Salt Lake City, UT, October 2009

CIGNONI P, MONTANI C, PEREGO R, AND SCOPIGNO R, Parallel 3D Delaunay triangulation. Comput Graph Forum 12(3): 129–142, 1993

CIGNONI. P. A, Merge-First Divide & Conquer Algorithm for Ed Delaunay Triangulation, 1994.

CLOUGH. R. W, Speech by Professor R. W. Clough, Early history of the finite element method from the view point of a pioneer, Int. Journal for Numerical Methods Engineering, 2004.

COTTRELL. J. A., T.J.R. HUGHES, AND Y. BAZILEVS, Isogeometric Analysis: towards integration of CAD and FEA, John Wiley & Sons, Ltd, pages xii, 1-3, 2009.

COUGNY. H. DE., AND M. SHEPHARD. Parallel Volume Meshing Using Face Removals and Hierarchical Repartitioning. Comp. Meth. Appl. Mech. Engng., 174(3-4): 275–298, 1999.

DAVY. J. R AND P.M. DEW. A note on improving the performance of Delaunay triangulation. In Proc. of Computer Graphics International ’89, pages 209–226, 1989.

DAY. A. M, Parallel implementation of 3D convex hull algorithm. Computer Aided Design, 23(3):177–188, Apr. 1991.

DEVILLERS. O, Improved incremental randomized Delaunay triangulation. In: Proceedings of 14th Annual symposium on Computational Geometry. Washington, DC: ACM, pp 106–115, 1998.

DWYER. R. A, A simple divide-and-conquer algorithm for constructing Delaunay triangulation in O(n log log n) expected time. In Proceedings of the Second Annual Symposium on Computational Geometry, ACM, NY, USA, pp 276–284, 1986.

DWYER. R. A, A faster divide-and-conquer algorithm for constructing Delaunay triangulations. Algorithmica, 2:137–151, 1987.

EDAHIRO. M. I, KOKUBO. I, AND ASANO. T, A new point location algorithm and its practical efficiency - comparison with existing algorithms. ACM Trans. Graphics, 3:86-109, 1984

EDELSBRUNNER. H AND W. SHI, An O(nlog2h) time algorithm for the three-dimensional convex hull problem. SIAM J. Computing, 20:259-277, 1991.

EDELSBRUNNER. H, Geometry and topology for mesh generation (cambridge monographs on applied and computational mathematics), 2001.

EVANS. D. J., AND I. STOJMENOVIC. On parallel computation of Voronoi diagrams. Parallel Computing, (12):121–125, 1989.

FARBER. R, CUDA Application design and development, Morgen Kaufmann, 2011

FARHAT. C., AND ROUX. F.-X, A method of finite element tearing and interconnecting and its parallel solution algorithm. Int. J. Numer. Meth. Engng., 32: 1205–1227, 1991.

FLYNN. M, Some computer organizations and their effectiveness. IEEE Transactions on Computers, 1972 [cited at 35]

FORTUNE. S, A sweepline algorithm for Voronoi diagrams. Algorithmica, 2:153–174, 1987.

FREY. P. J., AND P-L. GEORGE, Mesh Generation application to finite elements 2nd Ed, 2008

GALTIER. J AND P. L. GEORGE. Prepartitioning as a Way to Mesh Subdomains in Parallel. In Special Symposium on Trends in Unstructured Mesh Generation, pages 107–122. ASME/ASCE/SES, 1997.

GEORGE. P. L., AND H. BOROUCHAKI, Delaunay Triangulation and Meshing, Application to Finite Elements, Hermes, 1998

GPGPU, http://gpgpu.org/, last seen October 2013.

GOULD. N. I. M., Y. HU, AND J. A. SCOTT, A numerical evaluation of sparse direct solvers for solution of large sparse, symmetric linear systems of equations, Technical Report, CCLRC, 2005.

GUIBAS. L AND J. STOLFI. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. ACM Transactions on Graphics, 4(2):75–123, 1985.

GRAHAM. R. L, An efficient algorithm for determining the convex hull of a finite planar set, Information Processing Letters, vol. 1, no. 4, pp. 132 – 133, 1972.

GREEN. P. J., R. SIBSON, Computing Dirichlet tessellations in the plane, The Computer Journal, 21(2):168-173,1978.

HARDWICK J. C. Implementation and evaluation of an efficient parallel Delaunay triangulation algorithm. In: Proceedings of Ninth Annual Symposium on Parallel Algorithm and Architectures, p. 22–5, 1997.

HEISTER. T, KRONBICHLER. M, AND BANGERTH. W, Generic finite element programming for massively parallel flow simulations, ECCOMAS CFD 2010, Lisbon, Portugal, 2010

HENNESSY. J AND D. PATTERSON, Computer Architecture: A Quantitative Approach, 4th edition, Morgan Kauffman, San Francisco, 2007

HILL. M. D., AND MARTY. M, Amdahl's Law in the multicore era. IEEE Computer Vol 41,Issue 7 , pp 33–38, 2008.

HOFF III. K. E., CULVER. T, KEYSER. J, LIN. M, AND MONOCHA. D, Fast computation of generalized Voronoi diagrams using graphics hardware. In proceeding of ACM SIGGRAPH 99, 1999.

ISENBERG. M., Y. LIU, J. SHEWCHUK, AND J. SNOEYINK, Streaming computation of Delaunay triangulations. ACM Transactions on Graphics, 25(3):1049–1056, 2006.

JEONG. C. S, An improved parallel algorithm for constructing Voronoi diagram on a mesh-connected computer. Parallel Computing, (17):505–514, 1991.

JEONG. C. S, Parallel Voronoi diagram in L1 (L8) on a mesh-connected computer. Parallel Computing, (17):241–252, 1991.

JONES. M. T., AND P. E. PLASSMANN. Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes. In Proceedings of the Scalable High-Performance Computing Conference, 1994.

KADOW. C, Adaptive Dynamic Projection-Based Partitioning for Parallel Delaunay Mesh Generation Algorithms. In SIAM Workshop on Combinatorial Scientific Computing, February 2004.

KEMP, JEREMY, AND MARK, All-Pairs Shortest Path Algorithms Using CUDA, E-Master thesis, Durham University, 2012

KIRK, D. B., WEN-MEI W. HWU, Programming Massively Parallel Processors, A Hands-on Approach. ElsevierInc, Morgan Kaufmann, 2010

KOHOUT. J., AND I. KOLINEROVA, Parallel Delaunay triangulation in E3 make it simple, 2003

KOHOUT. J., I. KOLINEROVA, AND J. ZARA, Practically oriented parallel Delaunay triangulation in E2 for computers with shared memory, 2004.

KOLINEROVA. I, AND J. KOHOUT, Optimistic parallel Delaunay triangulation, 2002.

KOT. A., A. CHERNIKOV, AND N. CHRISOCHOIDES, Effective out-of-core parallel Delaunay mesh refinement using off-the-shelf software, 20th IEEE International Parallel and Distributed Processing Symposium, 2006.

LAWSON. C. L, Transforming triangulations, Discrete Mathematics, 3(4): 365 – 372, 1972.

LINARDAKIS. L., AND N. CHRISOCHOIDES, Algorithm 870: A static geometric medial axis domain decomposition in 2D Euclidean space, ACM Transactions on Mathematical Software, 34(1): 1-28, 2008

LEE, D. T., B.J. SCHALTER, Two algorithms for Constructing a Delaunay Triangulation, Int. J. Comput.Inf.Sci.,9(9): 219-242, 1980.

LEE. S., C. I. PARK, AND C. M. PARK, An Efficient Parallel Algorithm for Delaunay Triangulation on Distributed Memory Parallel Computers, in Proc. The 1996 Int. Conf. PDPTA, Aug. 1996.

LEE. F, Constructing the constrained Delaunay triangulation on the Intel Paragon. In Proceedings of the 13th Annual Symposium on Computational Geometry, Washington, DC: ACM, pp. 464–467, 1997.

LEE. S., C. I. PARK, AND C. M. PARK. An improved parallel algorithm for Delaunay triangulation on distributed memory parallel computers. Parallel Processing Letters; 11 (2–3):341–52, 2001.

LINARDAKIS. L., AND N. CHRISOCHOIDES, Parallel Domain Decoupling Delaunay Method. SIAM Journal on Scientific Computing, in print, accepted Nov. 2004.

LINARDAKIS. L AND N. CHRISOCHOIDES, Delaunay Decoupling parallel guaranteed quality planar mesh refinement. SIAM Journal on Scientific Computing, 27, 4, 1394-1423, 2006.

MCLAIN. D. H, Two Dimensional interpolation from random data. The Computer J., 19(2): 178-181, 1976.

MPI, Message Passing interface, last seen September, 2013. http://www.mcs.anl.gov/research/projects/mpi/, last seen

NAVARRO. C. A., HITSCHFELD-KAHLER, N. AND SCHEIHING. E, A parallel GPU-based algorithm for Delaunay edge-flips, EurocG, 2011.

NAVARRO. C. A., HITSCHFELD-KAHLER, N. AND SCHEIHING. E, A quasi-parallel GPU based algorithm for Delaunay edge-flips, 2011.

NAVE. D., N. CHRISOCHOIDES, AND P. CHEW, Guaranteed-quality parallel Delaunay refinement for restricted polyhedral domains, Proc. 18th ACM Symp. Comp. Geometry, 135-144, 2002.

NELSON. D., AND R. LUNNAN, Delaunay Triangulation using parallel Incremental Extrapolation on GPUs, https://www.cfa.harvard.edu/~dnelson/ArepoCUDA/, last accessed, 2012, November.

NVIDIA1, CUDA C Best Practice Guide, January 2012.

NVIDIA2, https://developer.nvidia.com/cuda-gpus, September, .2013.

NVIDIA3, CUDA toolkit documentation, http://docs.nvidia.com/cuda/cuda-c-programming-guide/, January, .2014.

OKUSANYA. T., J. PERAIRE, Parallel Unstructured Mesh Generation, 5th Int. Conf. on Numerical Grid Generation, 1996.

OKUSANYA. T., AND J. PERAIRE, 3D Parallel unstructured mesh generation, 1997.

OPENMP Specification, http://openmp.org/wp/about-openmp/, last seen September 2013.

PARAVIEW, http://www.paraview.org/, seen 2013

QI. M., T-T. CAO, AND T-S. TAN, Computing 2D Constrained Delaunay Triangulation Using Graphics Hardware, IEEE Transactions on Visualization and Computer Graphics, vol 19 (5), 736-748, 2013.

RAINLAND. L., C. JOSE, AND M. MARSHAL, Parallel Unstructured Grid Generation. Unstructured Scientific Computation on Scalable Multi processors, Ed Piyush Mehrotra and Joel Saltz, MIT Press, pp 31-64, 1990.

RIVARA. M. C., D. PIZARRO, AND N. CHRISOCHOIDES, Parallel Refinement of Tetrahedral Meshes Using Terminal-Edge Bisection Algorithm. In 13th International Meshing Roundtable, September 2004.

RONG. G., AND T-S. TAN, Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. Proceeding of the symposium on interactive 3D Graphics and Game (pp. 109-116): ACM Press, 2006.

RONG. G., AND T-S TAN, Variant of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams, 2007

RONG, G. D., T.S.TAN, THANH-TUNG CAO AND STEPHANUS, Computing Two-dimensional Delaunay Triangulation Using Graphics Hardware. ACM Symposium on Interactive 3D Graphics and Games,15-17 Feb 2008, Red-wood City, CA, USA, pp.89-97, 2008

RUETSCH. G, How to optimize Data Transfers in CUDA Fortran, Nvidia, 2012

https://developer.nvidia.com/content/how-optimize-data-transfers-cuda-fortran

SAID. R., N. P. WEATHERILL, K. MORGAN, AND N. A. VERHOEVEN, Distributed Parallel Delaunay Mesh Generation. Computer Methods in Applied Mechanics and Engineering, (177):109–125, 1999.

SHEN. J., L. GUO, L. QI, AND W. ZHU, Delaunay Triangulation Parallel Construction Method and its Application in Map Generalization, Melbourne, Australia, 2012.

SHEWCHUK. J. R Triangle: A Two Dimensional Quality Mesh Generator and Delaunay Triangulator. In Applied Computational Geometry Towards Geometric Engineering, M. Lin and D. Manocha, Eds., vol. 1148 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 203-222, 1996.

SHEWCHUK. J. R, Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, Discrete & Computational Geometry 18:305-363, 1997. Also Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996.

SIMON. H, Partitioning of unstructured problems for parallel processing, Comp. systems in Eng, 1991.

SPIELMAN. D. A., SHANG. H. T, AND ALPER. U, Parallel Delaunay refinement: algorithms and analyses. In Proceedings of 11th International Meshing Roundtable, Sandia National Laboratories, p. 205–18, September 15–18, 2002.

SU. P., AND R. DRYSDALE, A comparison of sequential Delaunay triangulation algorithms. Computational Geometry: Theory and Applications, 7:361–386, 1997.

TENG. Y. A., F. SULLIVAN, I. BEICHL, E. PUPPO, A data-parallel algorithm for three-dimensional Delaunay triangulation and its implementation, sc, pp.112-121, Proceedings of the 1993 ACM/IEEE conference on Supercomputing, 1993

TOPPING, B.H.V., J. MUYLLE, P. IVANYI, R. PUTANOWICZ AND B. CHENG. Finite Element Mesh Generation, Saxe-Coburg Publications, 2004.

TOP 500, http://top500.org/project/introduction/, last seen Feb 2013

TOPPING. B. H. V., AND A. I. KHAN, Parallel finite element computations, Saxe-Coburg Publications, 1996

TOPPING. B. H. V., AND P. IVANYI, High Performance Computation for Engineering, PhD Course, Pecs, 2010

VASILEIOU. P., AND PSARAKIS. E, A Hybrid 2-D Delaunay Triangulation Algorithm Exploiting the GPU Parallelism, Dept. of computer engineering and informatics, University of Patras, Greece, http://www.highperformancegraphics.org/media/Posters/HPG2012_Posters_Vasileiou.pdf, last accessed November 2012.

VICENTE. H., F. BATISTA, D. L. MILLMAN, S. PION, AND J. SINGLER, Parallel Geometric Algorithms for Multi-Core Computers, In Proc. SCG'09, Aarhus, Denmark , 2009

WATSON. D. F, Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes, Computer Journal, 24(2), 167-172, 1981.

WIDAS. P, Introduction to finite element analysis, http://www.sv.vt.edu/classes/MSE2094_NoteBook/97ClassProj/num/widas/history.html, Last seen 2013.

WIKIA CFD, http://en.wikipedia.org/wiki/Computational_fluid_dynamics, last seen on September, 2013.

WIKIB PARCOMP, http://en.wikipedia.org/wiki/Parallel_computing, last seen February 2013

WIKIC LINALGLIB, http://en.wikipedia.org/wiki/Comparison_of_linear_algebra_libraries, last seen September, 2013.

WIRTH. N, Algorithms and Data-Structures, Prentice Hall, 1986.

WU. P. AND E. N. HOUSTIS, Parallel adaptive mesh generation and decomposition, Engineering with Computers, 12, 155-167, 1996.

YANG. C. T., CHIH-LIN HUANG, AND CHENG-FANG LIN, Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters, Computer Physics Communications 182 (2011), 266-269, 2011

YUAN. Z., G. RONG, X. GUO, AND W. WANG, Generalized Voronoi Diagram Computation on GPU, Voronoi Diagrams in Science and Engineering (ISVD), 2011 Eighth International Symposium on, vol., no., pp.75,82, 28-30 June 2011.

List of Figures

List of Tables

Curriculum Vitae

Name	Vishnukanthan Kandasamy
Email	Vishnukanthan.kandasamy@rub.de
Date of Birth	19-10-1978, Sri Lanka
Nationality	German

Education

01.2010 – present	Ruhr-Universität Bochum, Lehrstuhl für Informatik im Bauwesen, **PhD candidate** under Prof.Markus König
10.2007 – 10.2009	Ruhr-Universität Bochum, Germany **MSc in Computational Engineering**
10.2004 – 12.2006	Universität zu Köln, Germany **MSc in Environmental Sciences**
10.1998 – 08.2003	University of Moratuwa, Sri Lanka **BSc Civil Engineering (Honors)**
01.1989 – 09.1997	Union College, Tellippalai, Sri Lanka Primary, secondary, and high school

Work Experience

01.2010 – 09.2013	Ruhr-Universität Bochum, Deutschland **Research Associate**
04.2010 – 09.2007	SANTEC Fuchs Sanierungstechnologie GmbH, Köln, Deutschland. **Working-student**
09.2003 – 10.2004	SD&CC, Sri Lanka. **Civil Engineer**
09.2003 – 04.2004	Universität Moratuwa, Sri Lanka. **Teaching Assistant**
10.2002 – 09.2003	Segal Property Developers (Pvt) Ltd, Sri Lanka. **Junior Civil Engineer**
05.2002 – 09.2009	Maga Engineering (Pvt) Ltd Complex Construction, Sri Lanka **Trainee**

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification at this or any other university. This dissertation is entirely the results of my own work.

Vishnukanthan Kandasamy

Bochum, March 2014