Parallelverarbeitung spezieller Triangulationen

D I P L O M A R B E I T

zur Erlangung des akademischen Grades
Diplom-Informatiker

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik

eingereicht von Henning Meyerhenke
geb. am 21. Februar 1978 in Paderborn

Betreuer: Prof. Dr. Hans-Dietrich Hecker

30. August 2004

Zusammenfassung

Die vorliegende Arbeit befaßt sich mit der parallelen Erstellung von Gitternetzen zur realistischen Modellierung von Oberflächen durch Delaunay-Triangulationen höherer Ordnung. Die Ergebnisse von Gudmundsson et al. [39], die diese speziellen Triangulationen eingeführt haben, beziehen sich ausschließlich auf eine serielle Abarbeitung. Da die zugehörigen Anwendungen, darunter Geographische Informationssysteme und die Finite-Elemente-Methode, oftmals Parallelverarbeitung erfordern, werden hier mehrere bekannte serielle Algorithmen für die effiziente Anwendung auf grobkörnigen Parallelrechnern modifiziert und im CGM-Modell analysiert, darunter die planare Delaunay-Triangulation, Delaunay-Triangulationen erster Ordnung, Voronoi-Diagramme höherer Ordnung, die Bestimmung der Ordnung einer Triangulation und die Tiefensortierung eines geeigneten Gitternetzes.

Darüber hinaus werden drei bekannte parallele Rechenmodelle (PRAM, BSP und CGM) sowie verschiedene serielle und parallele Algorithmen für die planare Delaunay-Triangulation vorgestellt und u. a. hinsichtlich ihrer Zeitkomplexität beurteilt. Einer der parallelen Algorithmen wird in zwei bekannten Varianten implementiert und wegen der nicht überzeugenden experimentellen Laufzeiten durch eine für viele Punktverteilungen im CGM-Modell optimale Alternativversion verbessert, die als Grundlage für die o. g. Verfahren dient.

Inhalt

Einleitung **5**

1 Grundlagen **7**
1.1 Algorithmische Geometrie 7
1.2 Höhere Ordnungen 12
1.2.1 Delaunay-Triangulationen erster Ordnung 17
1.3 Parallelverarbeitung 18
1.3.1 PRAM 21
1.3.2 Bulk-Synchronous-Processing 22
1.3.3 Erweiterungen des BSP-Modells 29
1.4 Fazit 31

2 Forschungsarbeiten zur Delaunay-Triangulation **32**
2.1 Eigenschaften und Anwendungen 32
2.2 Serielle Algorithmen 35
2.2.1 Flipping 35
2.2.2 Inkrementelle Algorithmen 36
2.2.3 Gleitgeradenmethode 36
2.2.4 Teile und herrsche 37
2.2.5 Dualität mit 3D-Konvexe-Hülle 37
2.2.6 Zerlegungsstrategie 38
2.2.7 Fazit 39
2.3 Parallele Algorithmen 39
2.3.1 Berechnung der dreidimensionalen konvexen Hülle 40
2.3.2 Der Algorithmus von Kühn 41
2.3.3 Inkrementelles Verfahren 42
2.3.4 Teile und herrsche 42
2.3.5 Parallele Zerlegung mit Dreieckspfaden 42
2.3.6 Parallele Zerlegung durch Transformation 43
2.3.7 Fazit 44

3 Varianten des parallelen Zerlegungsalgorithmus **45**
3.1 Grundlagen der Zerlegungsmethode . . . 45
3.2 Serieller und paralleler Algorithmus . . . 48
3.3 Zwei Varianten des parallelen Algorithmus . . . 51
3.3.1 Das Kommunikationsschema *Top-down* . . . 51
3.3.2 Das Kommunikationsschema *Sende-an-alle* . . . 57
3.4 Implementierungen und Experimente . . . 60
3.4.1 Implementierungen . . . 60
3.4.2 Ergebnisse der Experimente . . . 62
3.5 *Bottom-Up* und *Top-down* kombiniert . . . 63
3.6 Fazit . . . 70

4 Parallelverarbeitung im Kontext höherer Ordnungen **71**
4.1 Delaunay-Triangulationen erster Ordnung . . . 72
4.1.1 Algorithmus . . . 72
4.1.2 Netzoptimierungen mit Delaunay-Triangulationen erster Ordnung . . . 75
4.2 Voronoi-Diagramme höherer Ordnung parallel berechnen . . . 75
4.2.1 Bisherige Arbeiten . . . 76
4.2.2 Parallelisierung von Lees Algorithmus . . . 77
4.3 Parallele Bestimmung der Ordnung einer Triangulation . . . 80
4.4 Parallel nützliche k-OD-Kanten bestimmen . . . 83
4.5 Fazit . . . 84

5 Paralleles Preprocessing zur Visualisierung von TINs **85**
5.1 Grundlagen . . . 85
5.2 Algorithmus . . . 86
5.3 Fazit . . . 87

Schlußbemerkungen **88**

Literaturverzeichnis **89**

Abbildungsverzeichnis **93**

Einleitung

Dreieckszerlegungen - üblicherweise Triangulationen genannt - von geometrischen Objekten wie Punkten, Polygonen oder Polyedern haben vielfältige Anwendungsmöglichkeiten. Man kann sich anhand dieser Zerlegungen zunutze machen, daß das betrachtete Objekt in kleine Flächen aufgeteilt worden ist und die Punkte oder Kanten, die zu demselben Dreieck gehören, eine gewisse Zusammenhangseigenschaft oder Ähnlichkeit aufweisen, oft ist dies ganz einfach ihre räumliche Nähe.

Ein Geographisches Informationssystem (GIS) speichert geographische Rohdaten und verarbeitet diese zu Informationen für den Benutzer. Bei einem solchen System ist die Terrainmodellierung, also die Abbildung der Erd- oder Meeresoberfläche im Computer zum Zwecke der Weiterverarbeitung, Darstellung und Informationsgewinnung, ein wichtiger Teilaspekt. Da man nicht die Höhe (oder Tiefe) jedes einzelnen Punktes auf der Erde speichern kann, muß man sich mit einer relativ kleinen Auswahl von Meßpunkten zufrieden geben. Die Höhen aller anderen Punkte müssen dann mit Hilfe eines Digitalen Terrainmodells (DTM) durch eine geeignete Interpolation berechnet werden. Bereits 1533 schlug der in den Niederlanden geborene Mathematiker und Astronom Regnier Gemma Frisius, ein Lehrer Gerhard Mercators, vor, Triangulationen zur Grundlage der Kartographie zu machen, weil er genau dieses Prinzip, mit Hilfe weniger Meßpunkte unter Verwendung der Trigonometrie wirklichkeitsgetreue Karten erstellen zu können, erkannt hatte [35]. Heute sind solche Dreiecksgitter oder -netzwerke (engl.: *triangulated irregular network* (TIN)) neben wenigen anderen Datenstrukturen Standard bei der Speicherung und Weiterverarbeitung räumlicher Daten im Computer.

Im rechnergestützten Entwurf von technischen Bauteilen wird oftmals die sogenannte Finite-Elemente-Methode (FEM) verwendet, um Eigenschaften eines Bauteils wie die Bruchfestigkeit, Wärmeleitfähigkeit und Strömungseigenschaften durch Computersimulation zu überprüfen. Man zerlegt dazu das zu betrachtende geometrische Objekt in kleinere Elemente. In der Ebene sind diese Elemente üblicherweise Drei- oder Vierecke, wodurch ein Gitternetz (engl.: *mesh*) entsteht. So erreicht man eine Diskretisierung der für die Simulation benötigten kontinuierlichen Funktionen, die ansonsten über dem gesamten Eingabebereich berechnet werden müßten. Oft ermöglicht erst die Diskretisierung eine Lösung mit Computerhilfe, insbesondere bei partiellen Differentialgleichungen. Die finiten Elemente dieser Gitternetze müssen je nach Anwendung bestimmte Qualitätskriterien aufweisen. So dürfen beispielsweise die Innenwinkel nicht zu klein sein, da sonst die numerischen Berechnungen falsche Ergebnisse liefern würden.

Bei beiden genannten Anwendungen kommt es für die Gewinnung realistischer Informationen wesentlich darauf an, daß das zugrunde liegende Modell der Wirklichkeit sehr nahe kommt

und die bereits erwähnten Qualitätsanforderungen erfüllt werden. Thema dieser Arbeit ist daher die Erstellung solcher Triangulationen, die für die oben beschriebenen Anwendungen besonders positive Eigenschaften haben. Die bekannteste dieser Dreieckszerlegungen ist die Delaunay-Triangulation, die unter anderem den kleinsten Winkel der Triangulation maximiert. Für die serielle Konstruktion der Delaunay-Triangulation gibt es auf dem Gebiet der Algorithmischen Geometrie eine Reihe bekannter Verfahren, auch einige parallele Verfahren sind bereits vorgeschlagen worden [15, 22, 24, 49, 52]. Letzteres ist besonders wichtig, weil die Simulation durch die Finite-Elemente-Methode für viele Anwendungen wegen ihrer Komplexität auf Parallelrechnern durchgeführt wird. Ausgehend von einem dieser parallelen Algorithmen werden daher zwei seiner Varianten implementiert sowie eine verbesserte Abwandlung für ein grobkörniges paralleles Maschinenmodell entworfen.

Bei der Triangulation von Oberflächen hat die zweidimensionale Delaunay-Triangulation den Nachteil, daß sie die räumliche Dimension der Oberfläche nicht in Betracht zieht.[1] Eine Triangulation im Raum liefert allerdings nicht das gewünschte Ergebnis einer Dreieckszerlegung des Oberflächenmodells. Aufgrund des genannten Nachteils können sich unerwünschte Abweichungen des Modells von der Realität ergeben, sogenannte künstliche Dämme. Deshalb gehe ich in meiner Arbeit außerdem der Frage nach, wie man diese Problemstellen durch das Konzept höherer Ordnungen von Delaunay-Triangulationen, eingeführt von Gudmundsson et al. [39], beseitigen kann. Ein besonderer Schwerpunkt liegt dabei auf der parallelen Berechnung von Delaunay-Triangulationen erster Ordnung und von Voronoi-Diagrammen beliebiger Ordnung, die bei der Verbesserung der Modelle eine wesentliche Rolle spielen.

Die Arbeit gliedert sich wie folgt: Im ersten Kapitel werden zunächst die Grundlagen der Problemstellung vermittelt. Diese umfassen sowohl geometrische Datenstrukturen als auch verschiedene Modelle der Parallelverarbeitung einschließlich ihrer Motivationen. Im sich anschließenden Kapitel werden Eigenschaften und Anwendungsmöglichkeiten der Delaunay-Triangulation sowie serielle und parallele Algorithmen zu ihrer Berechnung präsentiert. Ein paralleler Zerlegungsalgorithmus wird anschließend in Kapitel drei detailliert analysiert und anhand zweier Implementierungsvarianten evaluiert. Ausgehend von diesen Varianten und den experimentell gewonnenen Laufzeiten wird danach ein verbesserter Algorithmus für ein grobkörniges paralleles Maschinenmodell formuliert. Der Fokus im vierten Kapitel liegt auf dem Konzept der Delaunay-Triangulationen und Voronoi-Diagramme höherer Ordnung im Kontext der Parallelverarbeitung. Dabei wird unter anderem ein paralleler Algorithmus zur Bestimmung von Voronoi-Diagrammen höherer Ordnung entworfen. Kapitel fünf zeigt mit der Methode der parallelen Tiefensortierung für die Visualisierung eine konkrete Anwendungsmöglichkeit der erzielten Ergebnisse, die im Schlußkapitel noch einmal zusammengefaßt und in den Stand der Forschung eingeordnet werden.

[1] Bei der Darstellung von Terrains durch TINs spricht man oft von 2,5-Dimensionalität. Da die Oberfläche der Erde als Abbildung f: (x,y) $\rightarrow$ z gespeichert wird und f eine Funktion ist (d. h. für festes x und y ergibt sich maximal ein Wert z = f(x,y)), entsteht dadurch kein echtes dreidimensionales Volumen, sondern eine Oberfläche, der die Dimensionalität 2,5 zugeschrieben wird.

Kapitel 1

Grundlagen

1.1 Algorithmische Geometrie

Zum besseren Verständnis werden zunächst einige grundlegende Objekte der algorithmischen Geometrie und deren Eigenschaften erläutert. Die Definitionen und Sätze dieses Unterkapitels sowie deren vollständige Beweise findet man in gängigen Lehrbüchern wie dem von Rolf Klein [47] oder dem sehr anschaulich geschriebenen von de Berg et al. [13].

Definition 1.1 *$S \subset \mathbb{R}^d$ mit $d \geq 2$ heißt konvex genau dann, wenn für je zwei Punkte $p, q \in S$ gilt: $\overline{pq} \subseteq P$.*

Das Geradensegment, das zwei Punkte einer konvexen Menge verbindet, liegt also immer vollständig innerhalb dieser Menge.

Definition 1.2 *(vgl. de Berg et al. [13, S. 2]) Sei $S \subset \mathbb{R}^d$ mit $d \geq 2$. Dann ist die konvexe Hülle CH von S die kleinste Menge, die S enthält und konvex ist. Präziser gesagt ist CH(S) der Durchschnitt aller konvexen Mengen, die S enthalten (s. Abb. 1.1).*

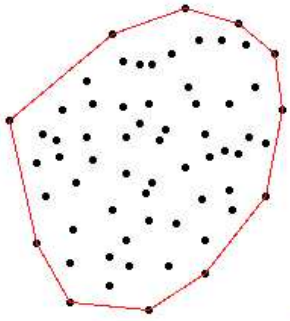

Abbildung 1.1: Planare Punktmenge und ihre konvexe Hülle

Konvexe Hüllen werden z. B. in der Digitalgraphik zur Beschleunigung der Kollisionserkennung von Objekten verwendet. Nur dann, wenn sich die konvexen Hüllen zweier Objekte schneiden,

können diese miteinander kollidieren. Falls sich also die betrachteten Objekte nur selten schneiden, genügt es, zunächst die einfachere Schnittberechnung der konvexen Hüllen durchzuführen. Erst wenn dieser Schnitt nicht leer ist, muß man die eigentlichen Objekte betrachten und deren Durchschnitt in einem normalerweise aufwendigeren Verfahren berechnen.

Bezeichnung 1.3 *Rand(CH(S)) bezeichne die geordnete Menge von Kanten, die im Gegenuhrzeigersinn die Punkte der konvexen Hülle von S miteinander verbinden und so ein konvexes Polygon ergeben. Ein Beobachter, der dem Kantenverlauf folgt, sieht demnach das Innere dieses Polygons immer zu seiner Linken.*

Für die Berechnung der konvexen Hülle in der Ebene gibt es eine ganze Reihe von Algorithmen. Man kann durch Reduktion auf das Sortierproblem zeigen, daß deren Laufzeit eine Komplexität von $\Omega(n \log n)$ für eine Eingabe aus n Punkten haben muß. Allerdings gibt es unter bestimmten Voraussetzungen nützliche Verbesserungen: So benötigt der Algorithmus von Chan [18] nur $O(n \log h)$ Schritte, wobei h die Anzahl der Punkte auf dem Rand der konvexen Hülle bezeichnet[2], und der recht einfach zu implementierende Algorithmus *Grahams Scan* [38] benötigt nur $O(n)$ Schritte, wenn die Eingabe sortiert vorliegt[3].

Wie man auf Abbildung 1.1 sieht, erhält man durch die Berechnung der konvexen Hülle eine gewissermaßen abgeschlossene zusammenhängende Punktmenge. Diese Zusammenhangseigenschaft zwischen Punkten kann man weiter verfeinern, indem man so lange wie möglich zwischen jeweils zwei Punkten eine Kante derart einfügt, daß sich Kanten höchstens in ihren Endpunkten schneiden. Kann man keine Kante mehr einfügen, weil sonst ein nicht erlaubter Schnittpunkt entstehen würde, sind alle entstandenen Flächen Dreiecke. Man nennt daher den Graphen aus den Punkten und den sie verbindenden Kanten *Triangulation* oder *Dreieckszerlegung*. Formal definiert man:

Definition 1.4 *Sei $S \subset \mathbb{R}^2$ eine Punktmenge in der euklidischen Ebene. Der planare Graph G = (S,E) ist genau dann eine* Triangulation *T von S, wenn $E = \{(i,j) \mid \overline{ij}$ ist ein Geradensegment, das höchstens in seinen Endpunkten geschnitten wird $\wedge\, i,j \in S\}$ maximal (also nicht durch Kanten kreuzungsfrei erweiterbar) ist.*

Die bekannteste Triangulation ist die Delaunay-Triangulation DT(S), benannt nach dem russischen Mathematiker Boris Nikolaevich Delone[4]. Zwar hatten sich vor ihm bereits andere mit der Thematik beschäftigt, aber Delone fand in den 1930er Jahren als erster wichtige Strukturmerkmale der nach ihm benannten Triangulation [27].

Damit deren gewünschten Eigenschaften gelten, muß man - wie häufig in der Algorithmischen Geometrie - die allgemeine Lage der Eingabe voraussetzen, um so Entartungen auszuschließen.

Bezeichnung 1.5 *Im folgenden ist mit allgemeiner Lage gemeint, daß keine d+1 Punkte aus $\mathbb{R}^d$ kollinear und keine d+2 Punkte aus $\mathbb{R}^d$ kozirkular sind.*

[2] Im *worst case* können alle n Punkte auf dem Rand liegen, so daß kein Widerspruch zur unteren Schranke $\Omega(n \log n)$ besteht.

[3] Auch hier besteht kein Widerspruch, da die untere Schranke allgemein, also auch für unsortierte Eingaben, gilt und Sortieren von n Punkten $\Omega(n \log n)$ Schritte erfordert.

[4] Wie de Berg et al. [13, S. 189] ausführen, entstand die französische Schreibweise des kyrillischen Namens, weil Delone seine Arbeiten auf französisch publizierte.

Definition 1.6 *Sei $S \subset \mathbb{R}^2$ eine Punktmenge in der euklidischen Ebene in allgemeiner Lage. Für beliebige p, q, r ∈ S gilt dann:*

1. *p, q, r bilden genau dann ein Delaunay-Dreieck in S, wenn der Kreis C(p,q,r) um $\triangle pqr$ keinen weiteren Punkt aus S enthält (*Leerer-Umkreis-Kriterium*).*
2. *Eine Triangulation T(S) ist genau dann eine Delaunay-Triangulation DT(S), wenn alle Dreiecke aus T(S) Delaunay-Dreiecke sind.*

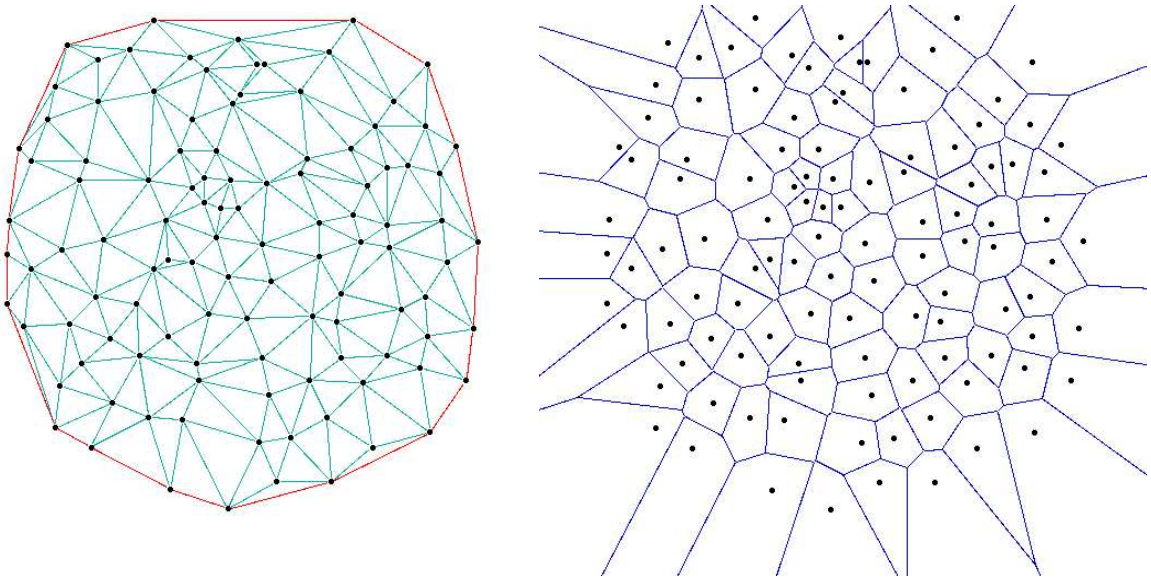

Abbildung 1.2: Delaunay-Triangulation und Voronoi-Diagramm einer planaren Punktmenge

Die Delaunay-Triangulation ist eng verbunden mit einem weiteren sehr bekannten Objekt der Algorithmischen Geometrie, dem *Voronoi-Diagramm* (siehe Abbildung 1.2). Dieses Diagramm ordnet jedem Punkt der Ebene seinen nächsten Nachbarn aus einer gegebenen Menge von Konfigurationspunkten zu. Dazu definiert man:

Definition 1.7 *Sei $S \subset \mathbb{R}^2$ eine Punktmenge in der euklidischen Ebene in allgemeiner Lage und seien $p, q \in S$. d(p,q) bezeichne den euklidischen Abstand zwischen den Punkten p und q. Dann ist*

1. *die Voronoi-Region von p definiert als $VR(p) = \{r \in \mathbb{R}^2 : d(p,r) \leq d(p,q)\, \forall q \in S\}$.*
2. *das Voronoi-Diagramm von S definiert als $VD(S) = \bigcup_{p \in S} VR(p)$.*

Bezeichnung 1.8 *Die Ränder einer Voronoi-Region nennt man* Voronoi-Kanten, *die Schnittpunkte der Voronoi-Kanten heißen* Voronoi-Knoten.

Es ist durchaus möglich, die Delaunay-Triangulation als duales Konstrukt zum Voronoi-Diagramm zu definieren und dann das Leerer-Umkreis-Kriterium daraus zu folgern. Hier wird der umgekehrte Weg eingeschlagen, um eine bessere Verständlichkeit zu erreichen:

Satz 1.9 *Sei $S \subset \mathbb{R}^2$ eine Punktmenge in der euklidischen Ebene in allgemeiner Lage. Dann ist DT(S) graphentheoretisch dual zu VD(S), d. h. zu jeder Voronoi-Region in VD(S) gibt es genau einen Knoten*

in DT(S) und die Knoten in DT(S) sind genau dann durch eine Kante miteinander verbunden, wenn ihre korrespondierenden Voronoi-Regionen eine gemeinsame Voronoi-Kante haben.

Beweis: Der Beweis, daß der duale Graph zu VD(S) eine Triangulation T von S ist, findet sich bei Preparata und Shamos [59, S. 209]. Dort wird auch gezeigt, daß der Kreis um einen Voronoi-Knoten durch die drei Punkte aus S, die diesen Voronoi-Knoten induzieren, keinen weiteren Knoten aus S enthält. Daraus folgt, daß es kein Dreieck in T gibt, dessen Umkreis einen anderen Punkt als die drei erzeugenden enthält. Somit ist T die Delaunay-Triangulation von S. □

Aufgrund dieser Dualität kann man die Lösung für eines der beiden Probleme berechnen und das Ergebnis dann sehr einfach mit Komplexität $O(n)$ in die Lösung des anderen Problems umwandeln.

Bezeichnung 1.10 *Bestehe ein konvexes Viereck aus den Punkten a, b, c und d und den sie verbindenden Kanten $\overline{ab}$, $\overline{bc}$, $\overline{cd}$ und $\overline{ad}$ (s. Abbildung 1.3). Bildet weiterhin $\overline{ac}$ eine Dreieckskante in diesem Viereck und ersetzt man sie durch $\overline{bd}$, so nennt man diesen Vorgang der Kantenersetzung* Kantenwechsel *oder nach dem englischen Vorbild* Kantenflip *(*edge flip*).*

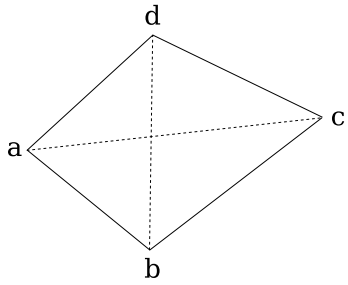

Abbildung 1.3: Konvexes Viereck mit zwei möglichen Diagonalen

Satz 1.11 *(vgl. Klein [47, S. 236ff.]) Sei S eine n-elementige planare Punktmenge in allgemeiner Lage. Dann hat DT(S) die lexikographisch größte Winkelfolge unter allen Triangulationen von S.*

Beweisidee: Sei T eine Triangulation von S. Gibt es kein Dreieck von T, dessen Umkreis einen anderen Punkt aus S im Innern enthält, so ist T = DT(S), weil jede Triangulation von S die gleiche Anzahl von Dreiecken hat. Gibt es ein solches Dreieck $\triangle pqr$, so liegt in seinem Umkreis (aber nicht im Dreieck selbst) ein Punkt t. Liege o. B. d. A. t über der Kante $\overline{pr}$ und q darunter. Sind t und das Dreieck so gewählt, daß der Winkel von t zu p und r maximal ist, muß auch $\triangle prt$ ein Dreieck von T sein. Nun kann man durch das Ersetzen der Kante $\overline{pr}$ durch die Kante $\overline{qt}$ (edge flip) eine neue Triangulation erstellen, deren Winkelfolge größer ist. Man kann also die Winkelfolge jeder von DT(S) verschiedenen Triangulation durch einen Kantenflip vergrößern. Aufgrund dessen muß die Winkelfolge von DT(S) maximal sein. Für den vollständigen Beweis siehe Klein [47, S. 236ff.]. □

Die Delaunay-Triangulation sorgt also dafür, daß man Dreiecke erhält, die über die gesamte Triangulation gesehen maximal große Winkel haben. An bestimmten Stellen können zwar andere

Triangulationen für günstigere Winkel sorgen, dies geht dann allerdings mit einem höheren Qualitätsverlust an anderen Stellen der Dreieckszerlegung einher. Leicht einzusehen ist durch den obigen Satz auch, daß durch DT(S) der kleinste Winkel aller möglichen Triangulationen maximiert wird. Wie bereits angedeutet, ist es für numerische Berechnungen wichtig, daß die Innenwinkel der Dreiecke nicht unter einen bestimmten Schwellenwert fallen. Tun sie dies, kann die Berechnung ungenau werden oder völlig entarten. Um dies bestmöglich zu verhindern, findet bei der Approximation von Oberflächen zur numerischen Simulation durch den Computer die Delaunay-Triangulation Verwendung. Die dazu verwendeten Datenstrukturen sollen nun genauer betrachtet werden:

Definition 1.12 *(vgl. van Kreveld [48, S. 40]) Ein TIN (von engl.:* triangulated irregular network*) speichert eine endliche Menge von Punkten mit ihrem Höhenwert, die Verteilung der Punkte ist unregelmäßig. Auf diesen Punkten ist eine planare Triangulation gegeben.*

Definition 1.13 *(vgl. Bern und Eppstein [14, S. 2]): Ein strukturiertes Netz (engl.:* structured mesh*) in zwei Dimensionen ist üblicherweise ein regelmäßiges Gitter aus Quadraten, dessen Punkte (außer an den Rändern) isomorphe Nachbarschaftsbeziehungen haben. Eine Dimension höher wird ein regelmäßiges Gitter aus Würfeln verwendet. Ein unstrukturiertes Netz (engl:* unstructured mesh*) ist meist eine Triangulation mit beliebigen Nachbarschaftsbeziehungen.*

Nachfolgend wird der Begriff TIN für Netze verwendet, die man als 2,5-dimensional bezeichnet. Sie modellieren Oberflächen, die zu einem beliebigen Wert der Definitionsebene maximal einen Wert des Wertebereichs annehmen können (Funktionscharakter), wie etwa bei der Terrainmodellierung. Im Unterschied dazu soll der Begriff Mesh solche Strukturen bezeichnen, die zwar aus zweidimensionalen Elementen (also Dreiecken, nicht Tetraedern) zur Oberflächenmodellierung bestehen, deren Oberflächen sich im Raum allerdings auch so krümmen dürfen, daß sie keinen Funktionscharakter haben. TIN und Mesh werden unter den Oberbegriffen *Netz* und *Gitternetz* zusammengefaßt.

Der Höhenwert (oder Tiefenwert, etwa bei der Profilanalyse von Meeresböden) zu TIN-Punkten gibt demnach bei der Terrainmodellierung an, welche Höhe dieser Ort in der Realität aufweist. Allerdings macht die planare Delaunay-Triangulation keinen Gebrauch von diesen Höhendaten, so daß unerwünschte Effekte auftreten können, die man *künstliche Dämme* nennt.

Beispiel 1.14 *Bei der Abbildung eines Tals zwischen zwei Gebirgsketten in einem Computermodell sollte der Talverlauf durch Kanten im TIN nachvollziehbar sein. Nun kann es durch die Verwendung der Delaunay-Triangulation und deren Vernachlässigung der Höhendaten allerdings dazu kommen, daß der Talverlauf durch eine Querkante zwischen den beiden Gebirgsketten durchbrochen wird (s. Abb. 1.4).*

Diese Querkante stellt einen künstlichen Damm dar, der eine Abweichung des Modells von der Realität darstellt und sich daher bei der Simulation anhand des Modells negativ auswirkt. Um die nützlichen Eigenschaften der Delaunay-Triangulation im wesentlichen zu erhalten, aber gleichzeitig künstliche Dämme möglichst zu vermeiden, wird das Leerer-Umkreis-Kriterium aufgeweicht, was im folgenden Abschnitt zu Delaunay-Triangulationen höherer Ordnung führt.

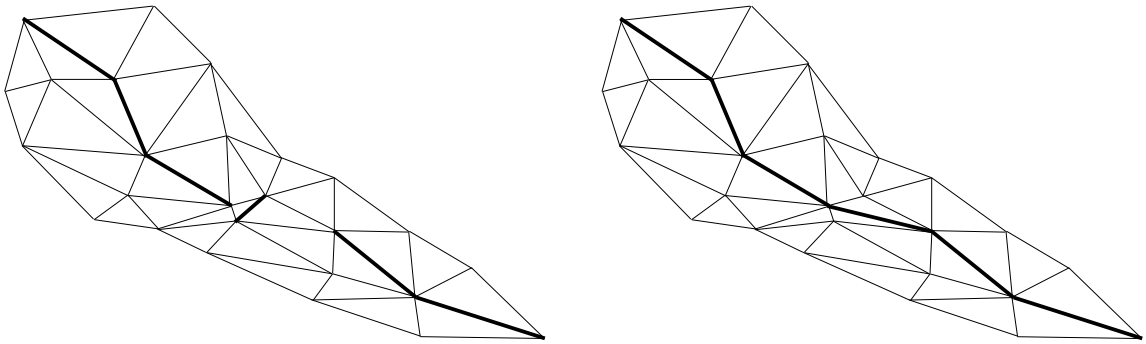

Abbildung 1.4: Künstlicher Damm in einem TIN (links) und seine Auflösung durch einen Kantenwechsel (rechts)

1.2 Höhere Ordnungen

Damit man unerwünschte Kanten aus der Delaunay-Triangulation zugunsten erwünschter Kanten entfernen kann, ohne die Winkeleigenschaften überall zu zerstören, läßt man je Dreieck eine begrenzte Anzahl von Punkten in seinem Umkreis zu.[5]

Definition 1.15 *Sei $S \subset \mathbb{R}^2$ eine Punktmenge in der euklidischen Ebene in allgemeiner Lage. Für $u, v, w \in S$ gilt:*

1. *Eine Kante $\overline{uv}$ ist eine Delaunay-Kante k-ter Ordnung (kurz: k-OD-Kante), falls es einen Kreis durch u und v gibt, der höchstens k Punkte aus P in seinem Inneren enthält.*
2. *Ein Dreieck $\triangle uvw$ ist ein Delaunay-Dreieck k-ter Ordnung (kurz: k-OD-Dreieck), falls der Kreis durch u, v und w höchstens k Punkte aus P in seinem Inneren enthält.*
3. *Eine Triangulation von P is eine Delaunay-Triangulation k-ter Ordnung (kurz: k-ODT) von P, falls jedes Dreieck der Triangulation ein k-OD-Dreieck ist.*

Für k = 0 entspricht die obige Definition der normalen Delaunay-Triangulation.

Lemma 1.16 *Sei S eine Punktmenge in der Ebene in allgemeiner Lage.*

1. *Jede Kante eines k-OD-Dreiecks ist eine k-OD-Kante.*
2. *Jede Kante einer k-OD-Triangulation ist eine k-OD-Kante.*
3. *Jede k-OD-Kante mit k > 0, die keine Delaunay-Kante ist, schneidet eine Delaunay-Kante.*

Beweis: Man betrachte für die erste Aussage den Kreis C um ein k-OD-Dreieck. C enthält höchstens k weitere Punkte aus S. Wegen allgemeiner Lage kann man C nun so verändern, daß er nur noch durch zwei Punkte verläuft und der dritte außerhalb von C liegt, ohne daß ein weiterer anderer Punkt aus S in sein Inneres gelangt. Diese beiden Punkte, durch die C nun noch verläuft, sind die Endpunkte einer k-OD-Kante. Der zweite Teil des Lemmas erschließt sich nun unmittelbar, der dritte Teil folgt wegen der Maximalität der Kantenmenge einer Triangulation. □

[5] Wo nicht explizit auf eigene Arbeit hingewiesen wird, bereitet dieses gesamte Unterkapitel 1.2 (inkl. Abschnitt 1.2.1) die Ergebnisse der Originalarbeit von Gudmundsson et al. [39] auf, die das Thema höhere Ordnungen bei Delaunay-Triangulationen eingeführt hat.

Bemerkung 1.17 *Die Gegenrichtung von Lemma 1.1 gilt nicht, da nicht jedes Dreieck, das drei k-OD-Kanten hat, auch ein k-OD-Dreieck ist. Man betrachte dazu ein Dreieck t und seinen Umkreis c; die Dreiecksseiten von t bilden dabei drei Sehnen von c. Es liege nun zwischen jeder Sehne und dem Rand von c jeweils ein weiterer Punkt aus S. Die Kanten von t sind alle 1-OD-Kanten, in c liegen allerdings drei weitere Punkte aus s, so daß t kein 1-OD-Dreieck ist.*

Definition 1.18 *Für jede k-OD-Kante $\overline{uv}$ und jede Delaunay-Kante $\overline{sp}$, die $\overline{uv}$ schneidet, gilt: Der Kreis $C(u, v, s)$ enthält p.*

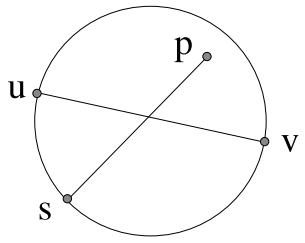

Abbildung 1.5: $C(u, v, s)$ enthält p

Folgerung 1.19 *Man betrachte eine k-OD-Kante $\overline{uv}$. Jeder Kreis durch u und v, der keine Knoten links (bzw. rechts) von $\vec{vu}$ enthält, enthält alle Knoten, die inzident zu $\overline{uv}$ schneidenden Delaunay-Kanten sind, rechts (bzw. links) von $\vec{vu}$.*

Beweis zu Beob. 1.18 und Folg. 1.19[6]: Die Beobachtung läßt sich nicht nur graphisch nachvollziehen (Abb. 1.5). Der Mittelpunkt des Kreises $C(u, v, s)$ wandert auf dem Bisektor von u und s im Vergleich zu $C(s, p, u)$ Richtung v, zusätzlich wird sein Radius größer. Somit muß p in ihm enthalten sein.

Weil die Endpunkte von Delaunay-Kanten, die $\overline{uv}$ schneiden, nicht links (bzw. rechts) von $\overline{uv}$ im Kreis um u, v und den dritten Punkt des k-OD-Dreiecks liegen, müssen sie nach Beobachtung 1.18 rechts (bzw. links) in diesem Kreis enthalten sein. □

Es besteht eine enge Verbindung zu Voronoi-Diagrammen höherer Ordnung[7], mit der man abschätzen kann, wie viele Punkte einen Beitrag zu einer k-OD-Kante leisten können:

Satz 1.20 *Sei $S \subset \mathbb{R}^2$ eine n-elementige Punktmenge, sei $k \leq n/2 - 2$ und seien $u, v \in S$. Die Kante $\overline{uv}$ ist eine k-OD-Kante genau dann, wenn es zwei zueinander inzidente Flächen F_1 und F_2 im Voronoi-Diagramm (k+1)-ter Ordnung von S derart gibt, daß u in der Menge der Punkte ist, die F_1 induziert und v zu der Menge der Punkte gehört, die F_2 induziert.*

Beweis: Sei m die kleinste ganze Zahl für zwei Punkte $u, v \in S$ derart, daß der Bisektor von u und v im Voronoi-Diagramm m-ter Ordnung von S erscheint. Weil auf einer Seite der Geraden durch u und v mindestens $n/2 - 1$ Punkte aus S liegen, erscheint der Bisektor von u und v in allen

[6] Dieser Beweis wurde in der zugrunde liegenden Originalarbeit ausgelassen.
[7] Eine kurze Einführung in Voronoi-Diagramme höherer Ordnung gibt Kapitel 4.2 dieser Arbeit. Dort finden sich auch weitergehende Literaturverweise.

Voronoi-Diagrammen der Ordnungen von m bis $n/2 - 1$. Dieser Bisektor teilt zwei Flächen, die Teilmengen von S wie im Satz beschrieben repräsentieren. □

Weil Voronoi-Diagramme der Ordnung $k+1$ in der Ebene $O((k+1)(n-k-1))$ Flächen haben (s. Fortune [34, S. 383]), folgt, daß $O(n+nk)$ Paare von Punkten eine k-OD-Kante erzeugen können. Diese Paare kann man nach Ramos [60] in erwarteter Zeit von $O(nk2^{c\,log^*k} + n\,log\,n)$ berechnen.

Auf dem Weg zu einem Algorithmus zur Bestimmung einer k-OD-Triangulation muß man zunächst bestimmen, ob eine k-OD-Kante bzw. ein k-OD-Dreieck auf jeden Fall in eine k-OD-Triangulation integriert werden kann. Daher definiert man:

Definition 1.21 *Sei S eine planare Punktmenge. Eine k-OD-Kante $\overline{uv}$ mit $u, v \in S$ ist* nützlich, *wenn es eine k-OD-Triangulation gibt, die $\overline{uv}$ beinhaltet. Ein k-OD-Dreieck $\triangle uvw$ mit $u, v, w \in S$ ist* gültig, *wenn es keinen Punkt aus S in seinem Innern enthält und seine drei Kanten alle nützlich sind.*

Um festzustellen, ob eine Kante nützlich ist, betrachtet man zunächst das Konzept der Hülle einer k-OD-Kante. Mit dem anschließenden Algorithmus zur Triangulation einer Hülle kommt man dann der Formulierung eines Algorithmus für die k-OD-Triangulation einen Schritt näher.

Definition 1.22 *Die Hülle einer k-OD-Kante $\overline{uv}$ ist der Abschluß der Vereinigung aller Delaunay-Dreiecke, deren Inneres $\overline{uv}$ schneidet.*

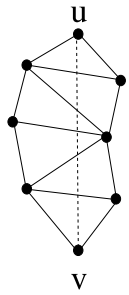

Abbildung 1.6: Hülle einer k-OD-Kante

Algorithmus 1.23 *Triangulation einer Hülle*

Sei $\overline{uv}$ eine k-OD-Kante und sei p_1 der Punkt rechts von $\vec{vu}$ derart, daß der Teil von $C(u, v, p_1)$ rechts von $\vec{vu}$ leer ist. Man beachte, daß p_1 ein Knoten auf dem Rand der Hülle von $\overline{uv}$ sein muß. Nun füge man die zwei Kanten $\overline{up_1}$ und $\overline{vp_1}$ zum Graphen hinzu und fahre rekursiv für diese beiden Kanten fort, bis die Hülle von $\overline{uv}$ rechts von $\vec{vu}$ fertig trianguliert ist. Analog verfährt man auf der linken Seite. Die erhaltene Triangulation nennt man die *gierige Triangulation* (engl.: *greedy triangulation*) der Hülle von $\overline{uv}$. □

Das folgende Lemma liefert nun die Grundlagen für die Bestimmung, ob eine k-OD-Kante nützlich ist:

Lemma 1.24 *Sei $\overline{uv}$ eine k-OD-Kante, sei s_1 der Punkt links von $\vec{vu}$ derart, daß der Kreis $C(u, s_1, v)$ keinen Punkt links von $\vec{vu}$ enthält. Sei s_2 analog für die rechte Seite definiert. Dann gilt: Die Kante $\overline{uv}$ ist eine nützliche k-OD-Kante genau dann, wenn $\triangle uvs_1$ und $\triangle uvs_2$ k-OD-Dreiecke sind, d. h. wenn die Umkreise um die Dreiecke jeweils höchstens k Punkte in ihrem Innern enthalten.*

Beweis: Wir zeigen zunächst, daß $\overline{uv}$ nicht nützlich ist, wenn $\triangle us_1v$ kein k-OD-Dreieck ist, also die Kontraposition der Richtung "⇒". Sei also $\triangle us_1v$ kein k-OD-Dreieck. Dann enthält der Kreis $C(u, s_1, v)$ mehr als k Punkte in seinem Innern rechts von $\vec{vu}$. Angenommen es gibt dennoch eine k-OD-Triangulation, die $\overline{uv}$ beinhaltet. Sei $\triangle us_iv$ das Dreieck in T, das dann links von $\vec{vu}$ liegt. Der Punkt s_i muß derart liegen, daß $\overline{vs_i}$ $\overline{us_1}$ schneidet oder daß $\overline{us_i}$ $\overline{vs_1}$ schneidet. Wegen der Symmetrie können wir ohne Einschränkung den letzteren Fall annehmen. Seien p_1 und p_2 die beiden Punkte aus S derart, daß $\triangle s_1p_1p_2$ zu T gehört und das Dreieck $\triangle uvs_1$ schneidet. Es ist möglich, daß $p_1 = u$ oder $p_2 = s_i$ gilt oder sogar beides. Der Kreis $C(s_1, p_1, p_2)$ beinhaltet den ganzen Teil des Kreises $C(u, v, s_1)$ rechts von $\vec{vu}$, weil p_1 und p_2 außerhalb von $C(u, v, s_1)$ liegen, siehe Abb. 1.7 (einer von beiden darf auch auf dem Kreisrand liegen). Daher enthält $C(s_1, p_1, p_2)$ mindestens k+2 Punkte, nämlich k+1 Punkte innerhalb von $C(u, s_1, v)$ und außerdem den Punkt v. Somit folgt, daß $\triangle us_iv$ kein k-OD-Dreieck sein kann, also existiert keine k-OD-Triangulation und $\overline{uv}$ ist nicht nützlich.

Zum Beweis der Rückrichtung betrachte man die gierige Triangulation von $\overline{uv}$. Nach Voraussetzung sind $\triangle uvs_1$ und $\triangle uvs_2$ k-OD-Dreiecke. Weil die Rekursion bei der gierigen Triangulation für die Kanten $\overline{vs_1}$, $\overline{vs_2}$,$\overline{us_1}$ und $\overline{us_2}$ zu Kreisen führt, die weniger Punkte als $C(u, s_1, v)$ oder $C(u, s_2, v)$ enthalten, müssen die zugehörigen Dreiecke auch k-OD-Dreiecke sein. Somit ist $\overline{uv}$ nützlich. □

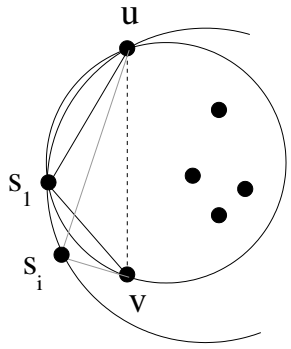

Abbildung 1.7: Wenn $\triangle us_1v$ kein k-OD-Dreieck ist, ist $\overline{uv}$ nicht nützlich.

Lemma 1.25 *Die Delaunay-Kanten, die eine nützliche k-OD-Kante $\overline{uv}$ schneiden, sind mit höchstens k Knoten auf jeder Seite einer k-OD-Kante verbunden.*

Beweis: Angenommen es gibt mehr als k Knoten, die zu Delaunay-Kanten inzident sind, die wiederum $\overline{uv}$ schneiden, etwa rechts von $\vec{vu}$. Sei s_1 der Punkt links von $\vec{vu}$ derart, daß der Kreis $C(u, s_1, v)$ keinen Punkt links von $\vec{vu}$ enthält. Dann muß dieser Kreis alle Punkte rechts von $\vec{vu}$ enthalten. Da dies mehr als k sind, ist $\triangle us_1v$ kein k-OD-Dreieck. Daraus folgt, daß $\overline{uv}$ nicht nützlich ist, was der Voraussetzung widerspricht. □

Folgerung 1.26 *Der Algorithmus zur Triangulation einer Hülle benötigt $O(k^2)$ Zeitschritte.*

Beweis: Für jede Kante, die bei der Triangulation der Hülle gefunden wird, muß man $O(k)$ Punkte betrachten, um den Punkt zu finden, der links bzw. rechts der Kante zu einem leeren Umkreis führt. Dies ist durch Minimierung des Delaunay-Abstands für die verbliebenen $O(k)$ Punkte möglich. Dazu berechnet man für jeden Punkt p auf der noch unbearbeiteten Seite der aktuellen Kante e den Kreis durch p und die Endpunkte von e. Dieser Kreis habe den Radius r und den Mittelpunkt c. Der korrekte nächste Punkt p ist derjenige, für den folgende Funktion dd minimal ist [22]:

```
dd(e,p) := IF (c liegt auf derselben Seite von e wie p) THEN r ELSE -r
```

Dieser Punkt p läßt sich für jede Kante mit Kosten in Höhe von $O(k)$ berechnen. Es müssen für die vollständige Triangulation der Hülle $O(k)$ Kanten betrachtet werden, so daß die Gesamtkomplexität $O(k^2)$ beträgt.[8] □

Mit Hilfe der obigen Erkenntnisse ist es nun mit dem folgenden Algorithmus möglich, aus allen k-OD-Kanten die nützlichen herauszufiltern.

Algorithmus 1.27 *Bestimmung nützlicher k-OD-Kanten*

1. Preprocessing: Erstelle das Voronoi-Diagramm k-ter Ordnung und die Delaunay-Triangulation von S.
2. Für jede Kante $\overline{uv}$:
   (a) Lokalisiere die zugehörigen Punkte u und v in der Delaunay-Triangulation, traversiere von u nach v entlang $\overline{uv}$ von Delaunay-Dreieck zu Delaunay-Dreieck und speichere dabei alle geschnittenen Delaunay-Kanten. Falls dies mehr als 2k-1 sind, ist $\overline{uv}$ nicht nützlich.
   (b) Bestimme die Endpunkte der geschnittenen Delaunay-Kanten links und rechts von $\vec{vu}$. Falls auf einer Seite mehr als k sind, ist $\overline{uv}$ nicht nützlich.
   (c) Bestimme s_1 und s_2 wie in Lemma 1.24 definiert. Berechne, wie viele Punkte - höchstens k (nützlich) oder doch mehr (nicht nützlich) - in den Umkreisen $C(u, s_1, v)$ und $C(u, s_2, v)$ liegen. Benutze dafür das Voronoi-Diagramm k-ter Ordnung, um den k-test-nächsten Nachbarn zu finden und dessen Abstand zum Mittelpunkt des aktuellen Kreises mit dem Kreisradius zu vergleichen. Die Datenstruktur beantwortet Fragen dieser Art in $O(log\, n)$ Zeit.

Damit benötigt der gesamte Test auf Nützlichkeit ohne Preprocessing pro Kante $O(log\, n + k)$ Zeit.

Satz 1.28 *Sei P eine Punktmenge in der Ebene. Man kann alle nützlichen k-OD-Kanten von P in $O(nk^2 + kn\, log\, n)$[9] Zeit berechnen.*

[8] Dieser Beweis wurde selbst geführt und entstammt nicht der Originalarbeit.

[9] Hier wird von der Originalarbeit abgewichen, die eine Zeitkomplexität von $O(nk^2 + n\, log\, n)$ angibt. Allerdings ist jenes Ergebnis nicht nachvollziehbar und wird gemäß einer E-Mail-Nachricht von Joachim Gudmundsson in einem künftigen Journal-Artikel zu Gunsten der Lösung dieser Arbeit revidiert.

Beweis: Mit einem Algorithmus von Ramos [60] für Voronoi-Diagramme höherer Ordnung kann man alle k-OD-Kanten in erwarteter Zeit von $O(nk2^{c\,log^*\,k} + n\,log\,n)$ berechnen. Da $O(nk)$ Kanten zu testen sind, braucht man in Abweichung zur Originalarbeit insgesamt $O(nk(log\,n + k)) = O(nk^2 + kn\,log\,n)$ erwartete Zeit, um alle nützlichen k-OD-Kanten zu berechnen. □

Gudmundsson et al. [39, S. 91] führen aus, daß dieser Algorithmus für kleine k effizient ist, es für große k ($k > \sqrt{n}$) aber effizientere Lösungen auf der Basis von Partitionsbäumen gibt. In der Praxis dürften so große k nicht auftreten. Statt dessen kann man dort von konstanten Werten für k ausgehen, weshalb die Alternativen hier nicht näher betrachtet werden. Ein anderer Algorithmus zur Triangulation der Hülle einer k-OD-Kante mit Zeitkomplexität $O(k\,log^3 k)$, der geeignetes Preprocessing erfordert, wird aus demselben Grund nicht weiter betrachtet.

1.2.1 Delaunay-Triangulationen erster Ordnung

Wie bereits erwähnt, spielen für praktische Belange normalerweise nur konstante Ordnungen eine Rolle. Denn schon mit Ordnung 1 lassen sich Netze bezüglich verschiedener Kriterien optimieren, wie in Kapitel 4.1.2 ausgeführt wird. Hier wird nach einem einführenden Lemma zunächst gezeigt, wie man eine 1-OD-Triangulation berechnet.

Lemma 1.29 *Jede nützliche 1-OD-Kante schneidet höchstens eine nützliche 1-OD-Kante. Eine dieser beiden Kanten ist eine Delaunay-Kante. Anders ausgedrückt: Es gibt keine nützlichen Nicht-Delaunay-1-OD-Kanten, die sich schneiden.*

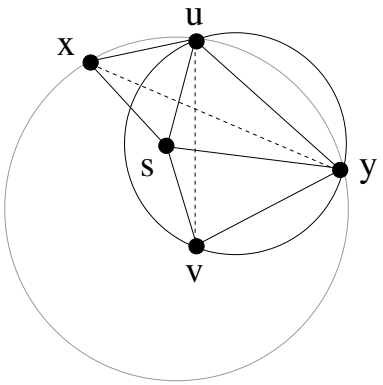

Abbildung 1.8: Jede nützliche 1-OD-Kante schneidet höchstens eine nützliche 1-OD-Kante

Beweis: Wegen Lemma 1.25 schneidet jede Delaunay-Kante höchstens eine nützliche 1-OD-Kante. Es bleibt folglich nur zu zeigen, daß das Lemma für jede 1-OD-Kante $\overline{uv}$ gilt, die nützlich und keine Delaunay-Kante ist. Sei dazu $\overline{sy}$ die Delaunay-Kante, die $\overline{uv}$ schneidet. Dann sind $\overline{us}$, $\overline{vs}$, $\overline{uy}$ und $\overline{vy}$ Delaunay-Kanten. Falls es eine nützliche 1-OD-Kante gibt, die die Delaunay-Kante $\overline{us}$ schneidet, muß sie gemäß Lemma 1.25 mit dem Punkt y verbunden sein, weil $\overline{sy}$ sonst zwei 1-OD-Kanten schneiden würde. Weil $\overline{xy}$ eine nützliche 1-OD-Kante ist, muß x durch Delaunay-Kanten mit u und s verbunden sein. Man betrachtet nun den Kreis $C(u, v, y)$. Nach Beobachtung 1.18 und wegen der Nützlichkeit von $\overline{uv}$ enthält $C(u, v, y)$ ausschließlich s. Den Kreis $C(u, y, x)$ erhält

man, indem man den vorigen vergrößert und dabei von v abläßt, bis der Kreisrand x erreicht. Nun liegen sowohl v als auch s in $C(u, y, x)$, weshalb $\triangle xuy$ kein 1-OD-Dreieck sein kann. Somit ist $\overline{xy}$ nicht nützlich, was zum Widerspruch zur Annahme führt und folglich die Behauptung korrekt ist. Wegen der Symmetrie gilt die Argumentation für alle Kanten, die $\overline{vs}$, $\overline{uy}$ oder $\overline{vy}$ schneiden. □

Definition 1.30 *Zwei Kanten e_1 und e_2 in einer Triangulation T heißen* unabhängig, *wenn sie nicht zu demselben Dreieck gehören.*

Folgerung 1.31 *Jede 1-OD-Triangulation kann aus einer Delaunay-Triangulation durch Kantenflips unabhängiger Delaunay-Kanten erhalten werden. Bei gegebener Delaunay-Triangulation können alle Kandidaten*[10] *für 1-OD-Kanten in linearer Zeit gefunden werden. Festzustellen, ob sie nützlich sind, benötigt eine Zeit von $O(n \log n)$.*

Beweis: Weil eine nützliche 1-OD-Kante e von höchstens einer Delaunay-Kante e' geschnitten wird (Lemma 1.29), müssen in einem solchen Fall beide zu demselben konvexen Viereck gehören. Da e eine 1-OD-Kante und nützlich ist, erhält man durch einen Kantenflip von e' zu e aus einer 1-OD-Triangulation wiederum eine 1-OD-Triangulation. Kantenwechsel unabhängiger Kanten können sich nicht beeinflussen (weil unabhängige Kanten nicht zu demselben Viereck gehören können, in dem die Kante geflippt wird) und ergeben eine korrekte 1-OD-Triangulation.

Hat man die Delaunay-Triangulation mit ihren $O(n)$ Kanten gegeben, findet man in $O(n)$ Zeit ihre konvexen Vierecke und die zugehörigen Kandidaten für 1-OD-Kanten. Mit Hilfe des Voronoi-Diagramms zweiter Ordnung, zu berechnen in $O(n \log n)$ Schritten, kann man dann wie bekannt die Nützlichkeit dieser Kanten in der Zeit $O(n \log n)$ bestimmen.[11] □

Durch die oben erläuterten Verfahren hat man die Möglichkeit, Delaunay-Triangulationen erster Ordnung sowie nützliche k-OD-Kanten für beliebiges $k < n$ seriell zu berechnen. Allerdings sind die Datenmengen, die üblicherweise bei der Simulation technischer Bauteile oder natürlicher Phänomene zu bearbeiten sind, so groß oder die dazu nötigen Berechnungen so kompliziert, daß eine serielle Berechnung gar nicht möglich wäre oder zu lange dauern würde. Das folgende Unterkapitel gibt daher einen Einblick, wie man solche Probleme allgemein durch Parallelverarbeitung bewältigt.

1.3 Parallelverarbeitung

Für viele Probleme genügt es nicht einfach, sie irgendwann mit Hilfe eines Computersystems zu lösen. Statt dessen muß die Lösung on-line, also direkt während der interaktiven Ausführung eines Programms durch den Benutzer, oder bis zu einem bestimmten Zeitpunkt vorliegen, damit diese Information dem Anwender noch von Nutzen ist. Gerade in den Naturwissenschaften sind viele Probleme aber gleichzeitig hochkomplex und/oder erfordern die Verarbeitung riesiger Datenmengen. Beispielsweise sammeln Meteorologen an vielen Meßpunkten der Erde in kleinen Abständen viele Meßwerte, um mit deren Hilfe das Wetter vorherzusagen.

[10] Ein Kandidat für eine 1-OD-Kante ist die Diagonale in einem konvexen Viereck, die die Delaunay-Kante dieses Vierecks schneidet.

[11] Dieser Beweis wurde selbst geführt und entstammt nicht der Originalarbeit.

Wenn nun beispielsweise ein einzelner Prozessor zur Bestimmung des Wetters der kommenden drei Tage aufgrund der Schwierigkeit des Problems bzw. der Menge der zu verarbeitenden Daten fünfzehn Tage ununterbrochen rechnen müßte, um eine einigermaßen zuverlässige Vorhersage zu treffen, so wäre sein Ergebnis wertlos, da der Betrachtungszeitraum der Vorhersage längst Vergangenheit ist. Außerdem ist es durchaus fraglich, ob der Speicher eines einzelnen Prozessors eine ausreichende Größe für die gesammelten Daten und die für die Berechnung benötigten Datenstrukturen hat.

Der Einwand, dieses Problem ließe sich durch die immer besser werdende Halbleitertechnologie lösen, läßt sich nicht aufrechterhalten. Zum einen verdoppelt sich die Leistungsfähigkeit eines Prozessors nach Moores Gesetz[12] nur etwa alle zwei Jahre, zum anderen ist selbst diese Verdopplung für die zur Zeit verwendete Halbleitertechnologie vermutlich nicht lange über das Jahr 2020 hinaus zu erwarten, da dem physikalische Gesetzmäßigkeiten (etwa die Lichtgeschwindigkeit) entgegenstehen.

Mit der Lösung dieses Problems der Ressourcenknappheit von Zeit und Speicher beschäftigt sich daher die Parallelverarbeitung. Dabei schaltet man eine Menge von Prozessoren auf eine geeignete Weise zusammen und teilt das Problem so auf diese Prozessoren auf, daß eine möglichst hohe Beschleunigung der Berechnungszeit erzielt wird.

Die Beschleunigung (engl.: *speedup*) eines parallelen Algorithmus berechnet sich durch:

$$Beschleunigung := \frac{Laufzeit\,des\,sequentiellen\,Algorithmus}{Laufzeit\,des\,parallelen\,Algorithmus}$$

Sie hängt dabei von der verwendeten Hardware (bspw. Prozessoren, Art der Zusammenschaltung), dem Algorithmus und dem Problem selbst ab. Eine zweite Kenngröße eines parallelen Algorithmus ist die Effizienz (engl.: *efficiency*), berechnet durch:

$$Effizienz := \frac{Speedup}{Anzahl\,der\,Prozessoren\,p}$$

Grundsätzlich strebt man als Speedup bei p verwendeten Prozessoren das (unter Vernachlässigung architektureller Effekte wie Caching) theoretische Optimum von p (also eine Effizienz von 1 oder 100%) an. Ein paralleler Algorithmus ist demnach besonders effizient, wenn er das Potential der zur Verfügung stehenden Prozessoren sehr gut ausschöpft. Steigt der Speedup eines Algorithmus proportional zu der Anzahl der eingesetzten Prozessoren, so spricht man von einer guten *Skalierbarkeit* des Algorithmus, eine offensichtlich wünschenswerte Eigenschaft.

Es gibt Probleme, bei denen eine Beschleunigung der Berechnungszeit durch Parallelverarbeitung aller Voraussicht nach nicht erzielt werden kann. Sie stellen die Klasse der P-vollständigen Probleme dar, für die es vermutlich keine schnellen[13] parallelen Algorithmen gibt (s. JaJa [44, S. 529ff.]). Diese Vermutung ist zwar bisher unbewiesen, allerdings wird in Fachkreisen allgemein von ihrer Gültigkeit ausgegangen. Probleme, die aufgrund ihrer Struktur nicht effizient paralle-

[12] Moores Gesetz, aufgestellt vom Intel-Mitbegründer Gordon E. Moore, ist im eigentlichen Sinne kein Gesetz, sondern eine auf empirisch gewonnenen Daten basierende Feststellung aus dem Jahre 1965 [54], wonach man in den kommenden zehn Jahren die Anzahl der Transistoren auf derselben Chipfläche (und damit die Leistungsfähigkeit eines Prozessors) jedes Jahr verdoppeln könne. Auch wenn sich die beobachtete Zeitspanne auf 18-24 Monate verlangsamt hat, gilt Moores Feststellung im Kern immer noch.

[13] "Schnell" heißt in diesem Zusammenhang, daß ein Problem bei einem Einsatz von polynomiell vielen Prozessoren polylogarithmische Zeitkomplexität (d. h. $O(log^k n)$ für festes k) hat.

lisierbar sind, werden in der vorliegenden Arbeit nicht betrachtet. Die Problematik sei nur der Vollständigkeit halber erwähnt, um dem Eindruck entgegenzuwirken, daß man zu jedem Problem eine schnelle Parallelisierung finden kann.

Das genaue Gegenteil dazu sind Probleme, die sich "auf triviale Weise" parallelisieren lassen. Davon spricht man im Allgemeinen, wenn die Aufteilung der Eingabe auf die Prozessoren ohne aufwendige Berechnung möglich ist und die Prozessoren während der Abarbeitung ihres Teilproblems nicht oder nur sehr wenig miteinander kommunizieren müssen. Bei solchen Problemen ist ein Speedup in der Nähe des Optimums oft sehr leicht durch einen geeigneten Programmentwurf (und dazu passende Hardware) zu erreichen.

Interessant für die Forschung über parallele Algorithmen und deren praktische Umsetzung sind daher vor allem solche Probleme, die zwar parallelisierbar sind, aber aufgrund ihrer Struktur Kommunikation zwischen den Prozessoren bei der Lösung ihrer Teilprobleme erfordern. Dann ist es nämlich zum Erreichen möglichst guter Speedups nötig, diese Kommunikation zu optimieren. Die in der vorliegenden Arbeit betrachteten geometrischen Probleme sind dieser Kategorie zuzuordnen. Ein besonderes Augenmerk gilt daher Art und Optimierung der Kommunikation bei der Problemlösung. Auch wenn man weder optimale Speedups noch sehr gute Skalierbarkeit erwarten darf, sollen diese experimentellen Parameter natürlich so gut wie möglich sein.

Um Probleme in eine dieser drei Klassen einordnen und theoretische Voraussagen über die Güte von parallelen Algorithmen zu ihrer Lösung treffen zu können, benötigt man ein paralleles Rechenmodell, das möglichst sowohl den Entwurf als auch die Analyse paralleler Algorithmen einfach und realistisch macht. Im sequentiellen Fall hat sich als Modell für Entwurf und Analyse von Algorithmen die Registermaschine (engl.: *random access machine* (*RAM*)) durchgesetzt. Ihr Erfolg basiert auf der Tatsache, daß sie reale Rechnerarchitekturen der von-Neumann-Familie realistisch abbildet und dennoch einfach zu handhaben ist. Für parallele Algorithmen gestaltet sich die Suche nach einem solchen Modell schwieriger, weil es viele verschiedene Parallelrechnerarchitekturen gibt, die sich durch mehrere, meist gegenseitig abhängige, Parameter unterscheiden. Zu diesen Parametern gehören unter anderem die Netzwerktopologie und -latenz, die Art der Prozessorallokation, -synchronisation und -ablaufplanung sowie die Art, wie die Prozessoren untereinander Daten austauschen.

In diesem Unterkapitel werden daher drei bekannte parallele Modelle vorgestellt. Das PRAM-Modell existiert seit Ende der 1970er Jahre und ist eine konsequente Fortsetzung des RAM-Modells für p durch gemeinsamen Speicher miteinander verbundene Prozessoren. Es wird noch immer zur Beurteilung paralleler Algorithmen verwendet, da es auf recht einfache Weise allgemeine Aussagen über die Performanz paralleler Algorithmen erlaubt. Allerdings vernachlässigt das PRAM-Modell in seiner ursprünglichen Version vollständig die Kosten für Interprozessor-Kommunikation, wie sie in realen Parallelrechnerarchitekturen auftreten. Es hat sich daher gezeigt, daß man für den Entwurf paralleler Algorithmen auch auf andere Modelle zurückgreifen sollte, damit die oft nicht unerheblichen Kommunikationskosten nicht unberücksichtigt bleiben (s. etwa Hambrusch [42, S. 92]).

Das BSP-Modell versucht als eines von mehreren Modellen, diese Lücke zu schließen. Es hat seit seiner Einführung unter dem Namen BSP [68] (vormals XPRAM [67]) große Beachtung als

Alternative zur PRAM erfahren und sich etabliert. Der Entwurf portabler paralleler Algorithmen hat sich durch BSP vereinfacht, das Ziel, mit diesem Modell paralleles Rechnen so populär zu machen wie sequentielles, ist aber nicht erreicht worden.

Das CGM-Modell ist eine Variation des BSP-Modells, das auf aktuelle Parallelrechner zugeschnitten ist und einfache, aber gleichzeitig realitätsnahe Entwurfs- und Analysemöglichkeiten bietet.

Die folgenden Abschnitte erläutern die drei Modelle genauer und lassen auch erkennen, warum im nachfolgenden Teil dieser Arbeit hauptsächlich das realistische CGM-Modell zum Entwurf und der Analyse der verwendeten parallelen Algorithmen eingesetzt wird.

1.3.1 PRAM

Die grundlegende Architektur des PRAM-Modells besteht aus p gewöhnlichen seriell arbeitenden Prozessoren, die alle mit einem gemeinsamen globalen Speicher verbunden sind.

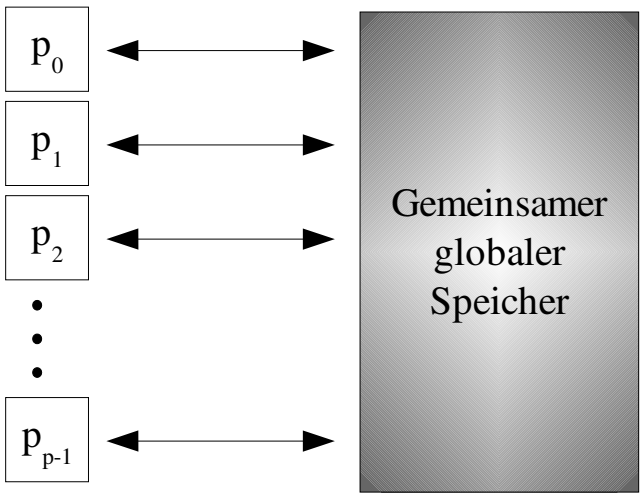

Abbildung 1.9: Schema einer PRAM

Während derselben Zeiteinheit können die vollkommen taktsynchron arbeitenden Prozessoren gleichzeitig und parallel auf diesen Speicher lesend oder schreibend zugreifen. Ob sogar dieselbe Speicherstelle von mehreren Prozessoren gleichzeitig gelesen bzw. beschrieben werden darf, hängt vom verwendeten PRAM-Modell ab. Man unterscheidet dabei folgende Typen:

	Lesen	Schreiben	Voller Name
EREW-PRAM	exklusiv	exklusiv	Exclusive-Read-Exclusive-Write-PRAM
CREW-PRAM	mehrfach	exklusiv	Concurrent-Read-Exclusive-Write-PRAM
CRCW-PRAM	mehrfach	mehrfach	Concurrent-Read-Concurrent-Write-PRAM

Tabelle 1.2: Drei grundlegende PRAM-Typen

Greifen bei einer CRCW-PRAM mehrere Prozessoren gleichzeitig schreibend auf dieselbe Speicherstelle zu, muß natürlich eine Vereinbarung getroffen werden, wie dieser Schreibkonflikt aufgelöst wird. Dazu gibt es verschiedene Methoden; bei JaJa [44, S. 15] finden sich - ne-

ben dem Hinweis, daß es noch mehr gibt - drei: die gemeinsame CRCW-PRAM ("common CRCW PRAM"), die gleichzeitige Schreibzugriffe auf dieselbe Speicherstelle nur erlaubt, wenn alle diese Zugriffe denselben Wert schreiben wollen. Bei der beliebigen CRCW-PRAM ("arbitrary CRCW-PRAM") erlaubt man, daß nach dem Rechenschritt der Wert eines beliebigen Prozessors an der Konfliktstelle gespeichert ist. Schließlich gibt es noch die Prioritäts-CRCW-PRAM ("priority CRCW PRAM"), bei der der Konflikt aufgelöst wird, indem der Prozessor mit der kleinsten Nummer die höchste Priorität hat und seinen Wert an die Konfliktstelle schreiben darf.

Man kann zeigen (s. Jaja [44, S. 495ff.]), daß sich die Modelle EREW, CREW und die oben genannten CRCW-Modelle zwar in ihrer Mächtigkeit unterscheiden (CRCW ist mächtiger als CREW, die wiederum mächtiger ist als EREW), dieser Unterschied aber nicht wesentlich ist. So kann das mächtigste Modell mit p Prozessoren, eine Prioritäts-CRCW-PRAM, auf dem schwächsten Modell mit p Prozessoren, einer EREW-PRAM, derart simuliert werden, daß sich nur eine zeitliche Verlangsamung der Ausführungszeit um den Faktor $O(log\, p)$ einstellt.

Als Leistungsmaße bei PRAM-Algorithmen werden die Anzahl der parallelen Zeittakte (engl.: *TIME*) und die Gesamtzahl der durchgeführten Operationen der PRAM (engl.: *WORK*) verwendet, nicht die Kommunikation. Durch den gemeinsamen Speicher können PRAM-Prozessoren Daten miteinander austauschen, ohne sich explizit Nachrichten senden zu müssen. Auf diese Weise wird der Entwurf von Algorithmen durchaus vereinfacht, allerdings ist die technische Realisierung eines globalen Speichers sehr schwierig. Daher erfolgt bei den meisten Parallelrechnerarchitekturen der Datenaustausch durch Nachrichtenkommunikation zwischen den lokalen Speichern der einzelnen Prozessoren. Diese Inkonsistenz zwischen theoretischem Modell und Architektur führt dazu, daß die Vorhersagen des Modells in puncto Laufzeit oft nicht mit den beobachteten Laufzeiten übereinstimmen. Um diese Inkonsistenz zu eliminieren, wurden daher verschiedene Variationen und Weiterentwicklungen der PRAM sowie neue parallele Modelle entworfen (ein Artikel von Hambrusch [42] bietet einen kurzen Überblick), darunter das im nächsten Abschnitt behandelte BSP-Modell.

1.3.2 Bulk-Synchronous-Processing

Das Modell des Bulk-Synchronous-Processing (BSP) von Leslie G. Valiant [68] versucht, im Kontext paralleler Computer eine Verbindung zwischen Hard- und Software herzustellen. Das Ziel dieses Ansatzes ist die Bildung einer Abstraktionsebene, die zu einer Vereinfachung beim Entwurf paralleler Algorithmen und bei der Implementierung paralleler Programme führt. Ähnlich wie beim Erfolg des seriellen von-Neumann-Modells könne diese Abstraktion von der physisch vorhandenen Maschine zu portablen Programmen für eine Vielzahl verschiedener paralleler Architekturen führen, so Valiants Argumentation [68, S. 104f.]. Um eine große Praxisnähe zu erreichen, werden die Kosten für Kommunikation zwischen Recheneinheiten in Abhängigkeit von bestimmten Leistungsparametern des Systems explizit berücksichtigt.

Das BSP-Modell wird folgendermaßen als Kombination von drei Attributen definiert (vgl. Valiant [68, S. 105]):

1. Eine Anzahl von Komponenten, die Berechnungen und/oder Speicherzugriffe durchführen.

2. Ein Router, der Nachrichten Punkt-zu-Punkt zwischen einem Komponentenpaar über ein Netzwerk versendet. Das Senden derselben Nachricht an mehrere Ziele gleichzeitig ist als Elementaroperation im Modell nicht vorgesehen.

3. Eine Vorrichtung, mit der alle oder auch nur ein Teil der Komponenten synchronisiert werden können.

Im Gegensatz zur PRAM haben die Prozessoren einer BSP-Maschine also keinen gemeinsamen Speicher, sondern kommunizieren miteinander durch das Versenden von Nachrichten über ein gegebenes Netzwerk und den zugehörigen Router. Beim Entwurf und bei der Analyse eines Programms innerhalb dieses Modells unterteilt man das Programm in einzelne Schritte, sogenannte "Supersteps". Zwischen diesen Supersteps werden einige oder im Normalfall sogar alle Prozessoren miteinander synchronisert [68, S. 105].

Einen Superstep selbst kann man folgendermaßen in drei Phasen einteilen: Zunächst führt jeder Prozessor Berechnungen durch, die auf lokal verfügbaren Daten operieren. In Phase zwei werden die Daten aufbereitet, die in der dritten und letzten Phase zwischen den Komponenten per Netzwerk und Router versendet werden [68, S. 105].

Die Aufteilung des Programms zunächst in Supersteps und weiter in Superstep-Phasen erzeugt eine offensichtliche Entkopplung von lokaler Berechnung und Kommunikation. Dadurch wird zunächst die Komplexität des Algorithmus und somit letztlich auch die Wahrscheinlichkeit von Programmierfehlern wie z. B. Verklemmungen reduziert. Diese Entkopplung läßt sich insbesondere bei der Analyse noch weiter steigern, indem man jede Phase zu einem (dann phasenlosen) Superstep macht.

Für die Analyse eines Algorithmus setzt man - gleichgültig ob mit oder ohne Phaseneinteilung der Supersteps - die folgenden Parameter ein [68, 36]:

- n als Größe der Eingabe
- p als Anzahl der Prozessoren im System
- L ist im Originalmodell der Parameter, der angibt, nach wie vielen Zeiteinheiten periodisch geprüft wird, ob die für die Synchronisierung vorgesehenen Prozessoren ihren Superstep bereits beendet haben. Eine Zeiteinheit entspricht hierbei der Zeitspanne zur Abarbeitung einer elementaren Rechenoperation. Im hier und von Gerbessiotis und Valiant in einem späteren Artikel [36] verwendeten Alternativmodell endet ein Superstep, wenn mindestens L Zeiteinheiten verstrichen sind und alle Prozessoren die für den Schritt vorgesehenen Arbeiten beendet haben. (Beide Modelle unterscheiden sich in ihren Laufzeiten nur durch konstante Faktoren.)
- g gibt das Verhältnis der Zeiten für die Bearbeitung eines Datums bei lokaler Berechnung bzw. bei Kommunikation über das Netzwerk an und ist somit ein Maß für die Granularität des Systems. Anders ausgedrückt gibt g die Anzahl von Berechnungsoperationen aller Prozessoren des Systems an, die in der Zeit durchgeführt werden können, die der Router benötigt, um ein Datenwort zu versenden.

- h gibt die Obergrenze für die Anzahl der Daten an, die ein Prozessor während des Algorithmus pro Kommunikationsrunde versendet bzw. empfängt. Man spricht auch von der Kommunikation in *h-Relationen*.

Die Laufzeit eines BSP-Programms läßt sich mit diesen Parametern ausdrücken und im O-Kalkül bestimmen, wobei man die Gesamtlaufzeit durch λ, die Anzahl der Supersteps, angibt. Wie lange die Ausführung eines einzelnen Supersteps benötigt, hängt von folgenden Faktoren ab:

- Kosten für die lokale Berechnung w_{max}: Diese werden üblicherweise in der O-Notation ausgedrückt und ergeben sich aus dem Maximum der Ausführungsschritte, die die Prozessoren während des Supersteps für die lokale Berechnung benötigen.
- Kosten für die Kommunikation: Man gibt die Kosten einer Kommunikationsrunde mit $g \cdot h$ Zeiteinheiten als obere Schranke an. Dafür multipliziert man die Anzahl der theoretisch möglichen lokalen Operationen pro versendetem Datenwort mit der tatsächlichen Anzahl der versendeten Datenwörter. So erhält man ein Maß für die Anzahl der Berechnungsoperationen, die einem durch die Netzwerkkommunikation theoretisch "entgangen" sind.
- Kosten für die Synchronisation, angegeben durch den Parameter L.

Für jeden Superstep sind demnach $max\{w_{max} + gh,\ L\}$ Zeiteinheiten nötig. Für die Analyse trennen wir im folgenden die Supersteps in solche, die lokal rechnen und dafür eine Laufzeit von w_{max} benötigen und solche, die kommunizieren und dafür im schlechtesten Fall $gh + L$ Schritte brauchen. Diese in der Literatur übliche Vereinfachung erhöht die Kosten maximal um den konstanten Faktor 3 (s. Juurlink et al. [46, S. 304]).

Ein weiterer sehr wichtiger Parameter des Modells ist der Effizienzquotient, im Original [68, S. 107] *slack* oder *slackness* genannt. Dieser gibt an, für welche Relation $\frac{n}{p}$ der Algorithmus asymptotisch effizient ist, sich also durch den Einsatz von p Prozessoren eine innerhalb des Modells und der O-Notation p-mal schnellere Abarbeitung des Problems ergibt.

Man strebt üblicherweise einen Effizienzquotienten von $\frac{n}{p} \geq 1 + \epsilon$ mit $\epsilon > 0$ an, weil dieser theoretisch optimal ist. Allerdings sind im Hinblick auf zur Zeit verfügbare Hardware und damit lösbare Problemgrößen durchaus Effizienzquotienten von $\frac{n}{p} \geq p^{2+\epsilon}$ für praktische Belange völlig ausreichend. Ein Beispiel: Es stehen $64 = 2^6$ Prozessoren zur Verfügung. Wir möchten auf diesen Prozessoren ein Problem mit der Eingabegröße 2^{18} lösen. Der Algorithmus muß dann für asymptotische Optimalität einen Effizienzquotienten von $\frac{2^{18}}{2^6} = 2^{12} \geq (2^6)^2 = p^2$ aufweisen.

Zusammenfassend kann man demnach sagen, daß ein guter BSP-Algorithmus sich durch eine geringe Anzahl von Supersteps, asymptotisch optimale lokale Berechnung und einen niedrigen Effizienzquotienten auszeichnet. Die Kosten für die Kommunikation und die Synchronisation hängen zwar im wesentlichen von der verwendeten Hardware ab, dürfen aber bei der Beurteilung nicht vernachlässigt werden. Eine der Stärken des BSP-Modells ist schließlich die Berücksichtigung des mitunter immensen Zeitaufwands für Nachrichtenkommunikation, die beim eher praxisfernen PRAM-Modell durch den gemeinsamen Speicher in dieser Form nicht entstehen.

Die detaillierte Analyse eines BSP-Algorithmus mit Hilfe dieser Parameter wird später in diesem Kapitel am Beispiel eines Sortieralgorithmus demonstriert. Vorher wird noch gezeigt, wie

Kommunikationsoperationen realisiert werden, die nicht zur Kategorie *Punkt-zu-Punkt* gehören und demnach nicht vom Router des Modells als Elementaroperation unterstützt werden.

Broadcast im BSP-Modell

Um zu verstehen, wie man vom Router des BSP-Modells nicht unterstützte Kommunikationsoperationen möglichst effizient ausführt, betrachten wir exemplarisch folgendes - "Broadcast" genanntes - Problem, das in drei Varianten von Hill et al. theoretisch und experimentell analysiert wird [43]: Ein Prozessor ("Quelle") soll eine in seinem Speicher lokal vorhandene Datenstruktur der Größe $n \geq p$ an alle p Prozessoren des BSP-Computers versenden. Zur Lösung dieses Problems bieten sich drei verschiedene algorithmische Grundansätze an, nämlich *1-Phasen-Broadcast*, *Baum-Broadcast* und *2-Phasen-Broadcast*.

Beim 1-Phasen-Broadcast versendet der Quellprozessor p-1 Nachrichten der Größe n an die restlichen Prozessoren. Dies ist in einem Superstep möglich und verursacht Kosten in Höhe von $O(png + L)$, da $O(p)$ Nachrichten der Größe n versendet werden. Selbstverständlich ist dies nicht optimal und skaliert aufgrund des linearen Faktors p sehr schlecht.

Eine bessere Skalierung bietet da schon Baum-Broadcast, welches das bereits von der PRAM bekannte Baumparadigma (s. JaJa [44, S. 43ff.]) zum Verteilen der Nachrichten verwendet. Dabei versendet der Quellprozessor als Wurzel eines binären Baumes die Nachricht der Größe n an einen anderen Prozessor P'. Der Quellprozessor und P' bilden nun Ebene 1 des Binärbaums und wiederholen als Wurzeln ihres Teilbaums rekursiv und parallel dieses Verfahren durch Versenden der Datenstruktur an neue Prozessoren. Dieses Verfahren wird solange durchgeführt, bis der Baum mit Tiefe $\lceil log_2 p \rceil$ abgearbeitet ist und alle vorgesehenen p Prozessoren das Datum erhalten haben (siehe Abb. 1.10). Die Komplexität der Kommunikation beträgt pro Baumebene bzw. Superstep $O(ng + L)$ und damit für den gesamten Algorithmus $O(log\, p\, (ng + L))$.

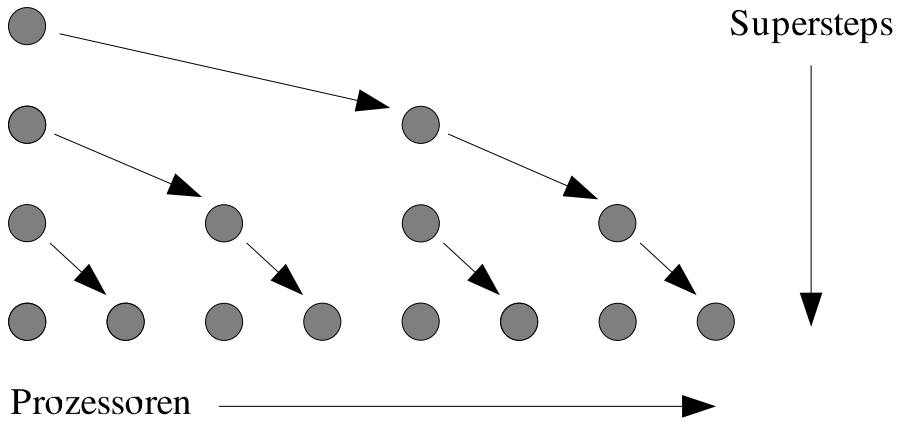

Abbildung 1.10: Schema eines Baum-Broadcasts

Der für die meisten Relationen von n und p beste Ansatz ist allerdings 2-Phasen-Broadcast. Dabei versendet der Quellprozessor in der ersten Phase p-1 Nachrichten der Größe $\lceil \frac{n}{p} \rceil$ an die übrigen Prozessoren, an jeden also einen anderen Teil der gesamten Datenstruktur. In Phase 2 verschickt dann jeder Prozessor seinen Teil der Datenstruktur an alle anderen Prozessoren, so

daß nach bereits zwei Supersteps der Algorithmus abgeschlossen ist. Es entstehen dabei folgende Kosten: $2 \cdot (p \cdot \frac{n}{p} \cdot g + L) = 2(ng + L) = O(ng + L)$.

Die folgende Tabelle soll die Laufzeiten gegenüberstellen, bevor sie für ein Fazit verglichen werden. Dabei wurde für die Bestimmung der Laufzeit zur Vereinfachung p-1 durch p ersetzt.

	Laufzeit	**Asymptotische Laufzeit**
1-Phasen-Broadcast	$png + L$	$O(png + L)$
Baum-Broadcast	$log\, p\,(ng + L)$	$O(log\, p\,(ng + L))$
2-Phasen-Broadcast	$2\,(ng + L)$	$O(ng + L)$

Tabelle 1.4: Übersicht der verschiedenen Broadcast-Laufzeiten

Wie Hill et al. [43] darlegen und durch experimentelle Ergebnisse untermauern, skaliert 2-Phasen-Broadcast mit steigender Prozessorzahl sehr gut und ist immer besser als Baum-Broadcast. Wegen der zweiten Phase ist es zudem nur dann schlechter als 1-Phasen-Broadcast, wenn L groß und n und p klein sind. Die Autoren weisen daher daraufhin, daß Baum-Broadcast trotz seiner großen Verbreitung in parallelen Bibliotheken immer die schlechteste Wahl ist [43, S. 42].

Ein BSP-Sortieralgorithmus und seine Analyse

Zur Veranschaulichung des Modells und den Analysemöglichkeiten, die es bietet, wird nun ein bekannter Sortieralgorithmus dargestellt. Es handelt sich dabei um "Parallel Sorting by Regular Sampling" und geht auf Shi und Schaeffer zurück [63]:

Algorithmus 1.32 *Parallel Sorting by Regualar Sampling (auch Sample-Sort oder PSRS genannt)*

Eingabe: BSP-Rechner mit p Prozessoren; jeder Prozessor $0 \leq i \leq p-1$ speichert lokal eine Menge S_i von $\frac{n}{p}$ Ganzzahlen aus der Eingabemenge $S := \bigcup_{0 \leq i \leq p-1} S_i$.

Ausgabe: Die berechnete Permutation von S ist global aufsteigend sortiert, d. h. für alle $0 \leq i \leq p-1$ gilt, daß $S_i := b(i)$ lokal aufsteigend sortiert ist, $S = \bigcup_{0 \leq i \leq p-1} S_i$ und für alle $0 \leq i \leq p-2$ gilt: $max\{S_i\} < min\{S_{i+1}\}$.

1. Jeder Prozessor sortiert lokal S_i mit einem schnellen sequentiellen Sortierverfahren.
2. Lokal nimmt sich jeder Prozessor aus seinem Feld p Elemente mit gleichem Abstand, also jedes $(\frac{n}{p^2})$-te Element. Diese Elemente werden lokale Teiler genannt.
3. Alle Prozessoren senden ihre lokalen Teiler an Prozessor 0.
4. Prozessor 0 sortiert alle p^2 lokalen Teiler mit einem schnellen sequentiellen Sortierverfahren und nimmt sich aus der sortierten Folge jedes p-te Element. Die Elemente dieses neuen Feldes G heißen globale Teiler.
5. Prozessor 0 versendet G, die p-1 globalen Teiler, an alle Prozessoren.

6. Sei B(i,j) für Prozessor i der j-te Abschnitt seiner lokalen Daten hinsichtlich der globalen Teiler, also die lokalen Daten, die wertmäßig zwischen G_{j-1} und G_j liegen. Jeder Prozessor i berechnet nun seine p Abschnitte B(i,j) mit $0 \leq j \leq p-1$ und sendet jeden Abschnitt B(i,j) an Prozessor j.

7. Sei b(j) die Vereinigung von B(0,j), B(1,j), ..., B(p-1,j), also ist b(j) die Menge an Daten, die Prozessor j in Schritt 6 empfangen hat. Jeder Prozessor j sortiert lokal b(j) mit einem schnellen sequentiellen Sortierverfahren. Alternativ können die B(i, j) $(0 \leq i \leq p-1)$ auch auf geeignete Art und Weise gemischt werden, da es sich bei ihnen um aufsteigend sortierte Folgen handelt.

8. Ausgabe von $S_i := b(j)$.

Vor der Analyse wird gezeigt, daß die in Schritt 6 berechneten Abschnitte nicht "zu groß" werden.

Lemma 1.33 *Nach der Ausführung des Algorithmus PSRS hat die lokal gespeicherte Menge b(j) eine Größe von höchstens* $\frac{2n}{p}$.

Beweis durch Widerspruch: Für alle Elemente e in b(j) gilt: $G_{j-1} \leq e \leq G_j$. Anders ausgedrückt wird jeder Abschnitt b(j) durch die Arbeitsweise des Algorithmus von zwei globalen Teilern begrenzt (an den Rändern des durch S definierten Intervalls kann man sich o. B. d. A. noch $-\infty$ und $+\infty$ als Teiler vorstellen).

Angenommen die Behauptung sei falsch, d. h. es existiert ein j $(0 \leq j \leq p-1)$, so daß $|b(j)| > \frac{2n}{p}$ gilt. Wie viele lokale Teiler kann man dann in diesem b(j) finden? Nach Konstruktion haben wir jedes $\frac{n}{p^2}$-te Element des Eingabefeldes jedes Prozessors zu einem lokalen Teiler gemacht. Daher gilt immer, daß sich in $\frac{2n}{p^2}$ aufeinanderfolgenden Elementen jedes Eingabefeldes mindestens ein lokaler Teiler befindet. Die Anzahl der lokalen Teiler in b(j) wiederum kann man abschätzen, indem man die angenommene Größe von b(j) durch die Größe der Bereiche teilt, in denen vor der Sortierung im letzten Schritt auf jeden Fall mindestens ein lokaler Teiler zu finden ist:

$$Zahl\, lokaler\, Teiler\, in\, b(j) \geq \frac{|b(j)|}{\frac{2n}{p^2}} = \frac{|b(j)|p^2}{2n} > \frac{2np^2}{2np} = p$$

Nimmt man also an, daß b(j) mehr als $\frac{2n}{p}$ Elemente enthält, so müssen darin mehr als p lokale Teiler liegen. Dies bedeutet aber gleichzeitig, daß zwischen zwei globalen Teilern mehr als p lokale Teiler liegen müssen. Da in Schritt 2 des Algorithmus jeder p-te lokale Teiler zu einem globalen Teiler wird, liegen zwischen zwei globalen Teilern allerdings höchstens p-1 lokale Teiler.

Dies führt zum Widerspruch, so daß die Behauptung wahr ist. □

Analyse des Algorithmus PSRS

Da jeder Prozessor seine lokalen Daten hinsichtlich derselben globalen Teiler in p Abschnitte einteilt, die in sich selbst und untereinander aufsteigend sortiert sind, wird erreicht, daß nach Schritt 6 jeder Prozessor ein bezüglich anderer Prozessoren disjunktes Intervall von S speichert und für alle $0 \leq j \leq p-2$ gilt: $max\{b(j)\} < min\{b(j+1)\}$. Werden die b(j) in Schritt 7 noch lokal sortiert, so gilt am Ende des Algorithmus zusätzlich, daß die S_i mit $0 \leq i \leq p-1$ lokal sortiert sind.

Damit arbeitet der Algorithmus PSRS korrekt und berechnet eine verteilt gespeicherte, sortierte Permutation der Eingabe S.

Die Analyse der Zeitkomplexität der einzelnen Schritte ergibt:

1. Lokales Sortieren benötigt (z. B. mit Mergesort) $O(\frac{n}{p} log(\frac{n}{p}))$ Zeit für die lokale Berechnung.
2. Das Auswählen von p Elementen hat eine Zeitkomplexität von $O(p)$.
3. In diesem Schritt findet keine lokale Berechnung statt. Die Komplexität der Kommunikation beträgt $O(gp^2 + L)$, weil jeder Prozessor p Teiler an Prozessor 0 sendet, also insgesamt p^2 Stück.
4. Das Sortieren (z. B. mit Mergesort) der lokalen Teiler kann in der Zeit $O(p^2 log\, p)$ erfolgen, das Auswählen der globalen Teiler benötigt $O(p)$ Zeit. Beide Werte beziehen sich auf lokale Berechnung, es erfolgt keine Kommunikation.
5. Hier wird nur kommuniziert, nicht lokal gerechnet. Es handelt sich dabei um einen Broadcast von p Werten. Mit 2-Phasen-Broadcast betragen die Kosten dafür $O(pg + L)$, genauer $2pg + 2L$.
6. Das Berechnen der einzelnen Abschnitte kann analog zum Mischen zweier sortierter Folgen geschehen und benötigt dann $O(\frac{n}{p} + p)$ lokale Berechnungszeit. Das Kommunizieren der Abschnitte erfordert Kommunikationskosten in Höhe von $O(\frac{n}{p} \cdot g + L)$, weil jeder Prozessor höchstens $\frac{n}{p}$ Zahlen versendet und nach Lemma 1.33 höchstens $\frac{2n}{p}$ Zahlen empfängt.
7. Sortiert man die erhaltenen Abschnitte, so benötigt dies $O(\frac{n}{p} log(\frac{n}{p}))$ lokale Berechnungszeit. Dies ist nicht schlechter als die benötigte Zeit in Schritt 1.
8. Die Ausgabe ist in Linearzeit möglich, also in $O(\frac{n}{p})$.

Offensichtlich ist die Anzahl der Supersteps konstant, also gilt $\lambda = O(1)$. Um insgesamt in allen Schritten mit lokaler Berechnung das theoretische Optimum von $O(\frac{n}{p} log(\frac{n}{p}))$ Zeit pro Superstep zu erreichen, muß wegen Schritt 4 gelten, daß $\frac{n}{p} \geq p^2$ ist. Aus Lemma 1.33 folgt außerdem $h = \frac{2n}{p}$ und somit $O(g \cdot \frac{n}{p} + L)$ als obere Schranke für die Kommunikationskomplexität.

Satz 1.34 *Der Algorithmus PSRS sortiert auf einem BSP-Computer n gleichmäßig auf p Prozessoren verteilt gespeicherte und paarweise verschiedene Ganzzahlen bezüglich lokaler Berechnung in asymptotisch optimaler Zeit von $O(\frac{n}{p} log(\frac{n}{p}))$, wenn ein Effizienzquotient von $\frac{n}{p} \geq p^2$ gilt. Hat der BSP-Computer eine Bandbreite $g \leq log(\frac{n}{p})$ und eine Latenz $L \leq \frac{n}{p} \cdot log(\frac{n}{p})$, ist auch die Kommunikation und damit der gesamte Algorithmus in asymptotisch optimaler Zeit ausführbar.*

Beweis: Der Algorithmus arbeitet gemäß seiner Analyse korrekt für eine Menge von Ganzzahlen S. Setzt man für die Parameter g und L $log(\frac{n}{p})$ bzw. $\frac{n}{p} \cdot log(\frac{n}{p})$ und wegen des Effizienzquotienten für p^2 $\frac{n}{p}$ oder einen kleineren Wert ein, so ist jeder Schritt in der angegebenen theoretisch optimalen Zeit durchführbar. □

1.3.3 Erweiterungen des BSP-Modells

Wegen seiner Praxistauglichkeit konnte sich das BSP-Modell etablieren, wegen einiger Nachteile sind allerdings mehrere Varianten entwickelt worden, die diese Nachteile beseitigen sollen. Darunter ist das Modell des grobkörnigen Parallelrechners (engl.: *coarse grained multicomputer*, abgekürzt CGM), das sich besonders durch eine einfache Analyse auszeichnet. Es besteht aus p Prozessoren, die über ein beliebiges Netzwerk miteinander verbunden sind und jeweils einen lokalen Speicher von $O(\frac{n}{p}) \gg O(1)$ haben. Hier besteht ein wesentlicher Unterschied zur feinkörnigen Sichtweise der PRAM, bei der $\frac{n}{p} = O(1)$ angenommen wird. Es gibt im CGM-Modell keinen gemeinsamen Speicher, der Datenaustausch zwischen den Prozessoren erfolgt in expliziten Kommunikationsschritten. Die Speicherbegrenzung impliziert natürlich auch, daß pro Superstep von einem Prozessor maximal $O(\frac{n}{p})$ Daten gesendet und/oder empfangen werden können. Zur Leistungsbewertung von CGM-Algorithmen werden sowohl die Komplexität der lokalen Berechnungen als auch die Anzahl der Kommunikationsschritte herangezogen.

Kommunikationsaufrufe werden in diesem Modell als Variationen einer globalen Sortieroperation der $O(n)$ Eingabedaten angesehen. Der Vorteil dieses Vorgehens ist die leichte Abschätzung der Kommunikationskosten auf verschiedenen Netzwerktopologien, weil parallele Sortierprobleme für die gebräuchlichen Topologien sehr gut erforscht sind[14]. Dehne et al. [26] zeigen, daß die Kommunikationsoperationen *Segmented Broadcast*, *Multinode Broadcast*, *Total Exchange* und *Partial Sum* für den Fall $\frac{n}{p} \geq p$ eine Zeitkomplexität von $O(\frac{n}{p} + T_s(n,p))$ haben, wobei $T_s(n,p)$ die Zeit für eine globale Sortieroperation angibt. Weil $\frac{n}{p}$ kleiner als das theoretische Optimum für das Sortieren ist, können diese Operationen in der asymptotischen Zeitanalyse durch $T_s(n,p)$ ersetzt werden. Allerdings wird diese Art der Notation in neueren Arbeiten [49][28, S. 329] nicht mehr verwendet, weil mit dem grobkörnigen Sortieralgorithmus von Goodrich [37] ein Verfahren zur Verfügung steht, das für gängige Effizienzquotienten eine Sortierung von n Datensätzen in optimaler Zeit und konstant vielen Kommunikationsrunden durchführt. Daher wird auch hier nur noch die Anzahl der Kommunikationsrunden gezählt und die dabei versendeten und empfangenen Daten in ihrer Anzahl nach oben beschränkt.

Beim Algorithmenentwurf versucht man, sowohl die Anzahl der Kommunikationsschritte als auch den Effizienzquotionten möglichst klein zu halten, um viele Kommunikationsoperationen bestehend aus kleinen Nachrichten durch wenige solcher Operationen mit langen Nachrichten zu ersetzen.

Zum besseren Verständnis wird exemplarisch ein CGM-Algorithmus von Dehne et al. [25] zur parallelen Bestimmung der konvexen Hülle erläutert. Er nutzt bekannte Grundlagen aus dem seriellen und parallelen Umfeld, etwa die Bestimmung der Tangenten zwischen zwei Ketten durch binäre Suche (vgl. dazu die PRAM-Algorithmen in JaJas Lehrbuch [44, S. 56ff. und S. 264ff.]). Zwar ist dieser Algorithmus hinsichtlich seiner Anzahl von Kommunikationsrunden und seines Effizienzquotienten bereits verbessert worden [29], er besticht als Beispiel aber durch seine Kürze und Klarheit. Wie bei Algorithmen zur Lösung dieses Problems nicht unüblich, wird nur die Berechnung der oberen konvexen Hülle gezeigt, für die untere läuft das Verfahren analog.

[14] So gilt etwa für ein 2D-Gitter (mesh) eine Sortierlaufzeit von $\Theta(\frac{n}{p}(log\, n + \sqrt{p}))$ und für den Hypercube für praktische Belange $O(\frac{n}{p}(log\, n + log^2 p))$ (s. Dehne et al. [26, S. 383]).

Algorithmus 1.35 *Obere konvexe Hülle [25, S. 305f.]*

Eingabe: $CGM(n,p)$, jeder Prozessor speichert lokal beliebige $O(\frac{n}{p})$ Punkte der Punktmenge S.

Ausgabe: Eine verteilt gespeicherte Repräsentation der oberen konvexen Hülle von S.

1. Sortiere die Eingabepunkte global gemäß ihrer x-Koordinate, was zu Mengen V_i auf Prozessor p_i führt.
2. Berechne lokal die obere Kette von V_i und speichere die Kanten in einem Array C_i in sortierter Form.
3. Führe einen *Multinode Broadcast* durch, bei dem Prozessor p_i folgende dreiteilige Nachricht an die anderen Prozessoren sendet: $c_i := C_i[\frac{|C_i|}{2}]$, den Vorgänger von c_i in C_i und analog seinen Nachfolger.
4. Berechne folgendermaßen alle paarweisen p^2 Tangenten: Jeder Prozessor p_i speichert p Geraden l_{ij}, die am Anfang alle durch c_i und c_j verlaufen (c_j ist der mittlere Punkt von Prozessor p_j). Die beiden nächsten Schritte werden insgesamt $2\,log\,n$ mal durchgeführt: Jeder Prozessor führt für die p Geraden l_{ij} einen Schritt der binären Suche auf seiner oberen Kette C_i durch. Bezeichne c_{ij} den Punkt, der durch diesen Schritt berechnet wurde und den die Gerade l_{ij} als nächsten trifft. Führe nun die Kommunikationsoperation *Total Exchange* durch, bei der Prozessor p_i p verschiedene Tripel an Prozessor p_j ($1 \leq j \leq p$) versendet. Das Tripel j besteht dabei aus c_{ij}, seinem Vorgänger und seinem Nachfolger in C_i.
5. Berechne lokal für jede Menge V_i ihren Beitrag zur oberen konvexen Hülle von S. Dabei handelt es sich um die (möglicherweise auch leere) Folge von Punkten auf der oberen Kette zwischen dem am weitesten rechts liegenden Tangentialpunkt eines Endpunktes einer rechten Tangente und dem am weitesten links liegenden Tangentialpunkt eines Endpunktes einer linken Tangente. Fallen diese beiden Punkte zusammen, muß nur geprüft werden, ob die Tangenten unter dem Schnittpunkt einen Winkel größer als π bilden. Falls ja, ist der Beitrag zur gesamten oberen Hülle dieses Prozessors leer.

Die Korrektheit ist bei Kenntnis der Grundlagen leicht einzusehen. Die Analyse der Zeitkomplexität gliedert sich so: Schritt 1 benötigt eine Zeit von $T_s(n,p)$ und ist abhängig von der verwendeten Topologie. Die Berechnung der lokalen konvexen Hülle kann wegen vorliegender Sortierung mit *Grahams Scan* in $O(\frac{n}{p})$ Schritten durchgeführt werden. Schritt 3 ist eine der bekannten Kommunikationsoperationen, die asymptotisch durch $O(T_s(n,p))$ abgeschätzt werden kann. Die Bestimmung der Tangenten erfordert im wesentlichen $2\,log\,n$ Kommunikationsaufrufe der Art *Total Exchange*, dessen Zeitbedarf mit $T_x(p)$ für das Versenden von p Daten bezeichnet wird. Der letzte Schritt ist lokal in der Zeit $O(p)$ zu bewältigen. Der Sortierschritt dominiert die Kommunikationszeit, falls $\frac{n}{p} \geq pT_s(p,p)$ gilt. Somit erhält man folgendes Ergebnis:

Satz 1.36 *(Dehne et al. [25, S. 306]) Das Problem 2D-Konvexe-Hülle kann für eine Menge von n Punkten auf einem grobkörnigen Multicomputer CGM(n,p) mit $\frac{n}{p} \geq pT_s(p,p)$ in der Zeit $O(\frac{n\,log\,n}{p} + T_s(n,p))$ gelöst werden.*

Es gibt noch einige andere parallele Modelle, die eine gewisse Relevanz in der Forschung haben. Dazu gehören das logP-Modell, das mittlerweile in mehrfacher Hinsicht als nachteilig im Vergleich zu BSP angesehen wird (siehe den Übersichtsartikel von Hambrusch [42, S. 93]), sowie Variationen des BSP-Modells, darunter BSP* von Meyer auf der Heide et al. [10], das die vergleichsweise hohen Latenzkosten bei kleinen Nachrichten im BSP-Modell berücksichtigt. Man muß allerdings sagen, daß ihre heutige und/oder zukünftige Bedeutung als nicht sehr groß einzuschätzen ist, weshalb auf eine weitere detaillierte Betrachtung in dieser Arbeit verzichtet wird.

1.4 Fazit

Dieses Kapitel hat in die Aufgabenstellung eingeführt und die Grundlagen der Algorithmischen Geometrie sowie der Parallelverarbeitung erläutert. Es wurde festgestellt, daß normale Delaunay-Triangulationen und solche höherer Ordnung dazu verwendet werden können, realistische Oberflächenmodelle wie TINs und Meshes zu erstellen. Da diese Datenstrukturen meist zur Lösung komplexer Probleme verwendet werden und daher im Umfeld der Parallelverarbeitung zum Einsatz kommen, sollten auch Delaunay-Triangulationen höherer Ordnung parallel berechnet werden können.

Dies soll innerhalb eines theoretischen Modells geschehen, das hinsichtlich heutiger Parallelrechnerarchitekturen praktische Relevanz hat, weshalb sich die Modelle CGM und BSP anbieten. Das CGM-Modell besticht vor allem durch seine Nähe zum von-Neumann-Modell und die damit einhergehende Einfachheit bei kaum geminderter Aussagekraft; es bildet daher die Grundlage für weitere Überlegungen in dieser Arbeit. Zur Identifizierung geeigneter Verfahren für die parallele Berechnung höherer Ordnungen werden im folgenden Kapitel bekannte serielle und parallele Algorithmen zur Bestimmung der normalen Delaunay-Triangulation vorgestellt.

Kapitel 2

Forschungsarbeiten zur Delaunay-Triangulation

Im vorigen Kapitel sind bereits einige grundlegende Eigenschaften von Delaunay-Triangulationen dargelegt worden. Dieses Kapitel setzt diese Einführung ohne Anspruch auf Vollständigkeit fort und beschreibt weitere wichtige bekannte Merkmale dieser Dreieckszerlegung. Aus Platzgründen lediglich skizzierte Beweise können in den zitierten Originalarbeiten nachgelesen werden.

Die gesammelten Kenntnisse werden im weiteren Verlauf des Kapitels zu Algorithmen führen, die das geometrische Problem der Berechnung der Delaunay-Triangulation für eine Punktmenge in der euklidischen Ebene lösen. Zunächst werden die wichtigsten seriellen Algorithmen vorgestellt, gefolgt von verschiedenen parallelen. Interessant dabei ist, daß sowohl serielle als auch parallele Verfahren teilweise sehr unterschiedliche Herangehensweisen zur Lösung desselben Problems wählen.

2.1 Eigenschaften und Anwendungen

Mit Hilfe der Eulerschen Polyederformel kann man sowohl die Raumkomplexität einer Delaunay-Triangulation bestimmen als auch mittelbar eine Aussage über die Zeitkomplexität ihrer Berechnung treffen.

Satz 2.1 *(Eulersche Polyederformel, s. auch Jungnickel [45, S. 41]) Sei G ein planarer, zusammenhängender und in die Ebene eingebetteter Graph mit v Knoten, e Kanten und f Flächen. Dann gilt:* $v + f = e + 2$.

Beweis durch Induktion über die Anzahl der Kanten e:

1. Induktionsanfang: Hat G keine Kanten, exisitiert nur die unbeschränkte Fläche außerhalb von G, so daß die Behauptung wegen $v = 1$, $f = 1$ und $e = 0$ gilt.
2. Induktionsvoraussetzung: Die Behauptung gelte für alle $e_o < e$.
3. Induktionsschritt: G habe e Kanten. Enthält G einen Kreis, entfernt man eine Kante dieses Kreises. Nach Induktionsvoraussetzung gilt nun die Behauptung. Fügt man die entfernte

Kante wieder hinzu, entsteht gleichzeitig auch eine neue Fläche, so daß die Behauptung wieder gilt. Hat G keinen Kreis, ist G ein Baum mit $e = v - 1$ und $f = 1$ (elementares Ergebnis der Graphentheorie). Damit gilt auch in diesem Fall $v + f = e + 2$. □

Folgerung 2.2 *Eine Triangulation in der Ebene mit v Knoten hat höchstens 3v-6 Kanten und höchstens 2v-4 Dreiecke.*

Beweis: Eine Kante gehört zu genau zwei Dreiecken (im Falle von Kanten der konvexen Hülle ist eines dieser Dreiecke das unbeschränkte entartete Dreieck außerhalb der konvexen Hülle) und zu jedem Dreieck gehören mindestens drei Kanten, also $e \geq \frac{3f}{2}$. Zusammen mit Satz 2.1 ergibt sich dann:

$$\frac{3f}{2} \leq v + f - 2 \Rightarrow \frac{f}{2} \leq v - 2 \Rightarrow f \leq 2v - 4 \Rightarrow e \leq v + 2v - 4 - 2 \Rightarrow e \leq 3v - 6$$

□

Satz 2.3 *Sei $S \subset \mathbb{R}^2$ eine n-elementige Punktmenge. Dann gilt: Die Berechnung von DT(S) benötigt $\Omega(n \log n)$ Schritte.*

Beweis: Preparata und Shamos [59, S. 212] zeigen durch Reduktion auf das Sortierproblem, daß die Berechnung von VD(S) eine Komplexität von $\Omega(n \log n)$ hat. Es bleibt daher nur noch als Reduktion zu zeigen, daß man in Linearzeit die Lösung von DT(S) in eine Lösung für VD(S) umrechnen kann, um die Behauptung zu beweisen.

DT(S) hat $O(n)$ Dreiecke, für die man in jeweils konstanter Zeit den Umkreismittelpunkt (Voronoi-Knoten) bestimmen kann. Die Adjazenzinformationen der Voronoi-Knoten, also die Voronoi-Kanten, ergeben sich wegen der Dualität direkt aus den Nachbarschaftsbeziehungen von Dreiecken der vorhandenen Delaunay-Triangulation. Die Datenstruktur muß pro Dreieck also nur durch den zugehörigen Voronoi-Knoten und dessen Kanten zu Voronoi-Knoten benachbarter Dreiecke ergänzt werden. Da dies jeweils $O(n)$ Stück sind, kostet die Umwandlung der Lösung $O(n)$ Schritte und somit gilt die Behauptung. □

Nun wissen wir, daß man für die Berechnung der Delaunay-Triangulation in der Ebene $O(n)$ Raum und $\Omega(n \log n)$ Zeit benötigt. Dieses Ergebnis werden wir benutzen, um die Qualität von Algorithmen hinsichtlich ihrer Laufzeit zu beurteilen.

Neben der Dualität der zweidimensionalen Delaunay-Triangulation mit dem zweidimensionalen Voronoi-Diagramm besteht eine solche enge Verbindung auch mit der dreidimensionalen konvexen Hülle. Dies kann man durch geometrische Transformation herbeiführen.

Definition 2.4 *Sei $S' \subset \mathbb{R}^3$ und sei CH(S') die konvexe Hülle von S'. Dann ist die untere konvexe Hülle von S' - bezeichnet als LowerCH(S') - definiert als der Teil von CH(S'), der vom Punkt (0, 0,$-\infty$) aus sichtbar ist.*

Satz 2.5 *(vgl. Aurenhammer und Klein [7, S. 221]) Sei $S \subset \mathbb{R}^2$ und sei $S' := \{(x, y, x^2+y^2) \mid (x, y) \in S\}$ die durch Transformation erhaltene Punktmenge in $\mathbb{R}^3$. Dann ist die Delaunay-Triangulation von S die Projektion von LowerCH(S') auf die x-y-Ebene.*

Beweisidee: Seien p, q und r drei Punkte aus S. Transformiert man sie zu p', q' und r' in den Raum, so kann man zeigen, daß der Kreis $C(p, q, r)$ genau dann keine weiteren Punkte aus S enthält, wenn unterhalb der durch p', q' und r' aufgespannten Ebene keine weiteren Punkte aus S' liegen. Daraus folgt, daß drei Punkte aus S genau dann ein Delaunay-Dreieck bilden, wenn die korrespondierenden Punkte aus S' eine Fläche der unteren konvexen Hülle von S' induzieren. □

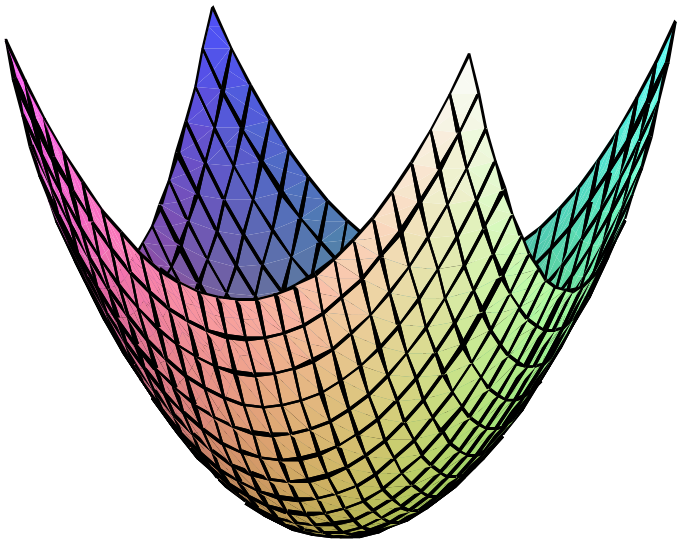

Abbildung 2.1: Paraboloid der Form $z = x^2 + y^2$

Eine interessante Nebenbemerkung zu Satz 2.5 ist, daß die Punkte von S' alle auf einem Paraboloiden mit Pol im Koordinatenursprung liegen, der durch Rotation um die z-Achse entsteht (siehe Abbildung 2.1). Führt man eine verschobene Transformation $S' := \{(x-a,\, y-b,\, (x-a)^2 + (y-b)^2 | \, (x, y) \in S\}$ durch, ist der Punkt (a, b) der neue Pol dieses Paraboloiden, da sich nur für ihn der transformierte Wert $(0, 0, 0)$ bildet.

Neben der bereits genannten technischen Simulation mit Gitternetzen bietet die Delaunay-Triangulation weitere Anwendungsmöglichkeiten. Man verwendet sie für TINs (die Delaunay-trianguliert sein können, aber nicht müssen) zur Interpolation von Höhenwerten der Punkten, die nicht im TIN gespeichert sind. Da nämlich die Triangulation einen gewissen Zusammenhang der Punkte vorgibt, kann man sich dies zunutze machen, um einen realistischen Näherungshöhenwert für die unbekannte Position zu bestimmen. Ähnliches gilt für die Verknüpfung zweier TINs, dabei sind nach van Kreveld [48] unter anderem Addition und Subtraktion, aber auch das Potenzieren eines einzelnen TINs sinnvolle und notwendige Operationen (z. B. zur Bestimmung von Höhendifferenzen oder von Erosionseinflüssen).

Das Finden kürzester Wege in einem Terrain ist eine Problemstellung von hohem praktischen Nutzen und wird unter anderem in Navigationssystemen verwendet. Lanthier et al. [50] geben ein paralleles Verfahren an, auf verschieden gewichteten (etwa mögliche Geschwindigkeit eines Objekts) Oberflächen den kürzesten Weg zu finden. Die Delaunay-Triangulation wird weiterhin bei einer Klasse von Cluster-Algorithmen mit Raumbezug als Preprocessing verwendet [31], um

davon ausgehend Clusterhierarchien aufzubauen. Alle diese Anwendungen dienen im geowissenschaftlichen Umfeld dazu, aus Daten Informationen und Wissen für den Benutzer zu erzeugen.

2.2 Serielle Algorithmen

Mittlerweile kennt man eine ganze Reihe von seriellen Algorithmen zur Bestimmung der planaren Delaunay-Triangulation. Eine Übersicht mit einem experimentellen Vergleich der verschiedenen Laufzeiten bietet der Artikel von Su und Drysdale [66]. Dort werden die wichtigsten damals bekannten Algorithmen untersucht und für den sequentiellen Fall bewertet. Aus dieser und anderen Quellen stammen die sechs Verfahren, die in diesem Unterkapitel besprochen werden.

Grundsätzlich kann man dabei zunächst deterministische von randomisierten Algorithmen unterscheiden. Das Verhalten eines deterministischen Algorithmus hängt ausschließlich von der Eingabe ab, wohingegen das Verhalten eines randomisierten Algorithmus auch auf zufällig gewählten Werten (etwa denen eines Zufallszahlengenerators) basiert. Der Terminologie von Mulmuley und Schwarzkopf [55, S. 633] folgend, liefert ein randomisierter Algorithmus zwar immer das korrekte Ergebnis, allerdings sind Zeit- und/oder Raumkomplexität nur mit einer gewissen Wahrscheinlichkeit garantiert[15].

Bei manchen Problemen bieten randomisierte Algorithmen den Vorteil, daß sie einfacher zu implementieren sind und/oder das Problem mit (sehr) hoher Wahrscheinlichkeit schneller lösen als deterministische. Kann man jedoch von einem deterministischen Algorithmus bei ähnlichem Implementierungsaufwand die gleiche Laufzeit im *worst case* erwarten wie von einem randomisierten, so ist der deterministische vorzuziehen, weil seine Güte immer garantiert ist. Letztlich ist jedoch immer eine Einzelfallbetrachtung des Problems und der zugehörigen Algorithmen notwendig.

Beim Speicherbedarf der vorgestellten Algorithmen ergeben sich keine grundlegenden Unterschiede, er beträgt grundsätzlich $O(n)$, weil zu jedem Delaunay-Dreieck nur konstant viele Werte gespeichert werden. Es wird daher auf eine Analyse des Speicherbedarfs zugunsten einer Laufzeitbetrachtung in theoretischer und praktischer Hinsicht verzichtet.

2.2.1 Flipping

Aus Satz 1.11 läßt sich ganz einfach folgender Algorithmus ableiten: Bestimme eine beliebige Triangulation T von $S \subset \mathbb{R}^2$. Solange es ein Viereck gibt, dessen Dreiecke lokal (also in diesem Viereck) nicht die Delaunay-Eigenschaft erfüllen, ersetzt man die bestehende Diagonale im Viereck durch die andere mögliche[16]. Gibt es kein solches Viereck mehr, hat man die größtmögliche Winkelfolge und damit die Delaunay-Triangulation erreicht.

Dieser Algorithmus hat eine Zeitkomplexität von $O(n \log n + f)$, wobei f die Anzahl der Kantenflips (Wechsel der Diagonalen) angibt. Da aber im schlechtesten Fall $\binom{n}{2} = \Omega(n^2)$ solcher Flips

[15] Im Unterschied dazu wird ein Algorithmus *probabilistisch* genannt, wenn er das korrekte Resultat nur zu einer gewissen Wahrscheinlichkeit berechnet, eine korrekte Ausgabe also nicht in jedem Fall garantiert ist [55, S. 633].

[16] Dies ist im planaren Fall in der Tat immer möglich, da die beiden Dreiecke ein konvexes Viereck bilden, wenn die lokale Delaunay-Eigenschaft der beiden Dreiecke nicht erfüllt ist.

vonnöten sind [33], ist dieser Algorithmus nicht immer optimal hinsichtlich seiner Laufzeit.

2.2.2 Inkrementelle Algorithmen

Inkrementelle Algorithmen fügen jeden der n Punkte aus der Eingabe S schrittweise in die Delaunay-Triangulation der bis dahin eingefügten Punkte ein und aktualisieren dann die Teillösung. Der bekannteste Vertreter dieser Klasse umschließt zur Vereinfachung die Eingabemenge S zu Beginn des Algorithmus mit einem ausreichend groß gewählten Dreieck, so daß jeder neu eingefügte Punkt $p \in S$ innerhalb der bestehenden Triangulation T' liegt. Deren Aktualisierung verläuft folgendermaßen: Zunächst berechnet man die Menge T_v der Dreiecke aus T', in deren Umkreis p liegt. T_v bildet ein von p ausgehend sternförmiges Polygon P, da für jeden Punkt $p' \in P$ das Segment $\overline{pp'}$ in P liegt (vgl. Fortune [33]). Sei nun T_p die Triangulation von P, deren Dreieckskanten alle von einem Eckpunkt von P zu p verlaufen; sie bildet eine korrekte lokale Delaunay-Triangulation von P (für den Beweis siehe de Berg et al. [13, S. 194]). Man ersetzt daher T_v durch T_p und erhält lokal und global eine aktuell gültige Delaunay-Triangulation.

Die Zeitkomplexität hängt wesentlich von der Anzahl der zu löschenden Dreiecke und somit der Einfügereihenfolge ab, weil dadurch die Größe von T_v beeinflußt wird. Im schlechtesten Fall entspricht T_v jedes Mal der aktuell berechneten Triangulation T', so daß man pro Einfügevorgang linearen Aufwand und damit eine nicht optimale Gesamtkomplexität der Laufzeit von $O(n^2)$ hat.

Fügt man allerdings die Punkte nacheinander in zufälliger Reihenfolge ein, erhält man einen randomisierten Algorithmus, der dem gleichen Schema folgt, aber eine bessere Laufzeit hat. Weil jede Eingabepermutation gleich wahrscheinlich ist, ergibt sich eine erwartete Zeitkomplexität von $O(n \log n)$. Man kann nämlich abschätzen, daß pro Einfügeschritt T_v eine erwartete Größe von $O(1)$ hat, so daß die Laufzeit jedes der $O(n)$ Schritte von der Punktlokalisierung in $O(\log n)$ Schritten zur Bestimmung von T_v dominiert wird [13, S. 197ff.].

Cignoni et al. [22] beschreiben einen inkrementellen Algorithmus, der bekannte Verfahren durch verschiedene Beschleunigungsmethoden verbessert. Der Algorithmus erstellt zunächst ein beliebiges Dreieck der zu berechnenden Delaunay-Triangulation. Ausgehend von dessen Kanten werden die Punkte gesucht, die mit diesen Kanten ein neues Delaunay-Dreieck bilden. Das Verfahren setzt man mit den noch nicht betrachteten Kanten der neuen Dreiecke fort. Die vollständige Delaunay-Triangulation ist erstellt, wenn man zu jeder Kante zwei Delaunay-Dreiecke gefunden hat (inklusive dem unendlichen degenerierten Dreieck außerhalb der konvexen Hülle der Eingabemenge). Durch *Gridding* und *Hashing* erreicht der Algorithmus gute Laufzeiten in der Praxis, obwohl er theoretisch nicht optimal ist (bei ungünstigen Verteilungen der Eingabe $S \in \mathbb{R}^2$ werden $O(n^2)$ Schritte benötigt). Es sei noch darauf hingewiesen, daß diese Art des Algorithmus manchmal auch als *gift wrapping* bezeichnet wird, weil das Verfahren methodische Ähnlichkeiten zum Algorithmus *Jarvis' March* zur Bestimmung der planaren konvexen Hülle aufweist, der entsprechend kategorisiert ist.

2.2.3 Gleitgeradenmethode

Der im Jahre 1987 von Steven Fortune veröffentlichte Algorithmus [32] zur Berechnung des Voronoi-Diagramms bzw. der Delaunay-Triangulation per Gleitgerade in der Ebene wird heu-

te allgemein als die eleganteste Lösung dieser beiden Probleme angesehen. Das von ihm benutzte Verfahren speichert und aktualisiert in zwei verschiedenen Kantenlisten den Fortschritt des Algorithmus, bei dem die Gleitgerade von unten nach oben über die Ebene geschoben und unter ihr die aktuell gültige Delaunay-Triangulation erstellt wird.

Die eine Kantenliste, die sogenannte Grenze, speichert eine geordnete Liste von Eingabepunkten. Ein Eintrag darin für den Punkt p entspricht einem Intervall I_p auf der Gleitgeraden, in dem der größte leere Kreis mit oberstem Punkt in I_p liegt, der den Eingabepunkt p schneidet (vgl. Fortune [34, S. 381]). Die andere Liste beinhaltet die noch zu bearbeitenden Ereignispunkte, an denen die Gleitgerade stoppen muß (daher Ereigniswarteschlange (engl.: *event queue*) genannt). Dabei unterscheidet man sogenannte *Kreis-* und *Site-Events*. Site-Events treten auf, wenn die Gleitgerade auf einen Punkt aus der Eingabe (engl.: *site*) trifft, Kreis-Events entstehen, wenn ein neues Delaunay-Dreieck gefunden worden ist. Letzteres passiert, wenn die Gleitgerade den obersten Punkt eines Umkreises von drei in der Grenze aufeinanderfolgenden Punkten erreicht.

Der Algorithmus hat eine optimale Laufzeit von $O(n \log n)$, da $O(n)$ Ereignispunkte auftreten und für jeden Ereignispunkt $O(\log n)$ Kosten entstehen. Diese ergeben sich durch die Aktualisierung der Prioritätswarteschlange zur Bestimmung des nächsten Ereignispunktes und durch die Aktualisierung der Grenze.

2.2.4 Teile und herrsche

Das Prinzip des "Teile und herrsche" (engl.: *divide and conquer*) besteht darin, durch Rekursion ein Problem wiederholt in kleinere Teilprobleme aufzuteilen, bis sich jedes Teilproblem aufgrund seiner geringen Größe sehr einfach lösen läßt. Die Hauptarbeit des Algorithmus wird dann üblicherweise beim Mischen der Teillösungen durchgeführt. Zur Berechnung der Delaunay-Triangulation mit diesem Konzept gibt es unter anderem den bekannten Algorithmus von Guibas und Stolfi [41] sowie dessen Variante mit einer besseren Laufzeit insbesondere für gleichverteilte Punktmengen von Dwyer [30].

Das Verfahren von Guibas und Stolfi teilt dabei die Ebene rekursiv durch senkrechte Schnitte in jeweils zwei neue annähernd gleich große Teilbereiche. Die Rekursion bricht ab, wenn ein Teilbereich höchstens drei Punkte enthält. Aus diesen wird dann eine Teillösung gebildet. Die Teillösungen werden gemischt, indem man die Kanten bestimmt, die zwischen den Teillösungen verlaufen (Querkanten), und jene löscht, die von den neu hinzugekommenen geschnitten werden. Dazu berechnet man zunächst die Querkante, die Teil der unteren konvexen Hülle der beiden Teillösungen ist. Von ihr ausgehend, wandert man im Grenzbereich der beiden Teillösungen nach oben und bestimmt durch fortlaufende Umkreistests wiederholt die nächste Querkante, bis man die Kante der oberen konvexen Hülle erreicht hat. Der Algorithmus hat eine für den schlechtesten Fall optimale Laufzeit von $O(n \log n)$, jedoch erreicht Dwyer durch einige Änderungen eine bessere erwartete Laufzeit von $O(n \log \log n)$ für viele Punktverteilungen [30, S. 146].

2.2.5 Dualität mit 3D-Konvexe-Hülle

Mit Kenntnis des Resultats von Satz 2.5, der Dualität der planaren Delaunay-Triangulation mit der dreidimensionalen konvexen Hülle, kann man die planare Punktmenge S wie im Satz be-

schrieben transformieren und dann von S' die dreidimensionale untere konvexe Hülle bestimmen. Deren Kantenbeziehungen zwischen den Punkten aus S' entsprechen exakt den Kantenbeziehungen der Delaunay-Triangulation zwischen den korrespondierenden Punkten aus S.

Als einfaches deterministisches Verfahren bietet sich für die Bestimmung der konvexen Hülle im Raum der Teile-und-herrsche-Algorithmus von Preparata und Hong [58] an. Dabei wird die Eingabemenge im Raum rekursiv in x-disjunkte Intervalle geteilt, bis diese Intervalle nur noch sehr wenige Eingabepunkte enthalten. Für diese bildet man per *brute force* die konvexe Hülle. Der entscheidende Schritt ist nun das Mischen der Teillösungen nach der Rückkehr vom rekursiven Aufruf. Hierzu wird eine zylinderförmige Triangulation T zwischen den Teillösungen erstellt, dann werden alle Flächen der alten Teillösungen entfernt, die von T verdeckt werden. Die Details des Algorithmus können bei Preparata und Shamos nachgelesen werden [59, S. 141ff.].

Seine Laufzeit von $O(n \log n)$ für eine Eingabegröße von n ist optimal im schlechtesten Fall. Es gibt mittlerweile auch ausgabeabhängige Algorithmen, die für eine Ausgabegröße M eine optimale Laufzeit von $O(n \log M)$ haben, siehe dazu den Artikel von Seidel [62, S. 372].

2.2.6 Zerlegungsstrategie

Die oben genannten Teile-und-herrsche-Algorithmen verrichten die Hauptarbeit nach den rekursiven Aufrufen, also in der Phase des Kombinierens der berechneten Teillösungen. Algorithmen, die den größten Teil ihrer Arbeit vor den rekursiven Aufrufen, also beim Zerlegen der Eingabe in Teilprobleme, tätigen, bezeichnen wir - der Terminologie von Joseph JaJa [44] folgend - als *Zerlegungsalgorithmen*[17].

Der erste Zerlegungs-Algorithmus zur Bestimmung der Delaunay-Triangulation in allen Dimensionen $d \geq 2$ stammt von Cignoni et al. [22]. Er teilt die Eingabe durch Hyperebenen wechselnder Dimension in zwei neue Teilprobleme. Vor dem rekursiven Aufruf für beide Teilprobleme werden allerdings alle Dreiecke (bzw. höherdimensionale Simplizes) inkrementell konstruiert, die von der teilenden Hyperebene geschnitten werden. Der dazu benutzte Algorithmus ist der in Abschnitt 2.2.2 beschriebene von Cignoni et al. [22]. Die Rekursion bricht ab, wenn es keine Kanten (bzw. Flächen der Dimension d-1) mehr in dem Teilbereich gibt, die nicht fertig bearbeitet sind. Wegen des verwendeten inkrementellen Algorithmus, dessen Laufzeit theoretisch nicht optimal ist, erfordert auch dieses Verfahren asymptotisch mehr Schritte als notwendig. Trotzdem erreicht man im Allgemeinen durch die eingesetzten Beschleunigungstechniken gute Laufzeiten in der Praxis.

Auf der Basis des Zerlegungsprinzips und mit Hilfe einer geometrischen Transformation haben Blelloch et al. [15] einen parallelen Algorithmus entworfen, der später in diesem Kapitel noch ausführlicher besprochen wird. Abweichend vom von Cignoni et al. [22] entwickelten Verfahren werden dabei vor dem Rekursionsaufruf keine Delaunay-Dreiecke berechnet, sondern lediglich Delaunay-Pfade (zusammenhängende Delaunay-Kanten von der unteren konvexen Hülle zur oberen), um die Aufteilung der Eingabe in zwei Teilprobleme vorzunehmen. Dieses Verfahren bricht ab, wenn die Teilbereiche nur noch aus solchen Pfaden bestehen und keine inneren Punkte

[17] In der Literatur wird diese Unterscheidung nicht grundsätzlich vorgenommen; Zerlegungsalgorithmen werden oft, wie auch bei Cignoni et al. [22, S. 129], der Kategorie "Teile und herrsche" zugerechnet. Eine Differenzierung ist hier aber gerade angesichts der späteren Parallelisierung sinnvoll.

mehr haben. Die zurückbleibenden Polygone können dann in linearer Zeit Delaunay-trianguliert werden, was zu einer Gesamtkomplexität von $O(n \log n)$ führt. Eine für diese Arbeit erstellte serielle Variante des Verfahrens wird ausführlicher in Kapitel 3.2 besprochen.

2.2.7 Fazit

Zusammenfassend läßt sich sagen, daß der Flipping-Algorithmus zwar sehr leicht zu implementieren ist, aber keine gute Laufzeit hat. Er eignet sich daher nur für lokale Verbesserungen geringer Größe. Die inkrementellen Algorithmen sowie der Zerlegungsalgorithmus von Cignonoi et al. sind zwar theoretisch nicht bzw. nur randomisiert optimal, können aber durch geeignete Beschleunigungstechniken eine gute Performanz erreichen. Für viele Verteilungen kann diese sogar besser als die Laufzeit der Gleitgeradenmethode sein, die vom theoretischen Standpunkt als beste und eleganteste gilt. Jedoch kann auch der Teile-und-herrsche-Algorithmus von Dwyer für viele Verteilungen eine bessere erwartete Laufzeit als die theoretische untere Schranke erreichen. Das Zerlegungsprinzip nach Blelloch et al. ist theoretisch optimal, benötigt jedoch für eine gute Performanz aufgrund der vielen Rekursionsschritte eine effiziente Implementierung. Wegen der Hin- und Rücktransformation können Algorithmen zur Bestimmung der konvexen Hülle im Raum im Allgemeinen nicht mit den schnellsten der vorgenannten Algorithmen mithalten.

Der experimentelle Vergleich von Su und Drysdale [66] identifiziert unter den getesteten Verfahren einen beschleunigten inkrementellen Algorithmus (nicht *gift wrapping*), Fortunes Gleitgeradenmethode und Dwyers Teile-und-herrsche-Algorithmus als die schnellsten. Ein Zerlegungsalgorithmus war nicht Bestandteil der Untersuchung.

2.3 Parallele Algorithmen

Die wichtigsten Teile eines parallelen Algorithmus zur Bestimmung der planaren Delaunay-Triangulation sind die Aufteilung der Eingabe auf die Prozessoren, die Bestimmung der Teillösung auf jedem Prozessor und das Kombinieren der Teillösungen zu einer verteilt gespeicherten Gesamtlösung des Problems. In diesem Kapitel wird daher erläutert, wie bekannte Verfahren diese Problematik lösen. Es werden dazu fünf Verfahren vorgestellt und hinsichtlich ihrer Implementierbarkeit sowie erwarteter paralleler Effizienz bewertet.

Zwei dieser Algorithmen sind randomisiert, weil sie das Konzept des *random sampling* verwenden. Dabei wird aus der Eingabe eine zufällige Teilmenge oder Probe (engl.: *random sample*) von geringer Größe ausgewählt. Das Problem (oder ein Hilfsproblem) wird dann für diese kleine Teilmenge gelöst, um anhand dieser Teillösung eine gute Aufteilung der gesamten Eingabe auf die Prozessoren zu erzielen. Die Komplexität der Algorithmen kann zwar asymptotisch abgeschätzt werden, wegen der Randomisierung gelten diese Schranken allerdings nur mit hoher Wahrscheinlichkeit. Folgende abkürzende Schreibweise wird dabei verwendet:

Bezeichnung 2.6 *(vgl. Dehne et al. [24, S. 548])* $X = \tilde{O}(f(n))$ *bezeichnet* $X = O(f(n))$ *mit hoher Wahrscheinlichkeit, d. h.* $X = \tilde{O}(f(n)) \Leftrightarrow (\forall c > c_0 > 1)\, Pr\{X \geq cf(n)\} \leq \frac{1}{n^{g(c)}}$*, wobei* c_o *eine feste Konstante und* $g(c)$ *ein Polynom in* c *mit* $g(c) \to \infty$ *für* $c \to \infty$ *ist.*

2.3.1 Berechnung der dreidimensionalen konvexen Hülle

Ein von Dehne et al. [24] für das CGM-Modell konzipierter Algorithmus berechnet die konvexe Hülle einer Punktmenge im $\mathbb{R}^3$. Er kann daher wegen der Dualität der Probleme auch zur Berechnung der Delaunay-Triangulation in der Ebene verwendet werden. Allerdings geschieht hier natürlich keine Aufteilung der Punkte in der Ebene, sondern im Raum, wodurch eine komplexe Aufteilungsmethode nötig wird.

Zu Beginn des Algorithmus enthält der lokale Speicher jedes Prozessors der CGM-Maschine $\frac{n}{p}$ Punkte der Eingabe S. Aus S wird dann von allen Prozessoren eine zufällige Teilmenge $R \subset S$ der Größe $O(\frac{n}{p})$ gebildet und auf jedem Prozessor gespeichert (*random sampling*). Jeder Prozessor berechnet dann mit einem schnellen sequentiellen Verfahren $CH(R)$ und entfernt die darin enthaltenen Punkte, weil sie nicht mehr als Teil der Gesamtlösung in Frage kommen. Die Flächen von $CH(R)$ werden dann auf eine geeignete Art und Weise partitioniert und auf die Prozessoren aufgeteilt, so daß Prozessor p_i die Teilmenge $R_i \subset CH(R)$ erhält. Sehr grob gesprochen wird dann auf jedem Prozessor eine Menge S_i gebildet, die jene Punkte enthält, die vom Inneren von $CH(R)$ gesehen nahe bei bzw. hinter den Flächen von R_i liegen. p_i bildet danach $CH(S_i)$ und entfernt wiederum die Punkte, die im Inneren von $CH(S_i)$ liegen.

Im nächsten Schritt werden die inneren Kanten entfernt, indem man zunächst zwei Kopien jeder Kante $\overline{uv}$ von $CH(S_i)$ erstellt (eine mit Schlüssel u, eine mit Schlüssel v) und diese Kantenkopien gemäß ihrer Schlüssel global sortiert. Man betrachtet nun jeden Endpunkt v und berechnet $CH(R_v)$, wobei R_v die Menge aller Strahlen ist, die durch die zu v inzidenten Kanten verlaufen und deren Ursprung in v liegt. Die zu v inzidenten Kanten innerhalb von $CH(R_v)$ werden entfernt. Gleiches gilt für v, falls v innerhalb von $CH(R_v)$ liegt (nämlich dann, wenn $CH(R_v) = \mathbb{R}^3$ gilt). Jede nicht gelöschte Kante e wird zu Prozessor p_i gesendet, wenn e ursprünglich Teil von $CH(S_i)$ war.

Der letzte wesentliche Schritt wählt die Kanten der globalen konvexen Hülle in jedem S_i aus. Dazu berechnet jeder Prozessor p_i für jedes Dreieck $t \in R_i$ den Punkt aus S_i, der den höchsten Abstand zu der Ebene durch t hat (und daher *weitester Punkt* genannt wird). Sei nun auf jedem Prozessor p_i G_i der Teilgraph von $CH(S_i)$, der von jenen Kanten induziert wird, die vor dem letzten Schritt übrig geblieben sind. Alle p_i berechnen die Menge E_i der Kanten von G_i, die von einem weitesten Punkt in S_i erreichbar sind. Die Vereinigung aller E_i ist die globale konvexe Hülle.

Die Details, insbesondere Korrektheitsbeweise und die Laufzeitanalyse, können in der Originalarbeit [24] nachgelesen werden. Dort kommt man zu dem Ergebnis, daß die dreidimensionale konvexe Hülle (und damit die zweidimensionale Delaunay-Triangulation) mit diesem Algorithmus auf einem grobkörnigen Parallelrechner $CGM(n,p)$ mit $\frac{n}{p} \geq p^{2+\epsilon}$, $\epsilon > 0$, mit einer Laufzeit von $\tilde{O}(\frac{n\, log_2\, n}{p} + \Gamma_{n,p})$ berechnet werden kann. Dabei gibt $\Gamma_{n,p}$ die Laufzeit einer Kommunikationsphase an, bei der höchstens $\tilde{O}(\frac{n}{p})$ von jedem Prozessor gesendet und empfangen werden [24, S. 556]. Trotz dieser asymptotisch optimalen erwarteten Laufzeit und $O(1)$ Kommunikationsrunden bietet sich eine Implementierung des Algorithmus aufgrund seiner Kompliziertheit nicht an und wurde auch bisher nicht publiziert.

2.3.2 Der Algorithmus von Kühn

Im Rahmen seiner Dissertation zu lokalen Eigenschaften in der Geometrie [49] hat Ulrich Kühn den folgenden randomisierten Algorithmus für die parallele Berechnung der Delaunay-Triangulation in $\mathbb{R}^2$ entworfen, der auf einem grobkörnigen Parallelrechner $CGM(n,p)$ bei $\frac{n}{p} \geq p^{2+\epsilon}$, $\epsilon > 0$, mit hoher Wahrscheinlichkeit eine optimale Laufzeit bei $O(1)$ Kommunikationsrunden hat [49, S. 91]. Das Verfahren benutzt zur Aufteilung der Punkte auf die Prozessoren *random sampling* und sei hier in den wesentlichen Schritten wiedergegeben.

Algorithmus 2.7 *Planare Delaunay-Triangulation DT(S)*

Eingabe: Menge $S \subset \mathbb{R}^2$ von n Punkten in allgemeiner Lage, so daß jeder Prozessor P_i, $1 \leq i \leq p$, genau $\frac{n}{p}$ Punkte speichert. Jeder Punkt ist auf genau einem P_i gespeichert.

Ausgabe: DT(S); jeder Prozessor $P_i, 1 \leq i \leq p$, speichert $\tilde{O}(\frac{n}{p})$ Punkte und Delaunay-Kanten.

1. Die Ebene wird durch *random sampling* zerlegt, indem alle Prozessoren gemeinsam eine zufällige Auswahl $R \subset S$ der Größe $\tilde{O}(\frac{n}{p})$ ermitteln. Jeder Prozessor berechnet dann das Voronoi-Diagramm $VD(R)$, eine Quadrangulierung Q (Zerlegung in Vierecke) von R anhand $VD(R)$ und eine Datenstruktur zur Punktlokalisierung der lokal gespeicherten Teilmenge von S in Q.

2. Jeder Prozessor berechnet die gleiche Verteilung der Vierecke der Quadrangulierung auf die Prozessoren derart, daß benachbarte Vierecke möglichst demselben Prozessor zugeteilt werden. Man erhält Viereckmengen Q_i, welche Teilmengen S_i der Eingabe mit jeweils $\tilde{O}(\frac{n}{p})$ Punkten beinhalten.

3. Nun klassifiziert jeder Prozessor per Punktlokalisierung seine lokalen Punkte danach, zu welchem Q_i sie gehören und sendet sie entsprechend an P_i. Nach der Partitionierung von S durch diesen Kommunikationsschritt wird nebenläufig auf allen Prozessoren ein serieller Algorithmus zur Bestimmung von $DT(S_i)$ ausgeführt.

4. Zum Herstellen der globalen Lösung werden abschließend die Delaunay-Dreiecke identifiziert. Dazu bestimmt jeder Prozessor P_i für jedes Delaunay-Dreieck $\triangle \in DT(S_i)$ den Umkreismittelpunkt $m_\triangle$ und prüft, ob $m_\triangle$ in einem Viereck der Teilquadrangulierung Q_i liegt. Ist dies der Fall, werden $\triangle$ und die zugehörigen Delaunay-Kanten als korrekt markiert, andernfalls wird $\triangle$ als möglicherweise inkorrekt markiert.

Der Algorithmus hat eine erwartete Laufzeit von $\tilde{O}(\frac{n \log n}{p})$, ist also mit hoher Wahrscheinlichkeit optimal. Außerdem benötigt er nur $O(1)$ Kommunikationsrunden, so daß er nicht nur hinsichtlich lokaler Berechnung (mit hoher Wahrscheinlichkeit), sondern auch hinsichtlich des Nachrichtenaustausches asymptotisch optimal ist. Eine Implementierung ist für die Originalarbeit nicht vorgenommen worden; angesichts der verschiedenen Datenstrukturen und Algorithmen ist die Umsetzung in ein paralleles Programm voraussichtlich äußerst zeitaufwendig. Aus demselben Grund kann man weiterhin davon ausgehen, daß die Laufzeit mit vergleichsweise hohen Konstanten behaftet ist, so daß eine praktisch effiziente Umsetzung zweifelhaft erscheint.

2.3.3 Inkrementelles Verfahren

Der serielle inkrementelle Algorithmus von Cignoni et al. wird in demselben Artikel [22] in einer einfachen parallelen Variante präsentiert. Dazu wird das sequentielle Programm und die gesamte Eingabemenge S auf jedem der p Prozessoren repliziert, weil die Autoren von einer Umgebung ausgehen, in der Prozesse zur Laufzeit nicht effizient erzeugt werden können.

Das die Eingabemenge umschließende Rechteck wird dann in p gleich große Rechtecke R_i aufgeteilt. Jeder Prozessor p_i bestimmt mit dem seriellen Algorithmus all jene Delaunay-Dreiecke, die zumindest teilweise in seinem Rechteck R_i liegen. Mehrfachspeicherungen von Delaunay-Kanten, die die Rechteckgrenzen schneiden, werden vermieden, indem jeder Prozessor ein Dreieck nur in die Lösung aufnimmt, wenn sein oberster linkester Punkt im eigenen Rechteck R_i liegt.

Insbesondere die p-fache Replikation der Eingabe ermöglicht zwar eine unabhängige Programmausführung ohne Kommunikation, löst aber nicht das Ressourcenproblem der Speicherknappheit. Da man beim Entwurf paralleler Algorithmen üblicherweise anstrebt, daß sich zu keiner Zeit mehr als $O(\frac{n}{p})$ Daten im Speicher eines einzelnen Prozessors befinden, ist dieser Ansatz zu kritisieren. Die berichteten Speedup-Werte sind gleichzeitig nicht so gut, daß die beschriebene Schwäche wettgemacht würde.

2.3.4 Teile und herrsche

Der einzige deterministische Algorithmus, der im Kontext des CGM-Modells zur Berechnung des planaren Voronoi-Diagramms (bzw. Delaunay-Triangulation) publiziert worden ist, stammt von Diallo et al. [28]. Sie beschreiben ein Teile-und-herrsche-Verfahren, das beim Mischen zweier Voronoi-Diagramme Vor(P) und Vor(Q) O(1) Kommunikationsaufrufe und $O(\frac{n \log n}{p})$ lokale Berechnungsschritte verwendet. Dabei werden die Kanten des Teildiagramms Vor(P) mit Hilfe planarer Multi-Punktlokalisierung in drei Mengen aufgeteilt: Kanten in PQ haben einen ihrer Endpunkte näher an P und einen näher an Q. Die Kanten von PP und QQ haben beide Endpunkte näher an der jeweiligen Menge. Für jede der drei Mengen wird dann bestimmt, welche ihrer Kanten die Kette von Voronoi-Kanten schneidet, die in Vor(P $\cup$ Q) zwischen P und Q liegen muß. Kanten, die die Kette schneiden, werden entsprechend gekürzt. Zur Bestimmung der Kanten der Kette sortiert man die neu gefundenen Endpunkte global. Das Verfahren wird für Vor(Q) analog wiederholt. Die nun berechneten Kanten bilden Vor(P $\cup$ Q) und werden auf die p Prozessoren verteilt. Wendet man dieses Mischverfahren in einem Teile-und-herrsche-Algorithmus an, ergeben sich $O(\log p)$ Misch- und Kommunikationsrunden und daher $O(n \log^2 n/p)$ lokale Berechnungsschritte. Letzteres ist nicht optimal, so daß weiterhin ein deterministischer CGM-Algorithmus mit optimaler lokaler Berechnung gefunden werden muß, um die vorgestellten randomisierten zu verbessern.

2.3.5 Parallele Zerlegung mit Dreieckspfaden

Cignoni et al. beschreiben in ihrem bereits erwähnten Artikel [22] zudem eine parallele Version ihres seriellen Zerlegungsalgorithmus (siehe Abschnitt 2.2.6 dieser Arbeit). Das parallele Verfahren

lädt dabei die gesamte Eingabe und den seriellen Zerlegungsalgorithmus in den Speicher jedes der p Prozessoren.

Stellt man sich den Ablauf des seriellen Algorithmus als (Rekursions)Baum T vor, verläuft der parallele Algorithmus nun folgendermaßen: Der Knoten v_k auf der Ebene $log_2\, p$ von T wird Prozessor p_k zugeordnet und p_k führt dann die algorithmischen Schritte des Baumpfades von der Wurzel zu v_k sowie alle Schritte des Teilbaums unter v_k aus. Auf diese Weise wird der gesamte Baum abgearbeitet, allerdings wird echte Parallelität erst ab Ebene $log_2\, p$ erreicht. Zwar wird verhindert, daß Delaunay-Kanten auf mehreren Prozessoren gespeichert werden, hinsichtlich der ausgeführten Operationen im Gesamtsystem ist dieser Ansatz aber nicht optimal. Außerdem ist wie bei der Parallelisierung des inkrementellen Verfahrens durch dieselben Autoren [22] die p-fache Replikation der Eingabe zu bemängeln.

2.3.6 Parallele Zerlegung durch Transformation

In einem sehr ausführlichen Artikel [15] beschreiben Blelloch et al. einen PRAM-Algorithmus, seine Analyse sowie experimentelle Daten zu seiner praktischen Umsetzung. Der Algorithmus, dessen Details in Kapitel 3 dargestellt werden, benutzt eine geometrische Transformation und dabei prinzipiell auch die Dualität der Delaunay-Triangulation in $\mathbb{R}^2$ zur konvexen Hülle in $\mathbb{R}^3$.

Die Verteilung der Eingabe auf die p Prozessoren wird bei diesem Verfahren so gelöst, daß Delaunay-Pfade als Grenzen zwischen den einzelnen Prozessorbereichen berechnet werden. Innerhalb dieser Grenzen bestimmt jeder Prozessor für den ihm zugewiesenen Bereich mit $O(\frac{n}{p})$ Punkten die lokale Delaunay-Triangulation. Da die Grenzpfade korrekte Teile der globalen Lösung sind, hat man diese danach bereits vollständig berechnet. Die Hauptschwierigkeit besteht daher in der Berechnung der genannten Grenzpfade.

Hierbei kann man sich wieder des Dualitätsprinzips bedienen. Zunächst transformiert man die Eingabe S zu S' auf den Paraboloiden in $\mathbb{R}^3$ mit dem Median des betrachteten Abschnitts als Pol wie in und nach Satz 2.5 beschrieben. Um einen senkrechten Delaunay-Pfad in der Ebene zu bestimmen, projiziert man danach S' in die y-z-Ebene zu S'' und berechnet dann die untere konvexe Hülle von S''. Die Punkte und Adjazenzbeziehungen zwischen ihnen ergeben rücktransformiert in die x-y-Ebene einen vertikalen Delaunay-Pfad durch den Pol des Paraboloiden (siehe Satz 3.10). Im Hinblick auf die Effizienz dieser Vorgehensweise sollte erwähnt werden, daß S'' sich direkt aus S berechnen läßt, der Zwischenschritt über S' ist nur gedanklicher Natur.

Die lokale Berechnung der Teillösung innerhalb der Grenzpfade wird von Blelloch et al. auf ganz ähnliche Weise weitergeführt. Der betrachtete Abschnitt eines Prozessors wird seriell in der Mitte durch einen neu berechneten Grenzpfad in zwei Teilabschnitte geteilt. Dieses Verfahren wird rekursiv so lange fortgesetzt, bis alle Abschnitte in ihrem Innern keine Punkte der Eingabe mehr enthalten. Die Delaunay-Kanten innerhalb der übriggebliebenen Polygone werden rekursiv durch Bestimmung einer Delaunay-Kante (unter Benutzung einer Dualitätseigenschaft) und Aufteilung des Polygons in zwei neue bestimmt.

Das Verfahren der Aufteilung der Eingabe auf die Prozessoren wird auch im Algorithmus von Lee et al. [52] verwendet, deren Arbeit auf einen früheren Artikel von Blelloch et al. zu diesem Thema verweist. Jedoch wird zur Bestimmung der lokalen Delaunay-Triangulation der inkremen-

telle Algorithmus von Cignoni et al. [22] aus Abschnitt 2.2.2 verwendet.

Blelloch et al. geben bei ihrer Analyse an, daß ihr Algorithmus auf einer CREW-PRAM $O(log^3\, n)$ Zeit und $O(n\, log\, n)$ Work benötigt [15, S. 252]. Lee et al. verzichten auf eine theoretische Analyse - ihr Verfahren ist durch die Benutzung des Algorithmus von Cignoni et al. theoretisch nicht optimal.

Grundsätzlich läßt sich feststellen, daß der in den beiden genannten Artikeln gewählte Ansatz zur Aufteilung der Eingabe auf die Prozessoren in zweifacher Hinsicht attraktiv ist. Zum einen läßt er sich relativ leicht implementieren, weil die verwendeten Algorithmen zu den Grundlagen der Algorithmischen Geometrie gehören, so daß die Hauptschwierigkeit in der Parallelisierung liegt. Zum anderen lassen die Experimente der beiden Originalarbeiten auf eine gute Effizienz einer parallelen Implementierung schließen. Theoretisch läßt sich dies daran festmachen, daß die Teillösungen nicht kommunikationsintensiv gemischt werden müssen. Die Hauptaufgabe beim parallelen Entwurf dieses Algorithmus besteht daher in der effizienten Bestimmung der Grenzpfade hinsichtlich der lokalen Berechnung und besonders bezüglich der Kommunikation zwischen den Prozessoren.

2.3.7 Fazit

Der Trend in der Parallelverarbeitung zu Rechnerarchitekturen mit grobkörniger Parallelität macht auch vor geometrischen Algorithmen nicht Halt. Dies wird durch die Entwicklung einer Reihe von grobkörnigen algorithmischen Lösungen für geometrische Probleme in den letzten Jahren belegt. Aus theoretischer Sicht sind die bisherigen Ergebnisse nicht befriedigend, weil es für die Berechnung der planaren Delaunay-Triangulation (bzw. des Voronoi-Diagramms) noch keinen deterministischen parallelen Algorithmus gibt, der im Kontext eines grobkörnigen Modells (BSP oder CGM) entworfen worden ist und eine optimale lokale Laufzeit aufweist. Das nächste Kapitel widmet sich daher der Frage, wie man aus dem Zerlegungsalgorithmus von Blelloch et al., dessen Praxistauglichkeit auf grobkörnigen Rechnern in den zitierten Arbeiten gezeigt wurde, einen Algorithmus für das BSP- bzw. CGM-Modell entwerfen kann, der die oben genannten Kriterien erfüllt, ein einfaches Kommunikationsschema mit wenigen Sende- und Empfangsoperationen hat und für möglichst kleine Werte $\frac{n}{p}$ effizient ist.

Kapitel 3

Varianten des parallelen Zerlegungsalgorithmus

Im vorigen Kapitel ist das Zerlegungsprinzip als gute Basis für einen parallelen CGM- oder BSP-Algorithmus zur Berechnung der planaren Delaunay-Triangulation identifiziert worden. In diesem Kapitel werden zu diesem Verfahren die geometrischen Grundlagen erläutert, der Algorithmus detaillierter vorgestellt und in drei Varianten ausführlich im CGM-Modell analysiert, wovon eine Variante niedrige Kommunikationskosten und optimale lokale Berechnungskomplexität aufweist. Zusätzlich werden die Umsetzung in zwei parallele Programme geschildert und die experimentell erzielten Laufzeiten bewertet.

3.1 Grundlagen der Zerlegungsmethode

Der parallele Algorithmus von Blelloch et al. [15], der eine Punkttransformation von der x-y-Ebene in die x-z-Ebene benutzt, ist bereits in Abschnitt 2.3.6 ansatzweise erläutert worden. An dieser Stelle werden nun zum genauen Verständnis die geometrischen Grundlagen dieses Verfahrens gelegt.

Bemerkung 3.1 *Die Begriffe* Rand(CH(S)) *und* CH(S) *werden in diesem Kapitel synonym verwendet. Wird zur Vereinfachung der Schreibweise gesagt, daß eine Kante e Teil der konvexen Hülle einer Punktmenge S ist, so sei damit ausgedrückt, daß die beiden Endpunkte von e Nachbarpunkte auf der konvexen Hülle von S sind.*

Hinweis 3.2 *Für die Aussagen dieses Abschnitts wird grundsätzlich allgemeine Lage vorausgesetzt, insbesondere sind keine drei Punkte kollinear und die Koordinaten der Punkte einer Punktmenge eindeutig, weil dann auch ihre Transformationsbilder eindeutig sind. (Genau genommen genügt bei der Verwendung vertikaler Grenzpfade die Eindeutigkeit der y-Koordinaten, bei horizontalen der x-Koordinaten).*

Definition 3.3 *Sei $S \subset \mathbb{R}^2$ eine Punktmenge der euklidischen Ebene und sei CH(S) die konvexe Hülle von S. Dann ist*

- *die untere konvexe Hülle von S - bezeichnet als* LowerCH(S) *- definiert als der Teil von CH(S), der vom Punkt* $(0, -\infty)$ *aus sichtbar ist.*
- *die obere konvexe Hülle von S - bezeichnet als* UpperCH(S) *- definiert als der Teil von CH(S), der vom Punkt* $(0, \infty)$ *aus sichtbar ist.*

Definition 3.4 *Ein Punkt p = (*p_1, p_2*) liegt* unterhalb *(bzw.* oberhalb*) einer nicht senkrechten Gerade g(x) = ax+b genau dann, wenn g(*p_1*) >* p_2 *(bzw. g(*p_1*) <* p_2*) gilt, d. h. am x-Wert* p_1 *hat g einen größeren (bzw. kleineren) y-Wert als p.*

Lemma 3.5 *Eine Kante* $e = \overline{pq}$ *ist genau dann Teil der unteren konvexen Hülle von S, wenn kein Punkt aus S unter der (wegen Koordinateneindeutigkeit nicht senkrechten) Gerade* g_e *durch e liegt.*

Beweis: " $\Rightarrow$ ": Sei $e \subseteq LowerCH(S)$. Angenommen, es gäbe ein $p' \in S$ unter g_e, der Gerade durch e. Dann kann nicht $e \subseteq LowerCH(S)$ gelten, weil p' zumindest einen Teil von $\overline{pq}$ von $(0, -\infty)$ aus gesehen verdeckt. Dies ist ein Widerspruch.

" $\Leftarrow$ ": Es liege kein Punkt aus S unter g_e, der Gerade durch e. Wegen der allgemeinen Lage liegt kein weiterer Punkt aus S auf e, und e ist nicht senkrecht. Weil kein Punkt aus S in der unteren Halbebene H_e^- unter (und exklusive) g_e liegt, wird e vom Punkt $(0, -\infty)$ vollständig gesehen und ist daher Teil der unteren konvexen Hülle von S. □

Lemma 3.6 *LowerCH(*$S_1 \cup S_2$*) = LowerCH(LowerCH(*S_1*) ∪ LowerCH(*S_2*)).*

Beweis: Sei für beide Teilbeweise L := $LowerCH(S_1) \cup LowerCH(S_2)$ sowie g_e die Gerade durch die Kante e.

" $\subseteq$ ": Sei Kante $e = \overline{pq} \subseteq LowerCH(S_1 \cup S_2)$. Dann gibt es keinen Punkt $p' \in S_1 \cup S_2$ (und damit weder in S_1 noch in S_2), der unter g_e liegt.

1. Fall, $e \subseteq S_1 \vee e \subseteq S_2$: Dann gilt $e \subseteq LowerCH(S_1) \vee e \subseteq LowerCH(S_2)$ und daher $e \subseteq L$. Weil kein Punkt aus L unter g_e liegt, gilt $e \subseteq LowerCH(L)$.

2. Fall, $e = \overline{pq}$ mit o. B. d. A. $p \in S_1$, $q \in S_2$: Angenommen, $p \notin LowerCH(S_1)$ oder $q \notin LowerCH(S_2)$). Dann gibt es einen Punkt aus S_1 oder S_2 unter $\overline{pq}$, so daß $\overline{pq} \not\subseteq LowerCH(S_1 \cup S_2)$ gelten muß. Dies ist ein Widerspruch, somit folgen $e \subseteq L$ und $e \subseteq LowerCH(L)$.

Dies zeigt $LowerCH(S_1 \cup S_2) \subseteq LowerCH(L)$.

" $\supseteq$ ": Sei Kante $e \subseteq LowerCH(L)$. Dann existiert kein Punkt aus L, der unter g_e liegt. Folglich gibt es keinen Punkt aus $LowerCH(S_1)$ *und* keinen Punkt aus $LowerCH(S_2)$ unter g_e und damit weder aus S_1 noch aus S_2. Da außerdem $e \subseteq S_1 \cup S_2$ gilt, folgt $e \subseteq LowerCH(S_1 \cup S_2)$ und somit $LowerCH(L) \subseteq LowerCH(S_1 \cup S_2)$. □

Das Lemma gilt im Übrigen auch für die gesamte konvexe Hülle, das heißt $CH(S_1 \cup S_2) = CH(CH(S_1) \cup CH(S_2)$ (s. Preparata und Shamos [59, S. 115], ohne Beweis).

Lemma 3.7 *Sei* $S \subset \mathbb{R}^2$ *und seien* $S' := \{(x, y, x^2+y^2) \mid (x, y) \in S\}$ *sowie* $S'' := \{(y, x^2+y^2) \mid (x, y) \in S\}$ *Transformationen von S in den Raum bzw. in die y-z-Ebene. Dann gilt: Jede Kante* e'' *der unteren konvexen Hülle von* S'' *entspricht genau einer projizierten Kante* e' *der unteren konvexen Hülle von* S'*.*

Beweis: Sei e'' $\subseteq$ LowerCH(S''). Die obere Halbebene durch e'' beinhaltet dann alle Punkte aus S''. Aufgrund der Transformationsvorschrift (S'' ist die Projektion von S' in die y-z-Ebene) folgt dann, daß der obere Halbraum durch e', der orthogonal auf der y-z-Ebene steht, alle Punkte aus S' enthält. Also gilt: e' $\subseteq$ LowerCH(S'). □

Die Umkehrung des Lemmas gilt nicht, da nicht jede Kante der dreidimensionalen unteren konvexen Hülle nach der Projektion in die y-z-Ebene auch dort Teil der unteren konvexen Hülle sein muß.

Definition 3.8 *Ein Pfad ist ein kreisfreier, zusammenhängender Teilgraph eines Graphen, in dem jeder Knoten den Grad 1 oder 2 hat. Ein Delaunay-Pfad ist ein Pfad in einer Delaunay-Triangulation.*

Folgerung 3.9 *Die Kanten von $LowerCH(S'')$ bilden rücktransformiert einen Delaunay-Pfad von S in der x-y-Ebene.*

Beweis: Von Satz 2.5 kennen wir die Dualität der unteren konvexen Hülle im Raum mit der planaren Delaunay-Triangulation. Da die Kanten von $LowerCH(S'')$ ihre Entsprechung in Kanten von $LowerCH(S')$ finden, ergeben sie rücktransformiert in die x-y-Ebene Delaunay-Kanten. Man muß also nur noch zeigen, daß sie einen Pfad bilden.

Offensichtlich bildet die untere konvexe Hülle von S'' in der y-z-Ebene einen Pfad. Rücktransformiert man deren Punkte und Kanten in den Raum, ergeben sich eindeutige Punkte der dreidimensionalen konvexen Hülle, weil Koordinateneindeutigkeit vorausgesetzt ist. Damit sind auch die sie verbindenden Kanten eindeutig und bilden einen Pfad, weil $LowerCH(S'')$ einen Pfad zwischen den korrespondierenden Punkten bildet. Aufgrund der Dualität gilt diese Aussage dann auch für die x-y-Ebene. □

Satz 3.10 *Sei $S \subset \mathbb{R}^2$ und sei $S'' = \{(p_y - q_y,\ \|p-q\|^2) \mid p \in S\}$ mit festem $q \in S$. Dann gilt: Die Kanten $e'' \subset LowerCH(P'')$ bilden rücktransformiert in die x-y-Ebene einen Delaunay-Pfad, dessen erster Punkt Teil der LowerCH(S) ist, durch q verläuft und dessen letzter Punkt Teil der UpperCH(S) ist.*

Beweis: Die zu S und S'' korrespondierende Transformation in den Raum ist $S' := \{((p_x - q_x,\ p_y - q_y,\ \|p-q\|^2) \mid p \in S\}$. Der Paraboloid, auf dem die Punkte von S' liegen, hat seinen Pol in $q \in S$ bzw. $q' \in S'$. Somit ist $q'' \in S''$ als unterster Punkt von S'' Teil der konvexen Hülle von S' und daher geht der Pfad in S durch q. Es genügt nun zu zeigen, daß Start- und Endpunkt des Pfades auf der unteren bzw. oberen konvexen Hülle von S liegen.

Seien p''_l und p''_r der linkeste bzw. der rechteste Punkt der unteren konvexen Hülle von S''. Nach Definition der unteren konvexen Hülle stellen sie dann auch das jeweilige Pfadende der unteren konvexen Hülle dar. Die beiden korrespondierenden Punkte $p_l \in S$ und $p_r \in S$ sind somit ebenfalls Pfadenden und haben nun den höchsten negativen bzw. den höchsten positiven y-Abstand aller Punkte aus S zu q. Sie müssen daher das y-Minimum bzw. y-Maximum der Punkte aus S und folglich Elemente der unteren bzw. oberen konvexen Hülle von S sein. □

Das Verfahren läßt sich natürlich auch mit horizontalen Delaunay-Pfaden durchführen. Dazu muß man die Transformationsvorschrift nur leicht abändern und die Punkte vom Paraboloiden

nicht in die y-z-Ebene, sondern in die x-z-Ebene projizieren. S'' sieht dann so aus: $S'' := \{(p_x - q_x, \|p-q\|^2) \mid p \in S\}$ für festes $q \in S$.

Eine Folgerung aus dem obigen Satz besteht darin, daß ein Delaunay-Pfad H die Eingabe S tatsächlich in drei Bereiche teilt, einen links des Pfades, einen rechts des Pfades und H selbst. Blelloch et al. zeigen in ihrem Artikel [15] mit dem folgenden Lemma, daß sich für jeden Punkt aus $S\backslash\{p' \mid p' \in H\}$ sehr einfach bestimmen läßt, ob er links oder rechts von H liegt:

Lemma 3.11 *(Blelloch et al. [15, S. 250f.]) Sei H ein vertikaler Delaunay-Pfad durch den Punkt $p \in S$ und sei L eine senkrechte Gerade durch p. Dann gibt es keinen Punkt in S, der links (bzw. rechts) der Geraden L liegt, aber rechts (bzw. links) des Pfades H.*

Folgerung 3.12 *Zur Berechnung des Delaunay-Pfades genügt es, die Punktmenge P in zwei Mengen L (Punkte links des Medians) und R (Median und Punkte rechts von ihm) zu zerlegen und den Delaunay-Pfad H so zu berechnen: $H'' := LowerCH(LowerCH(L'') \cup LowerCH(R''))$ sowie anschließende Rücktransformation von H'' zu H. Dabei ergeben sich L'' und R'' aus der bekannten Transformation der Punkte in die y-z-Ebene.*

Beweis: Nach Lemma 3.6 folgt: $H'' = LowerCH(L'' \cup R'') = LowerCH(S'')$. Nach Satz 3.10 ist der rücktransformierte Pfad H in der Tat der gewünschte Delaunay-Pfad durch den Median. □

Diese Ergebnisse lassen sich nun als Algorithmus formulieren. Der nächste Abschnitt zeigt dazu zunächst die serielle Vorgehensweise, die auf dem PRAM-Algorithmus von Blelloch et al. [15] basiert, und eine wesentliche Rolle bei der Umsetzung in ein Verfahren für grobkörnige Parallelrechner spielt.

3.2 Serieller und paralleler Algorithmus

Mit den Erkenntnissen des vorigen Abschnitts ist es relativ leicht, einen seriellen Zerlegungsalgorithmus zur Bestimmung der planaren Delaunay-Triangulation zu entwerfen. Man teilt die Eingabe dazu durch einen Delaunay-Pfad in zwei Teile und führt dies rekursiv so lange fort, bis es keinen Punkt mehr gibt, der nicht auf einem Pfad liegt. Übrig bleiben dann nur noch Polygone, die man auf verschiedene Arten Delaunay-triangulieren kann.

Definition 3.13 *Sei $S \subset \mathbb{R}^2$ eine Punktmenge der euklidischen Ebene und sei $CH(S)$ die konvexe Hülle von S. Dann ist*

- *die linke konvexe Hülle von S - bezeichnet als* LeftCH(S) *- definiert als der Teil von $CH(S)$, der vom Punkt $(-\infty, 0)$ aus sichtbar ist.*
- *die rechte konvexe Hülle von S - bezeichnet als* RightCH(S) *- definiert als der Teil von $CH(S)$, der vom Punkt $(\infty, 0)$ aus sichtbar ist.*

Algorithmus 3.14 *Serielle Delaunay-Triangulation: DT (Links, Rechts, Innen)*

Eingabe:

- *Links* bzw. *Rechts*: Delaunay-Pfad, der den aktuellen Bereich links bzw. rechts beschränkt.
- *Innen*: Punkte aus S, die zwischen Left und Right liegen, und gemäß ihrer y-Koordinate aufsteigend sortiert sind.

Ausgabe: Delaunay-Triangulation von S in einer geeigneten Datenstruktur gespeichert.

Erster Aufruf: Delaunay ($LeftCH(S)$, $RightCH(S)$, $Sort_y(S \backslash (LeftCH(S) \cup RightCH(S)))$.

1. Falls (*Innen* = $\varnothing$), dann Delaunay-trianguliere die übriggebliebenen Polygone. Nutze dazu das Verfahren *end_game* von Blelloch et al. [15, S. 255] oder den Algorithmus von Chin und Wang [21]. RETURN.
2. Bestimme den x-Median m von *Innen* mit einem schnellen deterministischen Median-Algorithmus.
3. Berechne die Transformation $Innen'' := \{(p_y - m_y, \|p - m\|^2) \,|\, p \in Innen\}$.
4. Berechne $DPfad'' := LowerCH(Innen'')$. Berechne $DPfad$ durch Rücktransformation von $DPfad''$ in die x-y-Ebene. Die Kanten von $DPfad$ bilden einen Delaunay-Pfad von $LowerCH(Innen)$ über m bis $UpperCH(Innen)$.
5. Erstelle die beiden neuen Teilprobleme $LinksInnen$ und $RechtsInnen$, die diejenigen Punkte aus $Innen$ enthalten, die links bzw. rechts von $DPfad$ liegen.
6. RETURN Delaunay($Links$, $DPfad$, $LinksInnen$) $\cup$ Delaunay($DPfad$, $Rechts$, $RechtsInnen$)

Analyse

Der Algorithmus arbeitet korrekt, denn die konvexe Hülle von S wird so lange durch Delaunay-Pfade rekursiv zerlegt, bis nur noch Polygone ohne innere Punkte übrigbleiben und diese durch korrekte Teilprogramme Delaunay-trianguliert werden. Weil jeder Pfad durch den Median m verläuft und sich die Punkte nach Lemma 3.11 an der Geraden durch m aufteilen, beinhalten LinksInnen und RechtsInnen jeweils höchstens halb so viele Punkte wie Innen. Das Verfahren bricht also nach logarithmisch vielen Schritten in Bezug auf die Eingabegröße ab.

Wird Schritt 1 ausgeführt, so berechnet er alle Delaunay-Kanten zwischen Links und Rechts. Das Verfahren *end_game* benötigt logarithmisch viele Rekursionsschritte und pro Schritt lineare Zeit, schneller ist sogar der Algorithmus von Chin und Wang [21], der die beschränkte Delaunay-Triangulation eines einfachen Polygons in Linearzeit berechnet. Die Schritte 2-5, die in jedem Rekursionsschritt außer dem letzten durchgeführt werden, benötigen alle lineare Zeit. Dies gilt auch für die Bestimmung der unteren konvexen Hülle in Schritt 4, weil die Daten dort vorsortiert sind.

Die Zeitkomplexität des Algorithmus berechnet sich demnach folgendermaßen:

$$T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$$

Rechnet man noch die Kosten für das Sortieren und das Bestimmen der linken und rechten konvexen Hülle von S hinzu (jeweils $O(n \log n)$ oder besser), gilt diese obere Schranke auch für das Gesamtproblem ohne die vereinfachenden Annahmen bezüglich der Eingabe.

Die parallele Version des obigen Algorithmus ist ohne grundlegende Veränderung in der Darstellung von Blelloch et al. [15, S. 249] für das PRAM-Modell konzipiert und analysiert worden und von Lee et al. [52, S. 344ff.] folgendermaßen für eine grobkörnige Umgebung umformuliert worden:

Algorithmus 3.15 *Parallele Delaunay-Triangulation (S)*

Eingabe: Menge S von n Punkten in der Ebene, verteilt gespeichert auf p Prozessoren, so daß jeder Prozessor $O(\frac{n}{p})$ Punkte speichert.

Ausgabe: Delaunay-Triangulation von S, verteilt gespeichert auf p Prozessoren, jeder Prozessor speichert jeweils $O(\frac{n}{p})$ Punkte und Kanten von DT(S).

1. Zerlege die Eingabemenge P in p Teilregionen und weise jede Teilregion genau einem Prozessor zu. Die Zerlegung kann folgendermaßen vorgenommen werden: Bestimme die p-1 Punkte q_i, die die $\lceil \frac{n}{p} i \rceil$-größten Punkte entlang der x-Achse sind. Konstruiere einen Delaunay-Pfad durch jedes q_i, indem alle Punkte auf den Paraboloiden mit Pol q_i transformiert, dann in die y-z-Ebene projiziert werden und schließlich davon die untere konvexe Hülle gebildet wird. Die Transformation berechnet sich wie folgt: $S' = \{(p_y - q_{i_y}, \|p - q_i\|^2) \,|\, p \in S\}$, d.h. der neue x-Wert ist der y-Abstand des Punktes zu q_i, der neue y-Wert der quadrierte euklidische Abstand.

2. Jeder Prozessor bildet die lokale Delaunay-Triangulation mit einem (schnellen) seriellen Algorithmus.

3. Das Mischen der Teilergebnisse besteht lediglich aus dem Löschen der Kanten des linken (oder rechten) Trennpfades auf jedem Prozessor, um Mehrfachspeicherungen von Kanten zu verhindern.

Die Aufteilung der Eingabemenge S auf die einzelnen Prozessoren wird hier durch die Berechnung von p-1 vertikalen Delaunay-Pfaden als Begrenzungen der p Bereiche gelöst. Wie diese Pfade ganz genau berechnet werden, machen Lee et al. [52] in ihrem Artikel nicht deutlich. In den einzelnen Prozessorbereichen zwischen den Grenzpfaden wird zur Berechnung der lokalen Delaunay-Triangulation der theoretisch nicht optimale inkrementelle Algorithmus von Cignoni et al. [22] verwendet.

Blelloch et al. [15] beschreiben ihren Algorithmus zwar sehr ausführlich, die konkrete Umsetzung des PRAM-Pseudocodes in ein paralleles Programm wird allerdings nicht erläutert. Ebenso

geben Lee et al. [52] nur das allgemeine Verfahren an; wie die Umsetzung erfolgte, wird nicht präzisiert. Daher bleibt die Frage offen, wie genau die Grenzpfade berechnet werden, insbesondere das Kommunikationsschema spielt im grobkörnigen Fall die entscheidende Rolle für eine gute parallele Effizienz. Im nächsten Schritt werden deshalb die dafür in Frage kommenden Methoden untersucht und die resultierenden Algorithmen detailliert erläutert.

3.3 Zwei Varianten des parallelen Algorithmus

Für die Entwicklung eines Kommunikationsschemas zur Berechnung der Grenzpfade ist die Beobachtung, daß zunächst grundsätzlich alle Punkte aus S ihren Beitrag zu einem Grenzpfad leisten könnten, sehr wichtig. Ohne zusätzliches Wissen muß man daher alle Punkte transformieren und die untere konvexe Hülle der transformierten Punkte bestimmen, um einen korrekten Grenzpfad zu berechnen. Andererseits weiß man, daß Punkte, die links (bzw. rechts) von einem Grenzpfad liegen, keinen Einfluß mehr auf den Bereich rechts (bzw. links) des Grenzpfades haben können. Dies ist leicht einzusehen, da es sich bei dem Grenzpfad um korrekte Delaunay-Kanten handelt, Kanten einer Triangulation sich nicht überschneiden dürfen und der Grenzpfad von der unteren zur oberen konvexen Hülle verläuft. Eine Delaunay-Kante, die einen Punkt links des Grenzpfades mit einem Punkt rechts des Grenzpfades verbindet, ist daher ausgeschlossen. Hat man einen Grenzpfad korrekt bestimmt, braucht man dann für die Berechnung der nächsten Pfade nur noch einen Teil der Punkte in Betracht zu ziehen.

Dieses Prinzip macht sich das *Top-down*-Schema zunutze. Nachdem der mittlere Pfad von allen Prozessoren berechnet worden ist, werden die Eingabe und die Prozessoren in zwei Teile links bzw. rechts des berechneten Grenzpfades geteilt. Dieses Verfahren wird rekursiv fortgesetzt, bis alle p-1 Grenzpfade bestimmt sind. Das Schema *Sende-an-alle* hingegen berechnet alle Grenzen mit allen Prozessoren. Natürlich ist dieses Verfahren im Hinblick auf die Anzahl aller im System durchgeführten Operationen nicht optimal. Es soll aber geklärt werden, ob die Art des Versendevorgangs Vorteile mit sich bringt.

In den folgenden Abschnitten werden die Schemata detailliert algorithmisch formuliert und analysiert. Zur Beurteilung der Verfahren in einer grobkörnigen parallelen Umgebung findet die Analyse im Kontext des CGM-Modells statt. Die praktische Umsetzung der beiden Varianten in parallele Programme für grobkörnige Parallelrechner sowie experimentelle Daten sind Thema von Kapitel 3.4.

3.3.1 Das Kommunikationsschema *Top-down*

Zur Berechnung der Grenzpfade und der damit einhergehenden rekursiven Aufteilung der Eingabe und der Prozessorgruppen bietet sich ganz offensichtlich eine *Top-down*-Methode an. Die Bezeichnung ist an den Durchlauf eines Bearbeitungsbaumes von der Wurzel zu den Blättern (*top-down*) angelehnt. Zunächst verteilt man die Eingabe abschnittsweise und gleichmäßig auf die Prozessoren. Die Wurzel des binären Bearbeitungsbaumes repräsentiert dann die gesamten Prozessoren und die auf ihnen gespeicherten Eingabedaten (siehe Abbildung 3.1). Nach der Berechnung des Grenzpfades durch den Median des aktuellen Bereichs teilt man die Prozessorgrup-

pe und somit auch die Eingabe mittig in zwei Hälften, repräsentiert durch die beiden Söhne der Wurzel. Dieses rekursive Verfahren bricht ab, wenn die Baumebene $log\, p$ erreicht ist, in der jeder Prozessor einem Baumknoten entspricht. Der zu diesem Knoten korrespondierende Prozessor berechnet dann die lokale Delaunay-Triangulation für diesen Abschnitt innerhalb der vollständig bestimmten Grenzpfade.

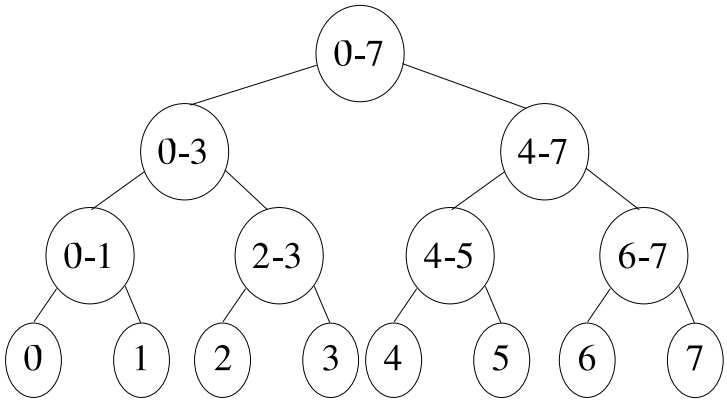

Abbildung 3.1: Prinzip der Eingabeaufteilung anhand eines Bearbeitungsbaumes

In einem Knoten des Bearbeitungsbaumes werden daher der *Grenzpunkt*, das ist der x-Median der zum Knoten korrespondierenden Teilmenge von S, und die *Grenzprozessoren* gespeichert. Grenzprozessoren eines Knotens sind die beiden Prozessoren, auf deren Bereichsgrenze der Grenzpunkt liegt, nämlich der rechteste Prozessor im linken Teilbaum und der linkeste Prozessor im rechten Teilbaum des aktuellen Knotens (s. Abbildung 3.2). Die dritte gespeicherte Information ist die gesamte Prozessorengruppe, die zur Bestimmung eines Grenzpfades benötigt wird. Nach Bearbeitung eines Knotens ist der Grenzpfad des Bereichs korrekt bestimmt.

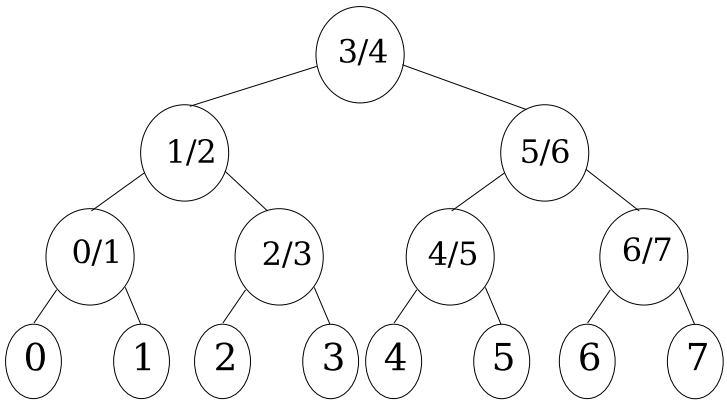

Abbildung 3.2: Jeweilige Grenzprozessoren der Baumknoten

Ziel des Entwurfs sollte es sein, das gegebene Problem in $O(log\, p)$ Supersteps und Kommunikationsrunden bei theoretisch optimaler Berechnungskomplexität von $O(\frac{n\, log\, n}{p})$ und möglichst niedrigem Effizienzquotienten zu lösen. Die logarithmische Zahl der Schritte ergibt sich direkt aus der Traversierung eines Baumpfades von der Wurzel zu einem Blatt.

Algorithmus 3.16 *Deterministische parallele Berechnung der Delaunay-Triangulation einer planaren Punktmenge*

Eingabe: CGM(n, p) mit $\frac{n}{p} \geq p^2 log^2\, p$, eine Menge S von n Punkten in der Ebene gleichmäßig auf $p = 2^k$, $k \in \mathbb{N}$ Prozessoren verteilt sowie die eigene Prozessornummer j mit $0 \leq j \leq p-1$.

Ausgabe: Delaunay-Triangulation DT(S), verteilt gespeichert auf den p Prozessoren, so daß jeder Prozessor $O(\frac{n}{p})$ Knoten und Kanten von DT(S) speichert.

1. Die Eingabemenge S wird parallel in p Abschnitte S_i ($0 \leq i \leq p-1$) zerlegt. Dabei gilt: $\forall i \in \{1,\, ...,\, p-1\} :\, min_x\{S_i\} \geq max_x\{S_{i-1}\}$, d. h. S wird in p x-disjunkte Intervalle aufgeteilt. Abschnitt S_j und die darin enthaltenen Punkte werden dann Prozessor p_j zugewiesen. Falls erforderlich, wird eine gleichmäßige Lastverteilung durchgeführt.

2. Jeder Prozessor p_j versendet den Punkt $m_j \in S_j$ mit dem kleinsten x-Wert an alle anderen Prozessoren.

3. Jeder Prozessor sortiert mit einem schnellen sequentiellen Verfahren seine lokalen Punkte aufsteigend gemäß der y-Koordinate.

4. Jeder Prozessor konstruiert seinen Pfad im Bearbeitungsbaum von der Wurzel bis zu seinem Blatt auf Baumebene log p. So wird vorberechnet, wann bei der Ausführung der folgenden Schleife welche Prozessoren Teil der aktuellen Prozessorgruppe bzw. deren Grenzprozessoren sind und welcher Punkt Grenzpunkt ist.

5. FOR i := 0 TO (log p) - 1 DO

   (a) Durch Abfrage im Pfad des Bearbeitungsbaumes werden die aktuelle Prozessorgruppe, der aktuelle Grenzprozessor und der aktuelle Grenzpunkt m_a bestimmt.

   (b) Jeder Prozessor berechnet seinen lokalen Pfad bezüglich des aktuellen Grenzpunktes m_a, das heißt er berechnet die Transformation $S_j'' := \{(p_y - m_{a_y}, \|p - m_a\|^2 \,|\, p \in S_j\}$, LowerCH($S_j''$) und die Rücktransformation der unteren konvexen Hülle von S_j'' in die x-y-Ebene. Diese bildet einen lokalen (Delaunay-)Pfad H_j.

   (c) Jeder Prozessor sendet seinen lokalen Pfad an den Grenzprozessor seiner Prozessorgruppe.

   (d) Die Grenzprozessoren mischen die erhaltenen lokalen Pfade zu einem gemäß y sortierten Pseudopfad H_p und berechnen dann die obige Transformation für den Pseudopfad. Durch die Bestimmung der unteren konvexen Hülle des transformierten Pseudopfades H_p'' und die Rücktransformation in die x-y-Ebene wird der Gruppenpfad H_g bestimmt.

   (e) Die korrespondierenden Grenzprozessoren versenden aneinander ihren Gruppenpfad.

   (f) Die Grenzprozessoren mischen den empfangenen fremden Gruppenpfad mit dem eigenen zum Mischpfad H_m. Durch erneute Transformation und Bestimmung der unteren konvexen Hülle von H_m'' auf den Grenzprozessoren erhalten diese schließlich den korrekten Grenzpfad H_L bzw. H_R als linke bzw. rechte Begrenzung der lokalen Eingabe in der Ebene.

(g) Vorbereitung auf neue Iteration: Die Prozessorgruppen werden jeweils in zwei Hälften geteilt, im Bearbeitungsbaum wird auf Ebene i+1 abgestiegen und die neuen Punkte von H_L bzw. H_R werden in S_j unter Beibehaltung der Sortierung eingefügt.

6. Jeder Prozessor berechnet lokal die Delaunay-Triangulation $DT(S_j \cup H_L \cup H_R)$ innerhalb der berechneten Grenzpfade. Zur Vermeidung der mehrfachen Speicherung von Grenzpfadkanten löscht jeder Prozessor die Kanten seines linken Grenzpfades; der nullte Prozessor braucht dies nicht zu tun.

Analyse

Die Korrektheit des Algorithmus ergibt sich aus den geometrischen Grundlagen des Zerlegungsverfahrens. Es werden zur Berechnung eines Pfades alle Punkte herangezogen, die unter Umständen einen Einfluß auf den Pfad haben könnten. Nach dem letzten Schleifendurchlauf sind alle Pfade auf korrekte Weise berechnet worden, sie bilden Teilmengen der vollständigen Delaunay-Triangulation. Die noch fehlenden Kanten werden durch den sequentiellen Algorithmus gefunden. Ggf. überflüssige Kanten werden entfernt, so daß eine korrekte globale, verteilt gespeicherte Lösung des Problems berechnet wird.

Im CGM-Modell wird vorausgesetzt, daß im lokalen Speicher jedes Prozessors zu keinem Zeitpunkt mehr als $O(\frac{n}{p})$ Daten enthalten sind. Diese Bedingung, die folglich auch für die Größe von h-Relationen bei der Kommunikation pro Superstep gilt, sollte daher auch für diesen Algorithmus gelten. Problematisch kann dies im Schritt 5c und den darauf aufbauenden Schritten werden, da dort ein Prozessor bis zu $\frac{p}{2}$ Pfade empfängt. Man muß sich daher die Frage stellen, wie groß ein solcher Pfad H_j werden kann.

Theoretisch kann er natürlich eine Größe von $O(\frac{n}{p})$ erreichen, denn in Ausnahmefällen können alle oder fast alle Punkte eines Prozessorabschnitts sehr nah an einer Grenze und dem aktuellen Grenzpunkt liegen und daher Teil des Grenzpfades sein. Allerdings ist dies ein sehr unrealistisches Szenario, denn im Normalfall liegen nur sehr wenige Punkte von S_j wirklich auf H_j, insbesondere dann, wenn p_j kein Grenzprozessor ist. Dies wird auch von Blelloch et al. berichtet [15, S. 255], die selbst für sehr ungleichmäßige Verteilungen eine typische Pfadgröße von etwa $\sqrt{\frac{n}{p}}$ angeben. Eigene Experimente zeigen, daß für gleichverteilte Punktmengen $2\sqrt{\frac{n}{p}}$ eine ausreichend groß gewählte obere Schranke für $|H_j|$ ist.

Hinweis 3.17 *Es wird daher im folgenden von einem* average-case-*Szenario ausgegangen, in dem jedes H_j eine maximale Größe von $O(\sqrt{\frac{n}{p}})$ hat, wenn von $O(\frac{n}{p})$ transformierten Punkten von S_j'' die untere konvexe Hülle gebildet wird.*

In Schritt 5c werden unter dieser Annahme vom Grenzprozessor $O(p\sqrt{\frac{n}{p}}) = O(\sqrt{np})$ Daten empfangen. In keinem anderen Schritt wird dieser Wert noch überschritten, so daß dies als asymptotische Obergrenze für die Größe aller h-Relationen angesehen wird. Dabei muß (für $n \gg p$ und $n, p \in \mathbb{N}$) beachtet werden:

$$\sqrt{np} \leq \frac{n}{p} \Leftrightarrow np \leq (\frac{n}{p})^2 \Leftrightarrow \frac{n}{p} \leq (\frac{n}{p})^2 \cdot \frac{1}{p^2} \Leftrightarrow p^2 \leq \frac{n}{p}$$

Es ist später zu klären, ob hierdurch die Wahl des Effizienzquotienten $\frac{n}{p}$ eingeschränkt ist. Vorher sollen jedoch die einzelnen Schritte des Algorithmus hinsichtlich ihrer Berechnungs- und Kommunikationskomplexität untersucht werden; wir stellen dazu folgendes fest:

Beobachtung 3.18 *Ist eine Punktmenge S gemäß y aufsteigend sortiert, so ist die Transformation von S in die y-z-Ebene S'' gemäß x aufsteigend sortiert, denn der x-Wert eines Punktes $p'' \in S''$ berechnet sich aus der Differenz des y-Wertes von $p \in S$ und einem für alle transformierten Punkte festen Wert. Bilden diese y-Werte in S eine aufsteigend sortierte Folge, so muß diese Sortierung auch für die x-Werte für Punkte in S'' gelten.*

Es folgt nun die Bestimmung der Zeitkomplexität der einzelnen Schritte des Algorithmus.

1. Dieser Schritt ist in $O(1)$ Supersteps mit dem Algorithmus *Parallel Sorting by Regular Sampling* (Abschnitt 1.3.2) durchführbar. Für $\frac{n}{p} \geq p^2$ ist dieses Verfahren theoretisch optimal, es benötigt $O(\frac{n\,log\,n}{p})$ lokale Berechnungsschritte und $O(1)$ Kommunikationsrunden. Dabei werden höchstens $O(\frac{n}{p})$ Daten pro Schritt von einem Prozessor gesendet bzw. empfangen. Wegen der nachfolgenden y-Sortierung ist die lokale Sortierung im letzten Schritt des Algorithmus nicht nötig, es ergibt sich aber durch das Auslassen keine asymptotische Veränderung der Laufzeiten. Dieser Algorithmus garantiert darüber hinaus eine akzeptable Lastverteilung; diese kann innerhalb von $O(1)$ Kommunikationsrunden durch $O(\frac{n}{p})$ lokale Berechnungsschritte noch optimiert werden.

2. Da jeder Prozessor seinen Trennpunkt an p-1 andere Prozessoren versendet, werden $O(p)$ Daten gesendet und empfangen.

3. Lokales Sortieren erfordert $O(\frac{n}{p}\,log\,\frac{n}{p}) = O(\frac{n\,log\,n}{p})$ Berechnungsschritte.

4. Der Baum hat eine Höhe von log p, und dies gilt daher auch für die Anzahl der Knoten auf einem Pfad von der Wurzel zu einem Blatt. Jeder Knoten speichert lediglich die Prozessorgruppe, den Grenzprozessor der Gruppe und den Grenzpunkt, also insgesamt $O(p)$ Daten pro Knoten. Der Pfad ist daher in $O(p\,log\,p)$ Schritten zu konstruieren.

5. Die Schleife durchläuft für jeden Prozessor seinen Pfad im Bearbeitungsbaum von der Wurzel zu einem Blatt und wird folglich $O(log\,p)$-mal ausgeführt.

   (a) Das Auffinden des entsprechenden Knotens im Bearbeitungsbaum ist wegen des Pfaddurchlaufs von der Wurzel zum Blatt durch einen Ebenenabstieg in konstanter Zeit möglich. Die enthaltenen Daten können in $O(p)$ Schritten ausgelesen werden.

   (b) Der aktuelle Grenzpunkt ist nun bekannt, die Transformation der lokalen Punkte benötigt auf jedem Prozessor $O(\frac{n}{p})$ Berechnungsschritte. Da S_j aufsteigend gemäß y sortiert ist, ist jedes S_j'' gemäß seiner x-Koordinate sortiert. Die Berechnung der konvexen Hülle mit *Grahams Scan* hat daher eine Zeitkomplexität von $O(\frac{n}{p})$. Die Rücktransformation erfordert $O(|H_j|)$ Schritte, deren Anzahl nach der Annahme in Hinweis 3.17 mit $O(\sqrt{\frac{n}{p}})$ nach oben abgeschätzt werden kann.

   (c) Nach der oben getroffenen Annahme ist dieser Schritt mit einer h-Relation von $O(p \cdot \sqrt{\frac{n}{p}}) = O(\sqrt{np})$ durchführbar.

(d) Das Mischen der erhaltenen lokalen Pfade zu einem Pseudopfad kann folgendermaßen durchgeführt werden: In die Ergebnisfolge wird fortlaufend der kleinste noch nicht betrachtete Wert aller lokalen Pfade geschrieben. Das Finden dieses Minimums kann man in der Zeit $O(log\, p)$ erreichen, indem man eine Prioritätswarteschlange benutzt. Am Anfang fügt man in sie die ersten Elemente der p Pfade ein. Dann löscht man immer das Minimum, fügt den nächsthöheren Wert desselben lokalen Pfades in die Prioritätswarteschlange und das gelöschte Minimum in den Pseudopfad ein. Da $O(\sqrt{np})$ Elemente in den Pseudopfad geschrieben werden, erfordert dieses Verfahren insgesamt $O(log\, p \cdot \sqrt{np})$ Berechnungsschritte. Für welche Relationen von n und p dies effizient realisierbar ist, wird in Satz 3.19 gezeigt. Die Berechnung des Gruppenpfades ist wegen der y-Sortierung in $O(\sqrt{np})$ Schritten möglich.

(e) Beim Austausch der Gruppenpfade werden $O(\sqrt{np})$ Punkte an einen Nachbarprozessor versendet.

(f) Das Mischen der beiden Gruppenpfade erfolgt in der Zeit $O(\sqrt{np})$. Gleiches gilt für die Bestimmung des endgültigen Grenzpfades durch Berechnung der Transformation, der unteren konvexen Hülle und der Rücktransformation.

(g) Das Aufteilen der Prozessorgruppe in zwei Gruppen hat eine Zeitkomplexität von $O(p)$ und erfordert - je nach Implementierung - maximal das Versenden und Empfangen von $O(p)$ Nachrichten. Das Einfügen der neuen Punkte des gerade berechneten Pfades kann durch Mischen zweier sortierter Folgen in Linearzeit erfolgen.

6. Die Berechnung der lokalen Delaunay-Triangulation für S_j innerhalb der Grenzpfade ist mit einem schnellen sequentiellen Verfahren in $O(\frac{n\, log\, n}{p})$ Schritten lösbar. Benutzt man einen Algorithmus, nach dessen Ausführung man noch Kanten, die außerhalb der Grenzpfade liegen, löschen muß, ist dies in $O(\frac{n}{p} \cdot log(\sqrt{\frac{n}{p}}))$ Schritten durchführbar, indem man für jede Kante entscheidet, ob ihre beiden Endpunkte auf einem Grenzpfad liegen, die Kante selbst aber außerhalb des durch die Grenzpfade beschränkten Abschnitts.

Satz 3.19 *Unter der Voraussetzung, daß $|H_j| = O(\sqrt{\frac{n}{p}})$ gilt, kann man mit diesem Algorithmus die planare Delaunay-Triangulation einer Punktmenge in allgemeiner Lage auf einem grobkörnigen Multicomputer CGM(n, p) mit $\frac{n}{p} \geq p^2\, log^2\, p$ mit optimaler lokaler Berechnungskomplexität von $O(\frac{n\, log\, n}{p})$ bei $O(log\, p)$ Kommunikationsrunden und in gleich vielen Supersteps lösen.*

Beweis: Die dominierenden Komplexitäten sind $O(\frac{n\, log\, n}{p})$, $O(p\, log\, p)$ und $O(log\, p \cdot \sqrt{np})$. Ersteres ist die zu erzielende obere Grenze, es muß also nur gezeigt werden, daß auch die beiden anderen Werte unter dieser Grenze bleiben. $O(p\, log\, p) = O(\frac{n\, log\, n}{p})$ gilt bereits, wenn $p < \frac{n}{p}$ gilt. Wir setzen aber sogar $\frac{n}{p} \geq p^2 log^2 p$ voraus. Dies führt dann auch zu folgendem:

$$O(log\, p \cdot \sqrt{np}) = O(log\, p \cdot \sqrt{\frac{n}{p} \cdot p^2 \cdot \frac{log^2\, p}{log^2\, p}}) = O(log\, p \cdot \sqrt{\frac{n}{p} \cdot \frac{n}{p\, log^2\, p}}) = O(\frac{n}{p})$$

Da Schritt 5c Teil der Schleife ist, die $O(log\, p)$-mal ausgeführt wird, ist die Gesamtkomplexität des Schrittes nach oben beschränkt durch $O(\frac{n\, log\, p}{p}) = O(\frac{n\, log\, n}{p})$. Die übrigen Aussagen bezüglich Kommunikationsrunden, Supersteps und Korrektheit sind nach der Analyse offensichtlich. □

Anhand des (im Vergleich zum grundsätzlich angestrebten Optimalwert) hohen Effizienzquotienten von $\frac{n}{p} \geq p^2 \, log^2 \, p$ wird bereits ein Nachteil dieses Verfahrens deutlich. Das Versenden der lokalen Pfade von allen Prozessoren einer Gruppe an einen einzigen Prozessor kann sich zum Flaschenhals entwickeln. Darüberhinaus findet Parallelität nur auf derselben Baumebene statt. Zwischen jeder Ebene findet eine (dem Verfahren immanente) Synchronisierung statt, weil ein Grenzpfad erst berechnet sein muß, bevor ein Prozessor seinen Beitrag zu einem anderen Grenzpfad korrekt bestimmen kann.

Es stellt sich daher die Frage, ob man durch die Wahl eines anderen Kommunikationsschemas diese Probleme umgehen kann. Eine sehr einfache Alternative stellt das Schema *Sende-an-alle* dar, bei dem jeder Prozessor seinen lokalen Pfad zu jedem der Grenzpunkte berechnet und an den entsprechenden Prozessor versendet. Dieses Verfahren erfordert zwar deutlich mehr lokale Berechnung, es könnte sich aber durch eine bessere Verteilung der Kommunikation als vorteilhaft herausstellen und wird daher im nächsten Abschnitt untersucht.

3.3.2 Das Kommunikationsschema *Sende-an-alle*

Um auf Parallelrechnern mit grobkörniger Parallelität einen hohen Speedup zu erreichen, muß nicht unbedingt ein Algorithmus die beste Wahl sein, der bei seinen lokalen Berechnungen asymptotisch optimal ist. Unter Umständen lohnt es sich, mehr lokale Berechnung als theoretisch optimal wäre durchzuführen, wenn dadurch die Kommunikation günstiger gestaltet werden kann. Diesen Ansatz verfolgt das Schema *Sende-an-alle*. Hierbei finden nur Punkt-zu-Punkt-Kommunikationsaufrufe statt, so daß ein Flaschenhals wie beim *Top-down*-Verfahren nicht ohne Weiteres entstehen kann.

Algorithmus 3.20 *Deterministische parallele Berechnung der Delaunay-Triangulation einer planaren Punktmenge*

Eingabe: Menge S von n Punkten in der Ebene gleichmäßig verteilt auf p Prozessoren einer CGM(n, p) mit $\frac{n}{p} \geq max\{\frac{2^p}{p}, \, p^2 log^2 \, p\}$ sowie die eigene Prozessornummer j mit $0 \leq j \leq p-1$.

Ausgabe: Delaunay-Triangulation DT(S), verteilt gespeichert auf den p Prozessoren.

Die ersten drei Schritte sind identisch zum *Top-down*-Verfahren:

1. Die Eingabemenge S wird parallel in p Abschnitte S_i ($0 \leq i \leq p-1$) zerlegt. Dabei gilt: $\forall i \in \{1, \, ..., \, p-1\} : \, min_x\{S_i\} \geq max_x\{S_{i-1}\}$, d. h. S wird in p x-disjunkte Intervalle aufgeteilt. Abschnitt S_j und die darin enthaltenen Punkte werden dann Prozessor p_j zugewiesen. Falls erforderlich, wird eine gleichmäßige Lastverteilung durchgeführt.

2. Jeder Prozessor p_j versendet den Punkt $m_j \in S_j$ mit dem kleinsten x-Wert an alle anderen Prozessoren.

3. Jeder Prozessor sortiert mit einem schnellen sequentiellen Verfahren seine lokalen Punkte aufsteigend gemäß der y-Koordinate.

4. FOR i := 1 TO p-1 DO

(a) IF ($i \notin \{j, j+1\}$)THEN

i. Transformiere die lokalen Daten in die y-z-Ebene bezüglich Grenzpunkt m_i zu $S_i'' :=:= \{(p_y - m_{i_y}, \|p - m_i\|^2 \,|\, p \in S_j\}$.

ii. Berechne LowerCH(S_i''). Bilde die Rücktransformation und erhalte so $H_i^{p_j}$, den lokalen Pfad bezüglich m_i von Prozessor j.

iii. Versende $H_i^{p_j}$ an Prozessor i und empfange von diesem $H_j^{p_i}$.

5. Prozessor p_j berechnet anhand der bekannten Transformation die beiden lokalen Pfade H_j^j und H_{j+1}^j durch die Grenzpunkte m_j bzw. m_{j+1}.

6. Prozessor p_j mischt die lokalen Pfade, die er von Prozessoren mit kleinerer Nummer erhalten hat, zu einem rechten Pseudopfad H_P^R und die lokalen Pfade, die er von Prozessoren mit größerer Nummer erhalten hat, zu einem linken Pseudopfad H_P^L. Er berechnet dann die Transformationen in die y-z-Ebene $(H_P^L)''$ und $(H_P^R)''$ und deren untere konvexe Hülle. Die Gruppenpfade H_j^j und H_j^{j+1} ergeben sich durch Rücktransformation von LowerCH($(H_P^L)''$) bzw. LowerCH($(H_P^R)''$).

7. IF (j > 0) THEN versende H_j^j an Prozessor j-1 und empfange H_j^{j-1} von ihm.

8. IF (j < p-1) THEN versende H_{j+1}^j an Prozessor j+1 und empfange H_{j+1}^{j+1} von ihm.

9. Berechne die Transformationen in die y-z-Ebene von H_j^{j-1} und H_{j+1}^{j+1} und dann $H_L'' := LowerCH((H_j^j)'' \cup (H_j^{j-1})'')$ sowie $H_R'' := LowerCH((H_{j+1}^j)'' \cup (H_{j+1}^{j+1})'')$. Die in die x-y-Ebene rücktransformierten H_L und H_R sind Delaunay-Pfade, die den Bereich der lokalen Punkte links bzw. rechts begrenzen.

10. Berechne lokal die Delaunay-Triangulation $DT(S_j \cup H_L \cup H_R)$ mit einem schnellen sequentiellen Verfahren innerhalb der berechneten Grenzpfade H_L und H_R. Kanten außerhalb des durch die Pfade begrenzten Bereichs gehören nicht zur Lösung. Zur Vermeidung der mehrfachen Speicherung von Grenzpfadkanten löscht jeder Prozessor die Kanten seines linken Grenzpfades; der nullte Prozessor braucht dies nicht zu tun.

Analyse

Zur Berechnung eines Grenzpfades werden alle Punkte aus S betrachtet und durch mehrfaches Bilden der Transformation und ihrer unteren konvexen Hülle gemäß der geometrischen Grundlagen des Verfahrens die korrekten Grenzpfade bestimmt. Diese Delaunay-Kanten werden durch Schritt 10 zu einer korrekten globalen und verteilt gespeicherten Delaunay-Triangulation ergänzt.

Wie bereits beim *Top-down*-Verfahren wird vorausgesetzt, daß die Länge eines Grenzpfades durch $O(\sqrt{\frac{n}{p}})$ nach oben abgeschätzt werden kann. Damit ergeben sich folgende Zeitkomplexitäten für die einzelnen Schritte:

1. Analog zum *Top-down*-Schema ist dies in $O(1)$ Supersteps und Kommunikationsschritten bei einer Zeit von $O(\frac{n \log n}{p})$ für lokale Berechnung durchführbar. Der Effizienzquotient ist $\frac{n}{p} \geq p^2$.

2. Es werden $O(p)$ Daten gesendet und empfangen.

3. Schnelles Sortieren benötigt $O(\frac{n\,log\,n}{p})$ lokale Berechnungsschritte.

4. Die Schleife wird $O(p)$-mal ausgeführt. Pro Iteration sind für die Transformationen, die Bestimmung der unteren konvexen Hülle der jeweiligen sortierten Folge und die Rücktransformation der Hülle $O(\frac{n}{p})$ Zeitschritte nötig sowie die Kommunikation von $O(\sqrt{\frac{n}{p}})$ Daten mit einen anderen Prozessor.

5. Die Bestimmung der beiden Pfade ist wegen der Sortierung in Linearzeit, d. h. $O(\frac{n}{p})$ Schritten, möglich.

6. Das Mischen der lokalen Pfade zu einem Pseudopfad benötigt wie beim *Top-down*-Ansatz $O(log\,p \cdot \sqrt{np})$ lokale Berechnungsschritte.

7. Es werden $O(\sqrt{\frac{n}{p}})$ Daten gesendet und empfangen.

8. Wie Schritt 7.

9. Das Mischen, die Transformationen, das Bestimmen der unteren konvexen Hülle und die Rücktransformationen benötigen alle $O(\frac{n}{p})$ Zeit, da die Pfade bereits gemäß ihrer y-Koordinaten sortiert sind.

10. Mit einem optimalen Algorithmus lassen sich die fehlenden Kanten in der Zeit $O(\frac{n\,log\,n}{p})$ berechnen und die überflüssigen Kanten analog zum *Top-down*-Schema in $O(\frac{n}{p} \cdot log(\sqrt{\frac{n}{p}}))$ Schritten entfernen.

Satz 3.21 *Unter der Voraussetzung, daß $|H_j| = O(\sqrt{\frac{n}{p}})$ gilt, kann man mit diesem Algorithmus die planare Delaunay-Triangulation einer Punktmenge in allgemeiner Lage auf einem grobkörnigen Multicomputer CGM(n, p) mit $\frac{n}{p} \geq max\{\frac{2^p}{p}, p^2\,log^2\,p\}$ mit optimaler lokaler Berechnungskomplexität von $O(\frac{n\,log\,n}{p})$ Zeitschritten bei $O(p)$ Kommunikationsrunden und in gleich vielen Supersteps lösen.*

Beweis: Die Schritte mit der höchsten Zeitkomplexität erfordern $O(p \cdot \frac{n}{p}) = O(n)$ bzw. $O(log\,p \cdot \sqrt{np})$ Operationen. Für $\frac{n}{p} \geq \frac{2^p}{p}$ gilt:

$$O(n) = O(\frac{p}{log\,n} \cdot \frac{n\,log\,n}{p}) = O(\frac{n\,log\,n}{p}),$$

weil dann p von log n dominiert wird. Für den Fall $\frac{n}{p} \geq p^2 log^2\,p$ gilt $O(log\,p \cdot \sqrt{np}) = O(\frac{n\,log\,n}{p})$. Es werden $O(p)$ Supersteps und Kommunikationsrunden benötigt, weil die Anzahl der Schleifendurchläufe $O(p)$ beträgt. □

Der hohe Effizienzquotient deutet darauf hin, daß dieser Algorithmus sehr viel lokale Berechnung erfordert. Auch $O(p)$ Kommunikationsrunden sind deutlich schlechter als beim *Top-down*-Verfahren. Es wird sich bei der konkreten Implementierung zeigen müssen, ob dieses Schema dennoch Vorteile - wie etwa die Vermeidung von Flaschenhälsen wegen ausschließlicher Punkt-zu-Punkt-Kommunikation - in sich birgt.

3.4 Implementierungen und Experimente

Die beiden im Abschnitt 3.3 vorgestellten Varianten des parallelen Zerlegungsalgorithmus sind für diese Arbeit in der Programmiersprache C++ implementiert worden. Die Nachrichtenkommunikation zwischen den parallel arbeitenden Prozessen wird durch die Bibliotheken LAM/MPI bzw. MPICH (beides sind Implementierungen der MPI-Norm [64]) durchgeführt. Einige geometrische Datenstrukturen und Operationen sowie die graphische Ausgabe werden durch Zuhilfenahme der Bibliothek LEDA [53] realisiert.

Die Experimente zur Bestimmung der Laufzeiten und davon abhängiger Parameter wurden auf zwei parallelen Maschinen durchgeführt. Bei der ersten handelt es sich um einen Beowulf-Cluster[18] aus acht Pentium4-Prozessoren mit jeweils 2,0 GHz Taktfrequenz und 512 MB Hauptspeicher. Sie sind durch Gigabit-Ethernet und einem Gigabit-Switch miteinander verbunden und kommunizieren im Programm via LAM/MPI.

Das zweite Testsystem ist ein Cluster des Typs *hpcLine* von Siemens mit 96 x 2 Pentium III-Prozessoren mit jeweils 850 MHz Taktfrequenz und 512 MB Hauptspeicher pro Doppelprozessorknoten, die in einem SCI-Netzwerk durch verteilte Switches als 12 x 8 2D-Torus miteinander verbunden sind und im Programm via MPICH kommunizieren.

3.4.1 Implementierungen

Vor der Auswertung der experimentellen Daten werden die einzelnen Teile der etwa 2600 Codezeilen umfassenden Implementierung anhand der Schrittfolge der Algorithmen genauer erläutert. Viele Funktionen werden in beiden Schemata verwendet, daher wird nur auf wesentliche Unterschiede explizit hingewiesen.

Zunächst einmal erfolgt das Kommunizieren von Eingabepunkten zwischen Prozessoren durch MPI-Sende- und Empfangsoperationen, die je nach Situation Punkt-zu-Punkt (meistens blockierend) oder kollektiv arbeiten. Eine vorherige Verständigung der Prozessoren über die Größe der versendeten Felder wird bei allen Kommunikationsaufrufen vermieden, indem immer eine feste Feldgröße gesendet wird. Diese Obergrenze reicht für alle Eingaben aus und kann auf den eingesetzten Clusterrechnern schnell verarbeitet werden. Im ersten Element des Feldes wird dabei gespeichert, wie viele Elemente tatsächlich zum versendeten Feld gehören. Der geringfügig höhere Speichereinsatz bewirkt so niedrigere Kommunikationskosten aufgrund von Latenzen.

In einem früheren Stadium der Implementierung kamen selbst geschriebene *Wrapper*-Methoden zum Einsatz, die es ermöglichen, Punkte innerhalb ihrer normalen Datenstrukturen an andere Prozessoren zu senden. Der damit verbundene Overhead wirkt sich allerdings signifikant auf die Laufzeit aus, weshalb nun die oben beschriebene feste Feldlänge benutzt wird.

Die Zerlegung der Eingabe auf die Prozessoren in x-disjunkte Abschnitte erfolgt durch eine abgewandelte Version des Algorithmus Sample-Sort (s. Kapitel 1.3.2). Dabei wird auf die Sortierung in x-Richtung im letzten Schritt verzichtet, wenn keine gleichmäßige Lastverteilung ge-

[18] Ein *Beowulf-Cluster* ist ein Cluster, dessen Hardware aus Standardkomponenten besteht und auf dem nur quelloffene Software zum Einsatz kommt. Die einzelnen Rechner des Clusters sind in einem privaten Netzwerk miteinander verbunden und ausschließlich für die schnelle Lösung rechenintensiver Probleme bestimmt (siehe dazu auch die Webseite des Beowulf-Projekts [11]).

wünscht ist. Letztere ist Teil der Implementierung, hat sich aber bei Experimenten auf den Clusterrechnern als nicht vorteilhaft erwiesen.

Die Bestimmung der Grenzpunkte ist bei erfolgter Sortierung implizit gegeben; im anderen Fall wird das x-Minimum durch einen Aufruf an die entsprechende STL-Methode bestimmt. Die STL-Bibliothek wird auch zum lokalen Sortieren der Punkte in y-Richtung und beim Mischen zweier sortierter Folgen verwendet. Beim *Top-down*-Schema wird dann der binäre Bearbeitungsbaum konstruiert, der mit den benötigten Baumoperationen implementiert worden ist (z. B. werden seine Knoten bei der Initialisierung mit den zugehörigen Informationen (Prozessorgruppe, Grenzprozessoren) belegt). Hingegen werden beim *Sende-an-alle*-Schema in einer Schleife die potentiellen Beiträge zu den einzelnen Grenzpfaden berechnet und nicht-blockierend versendet. Für den Fall p = 2 wird dieser Schritt übersprungen und direkt in den "Nachbarschaftsmodus" gewechselt.

Die Berechnung eines Delaunay-Pfads geschieht durch Transformation der Punkte in die y-z-Ebene und der Bestimmung ihrer unteren konvexen Hülle. Dazu ist der Algorithmus *Grahams Scan* implementiert worden. Die Rücktransformation der Hüllenpunkte geschieht durch Auswahl aus der ursprünglichen Menge anhand der berechneten Indizes (eine geometrische Rücktransformation im eigentlichen Sinne würde wegen des Wurzelziehens numerische Ungenauigkeiten verursachen).

Die lokale Delaunay-Triangulation wird unter Verwendung einer Methode von LEDA berechnet. Die falschen Kanten außerhalb der Grenzpfade werden folgendermaßen bestimmt: Liegen beide Endpunkte einer Kante auf demselben Grenzpfad (hier kommt binäre Suche zum Einsatz), aber die Kante auf dessen äußerer Seite, so wird sie entfernt. Auf ähnliche Art und Weise werden auf jedem Prozessor die rechten Grenzpfade gelöscht, um Doppelspeicherungen zu verhindern.

Die graphische Ausgabe fußt im wesentlichen auf Methoden der LEDA-Bibliothek. Die Bereiche der einzelnen Prozessoren werden verschiedenfarbig dargestellt (siehe Abbildung 3.3). Zur Korrektheitskontrolle wird die parallele Lösung mit der seriellen Lösung von LEDA (letztere in weiß) überlagert, zusätzlich wird die Anzahl der Kanten in beiden Lösungen verglichen. So läßt sich sowohl rechnerisch als auch visuell nachvollziehen, daß das implementierte Programm keine Kanten "vergißt" und keine falschen Kanten berechnet.

Der Delaunay-Algorithmus ist als Objektmethode einer TIN-Datenstruktur realisiert worden. Diese Art der Implementierung soll eine Benutzung des Quellcodes für weitere serielle und parallele geometrische oder TIN-Algorithmen ermöglichen. Das Programm arbeitet im Fall des *Sende-an-alle*-Schemas für alle geraden Anzahlen von Prozessoren, für das *Top-down*-Schema muß p eine Zweierpotenz sein.

Die Version, die Grundlage für die Experimente ist, enthält einige frühere Ideen und Konzepte nicht mehr, meist weil diese sich als zu langsam herausgestellt haben. Dazu gehören der inkrementelle Delaunay-Algorithmus von Cignoni et al. [22] inklusive Hashtabelle und das Verfahren *end_game* von Blelloch et al. [15]. Beide Algorithmen sollten der seriellen Vervollständigung der Delaunay-Triangulation innerhalb der berechneten Grenzpfade dienen. Es zeigte sich aber bereits während der Implementierungsphase, daß die Variante mit der LEDA-Implementierung und dem Löschen außerhalb liegender Kanten deutlich schneller ist.

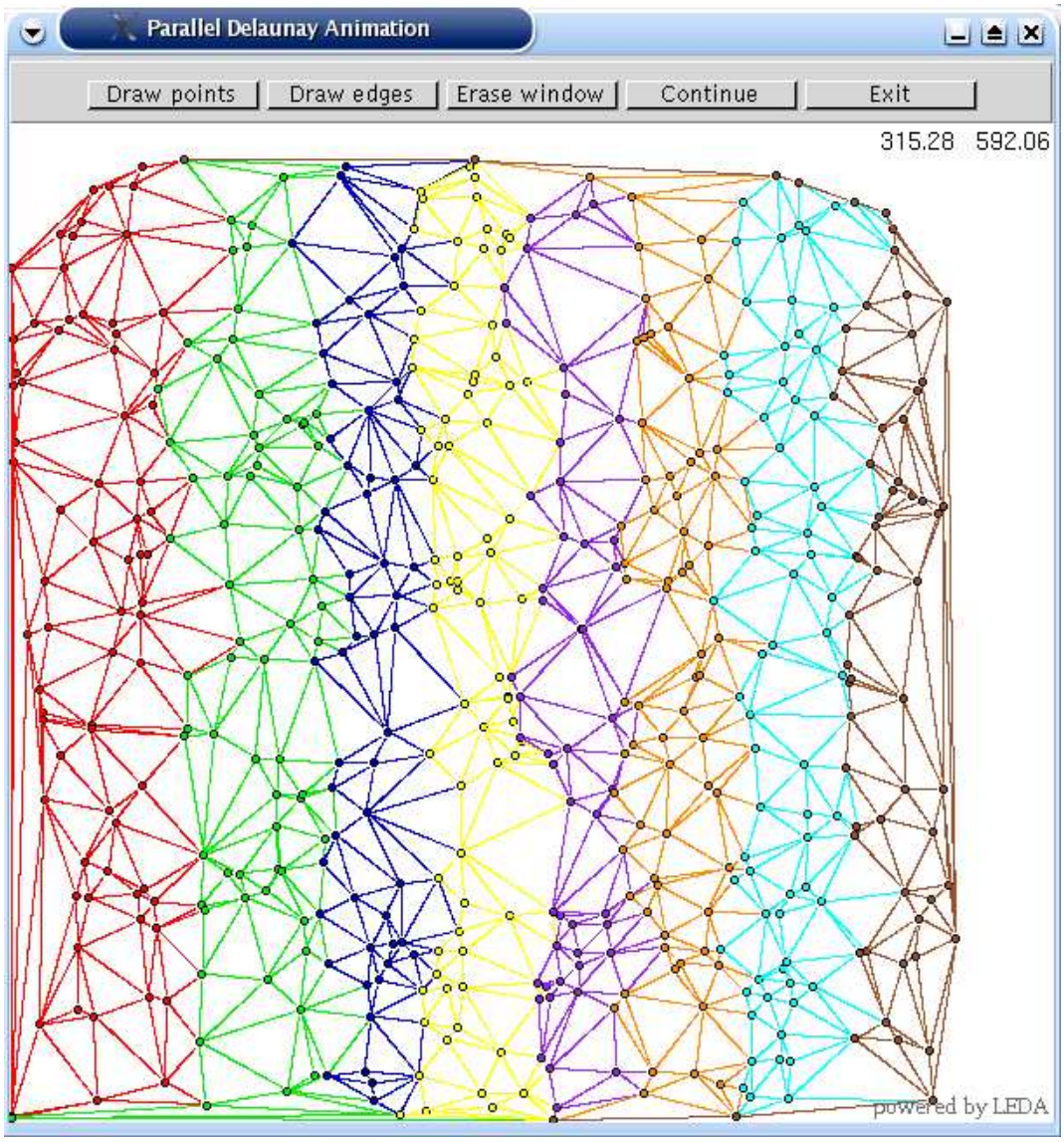

Abbildung 3.3: Graphische Ausgabe der Lösung bei 8 Prozessoren und insgesamt 512 Eingabepunkten

3.4.2 Ergebnisse der Experimente

Zur Ermittlung der Praxistauglichkeit der beiden Kommunikationsschemata sind Experimente auf den bereits erwähnten Clusterrechnern durchgeführt worden. Auf dem Beowulf-Cluster standen dazu acht Prozessoren zur Verfügung, auf dem *hpcLine*-Cluster 32 Prozessoren. Als Eingabe dienen mit Pseudozufallswerten generierte gleichverteilte Datensätze mit jeweils einer Million Punkte innerhalb eines Quadrats in der Ebene; für eine höhere Zahl an Punkten ist die Größe des Hauptspeichers der Rechner nicht ausreichend. In den Abbildungen 3.4 und 3.5 sind die erzielten Speedup-Werte für die getesteten Prozessorzahlen dargestellt.

Die Auswertung der Ergebnisse zeigt, daß das *Top-down*-Schema auf beiden Rechnern und bei allen Prozessorkonfigurationen mindestens gleich gut, meistens sogar deutlich besser ist. Man kann daher die Frage, ob sich durch nicht-blockierendes Senden im Algorithmus Vorteile er-

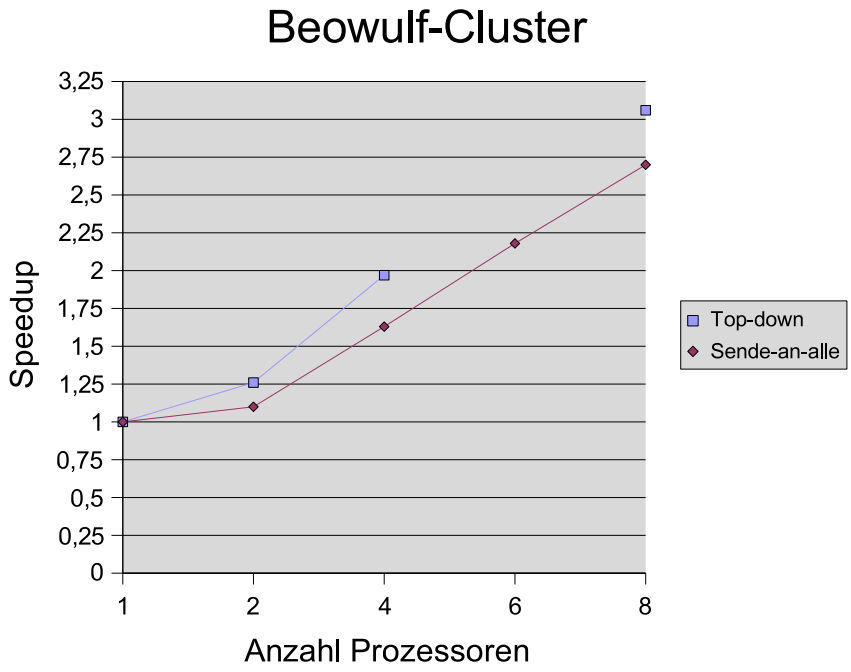

Abbildung 3.4: Speedupwerte auf dem Beowulf-Cluster

geben, klar mit nein beantworten. Bei steigenden Prozessorzahlen zeigt sich auf dem *hpcLine*-Cluster, daß die Speedupwerte nicht linear skalieren. Der Flaschenhals beim *Top-down*-Schema und die hohe Zahl lokaler Berechnungsschritte beim *Sende-an-alle*-Schema wirken sich hier deutlich negativ aus. Es ist außerdem festzuhalten, daß die Werte teilweise deutlich unter denen von Blelloch et al. [15, S. 264] und Lee et al. [52, S. 350] bleiben, die je nach verwendeter Maschine bei acht Prozessoren Speedups um fünf (hier nur etwa drei) erreichen. Woran dies liegt, kann nicht mit Sicherheit gesagt werden, weil wichtige Implementierungsdetails in den genannten Arbeiten nicht erläutert werden. Eine mögliche Erklärung könnte jedoch das niedrigere Verhältnis zwischen Berechnungs- und Kommunikationsgeschwindigkeit der für die beiden Forschungsarbeiten verwendeten Parallelrechner gegenüber den hier eingesetzten Clustersystemen sein.

Festzuhalten bleibt, daß beide Schemata erwartungsgemäß aufgrund der beschriebenen Schwächen nicht linear skalieren, ebensowenig tun dies die Programme, die in den zitierten Artikeln vorgestellt werden. Es stellt sich somit die Frage, ob eine Veränderung des Algorithmus möglich ist, die diese Schwächen ausgleicht und dadurch eine bessere Skalierbarkeit erreicht.

3.5 *Bottom-Up* und *Top-down* kombiniert

Als Nachteile der beiden in Abschnitt 3.3 vorgestellten Schemata haben sich die hohen Effizienzquotienten, bedingt durch ungünstige Kommunikationsstrukturen und/oder viele lokale Berechnungsschritte, erwiesen. Diese Problematik aufgreifend, wird nun ein Algorithmus vorgestellt, der die planare Delaunay-Triangulation mit einem niedrigeren Effizienzquotienten bei optimaler lokaler Berechnungskomplexität in $O(log\, p)$ Kommunikationsschritten konstruiert.

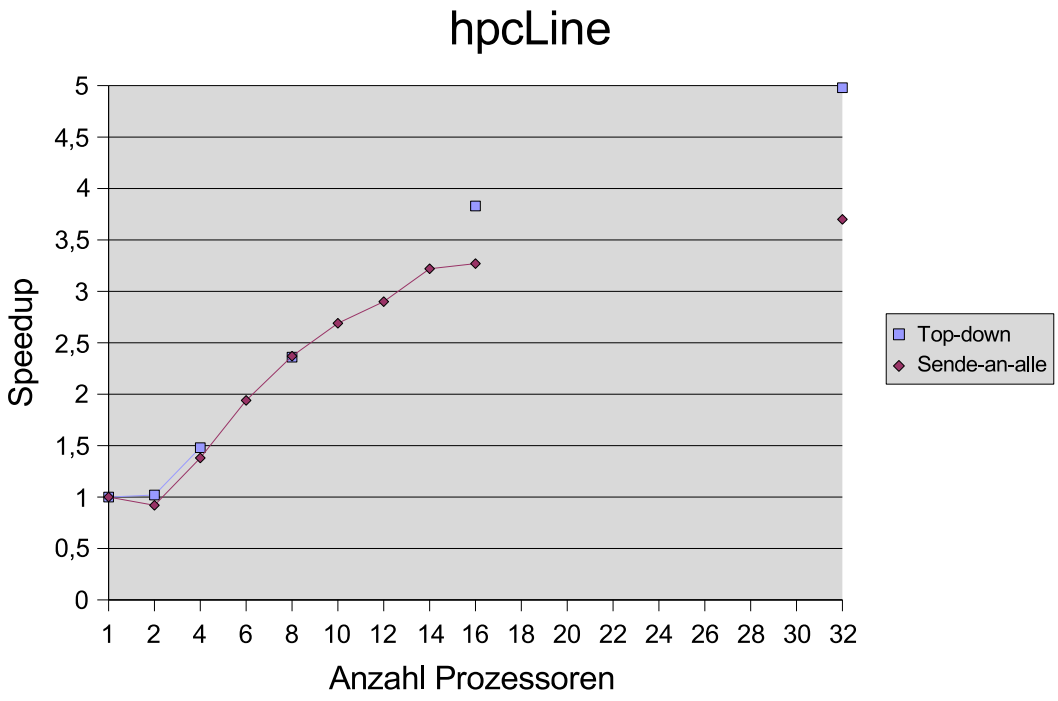

Abbildung 3.5: Speedupwerte auf dem *hpcLine*-Cluster

Die Idee besteht darin, daß jeder Prozessor seinen Pfad im Bearbeitungsbaum zunächst von unten noch oben (*bottom-up*) und dann erst von oben nach unten (*top-down*) durchläuft. Nach $O(\log p)$ Supersteps ist der Grenzpfad der Baumwurzel korrekt berechnet. Auf dem Weg nach unten werden dann die vorläufigen Pfade des *Bottom-up*-Durchlaufs korrigiert, so daß man nach weiteren $O(\log p)$ Supersteps alle Grenzpfade korrekt bestimmt hat.

Bevor der Algorithmus in allen Details beschrieben wird, soll er an einem Beispiel erläutert werden. Gestartet wird dabei in den Blättern des Bearbeitungsbaumes, die jeweils einen Prozessor und den ihm zugewiesenen Abschnitt der Eingabe repräsentieren. Beim Aufstieg in die nächsthöhere Ebene wird der Grenzpfad zwischen den Prozessoren mit gerader Nummer und ihren rechten Nachbarn bestimmt. Dieser Grenzpfad ist nur für den Bereich, den beide Prozessoren zusammen speichern, gültig, da er von Punkten außerhalb dieses Bereichs noch beeinflußt werden kann.

Bei jedem weiteren Aufstieg im Baum ist folgendes zu tun: Zunächst berechnen die beiden neuen Grenzprozessoren ihren lokalen Pfad durch den aktuellen Grenzpunkt. Nach dem Austausch ihrer lokalen Pfade mit dem benachbarten Grenzprozessor bestimmen sie aus ihnen den Grenzpfad. Später in diesem Unterkapitel wird gezeigt, daß dieser für den Eingabeabschnitt, der durch den Baumknoten repräsentiert wird, korrekt ist. Jeder linke Grenzprozessor versendet nun diesen Grenzpfad an den Prozessor aus der eigenen Prozessorgruppe, der in der nächsthöheren Baumebene Grenzprozessor sein wird (s. Abb. 3.6). Dieses Verfahren wird fortgesetzt, bis die Baumwurzel erreicht, der letzte fehlende Grenzpfad aber noch nicht berechnet ist.

Jetzt muß der Baum noch von der Wurzel zu den Blättern durchlaufen werden, um die vorläufigen Grenzpfade zu endgültigen zu korrigieren. Dazu wird zunächst der Grenzpfad zwischen

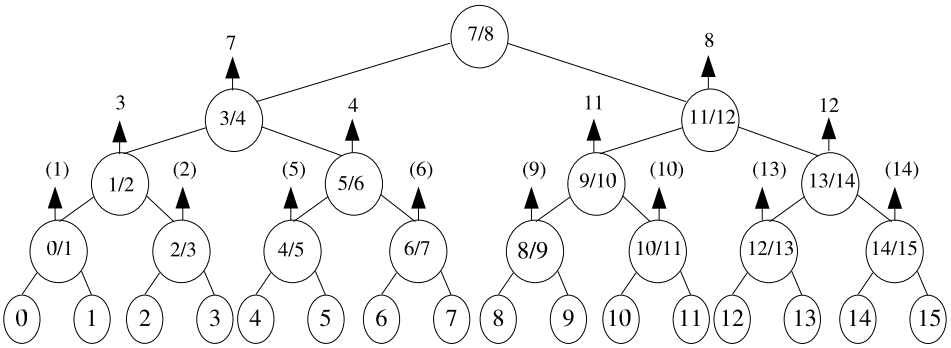

Abbildung 3.6: *Bottom-up-* vor *Top-down*-Durchlauf im Bearbeitungsbaum

den beiden aktuellen Grenzprozessoren berechnet. Dann sendet der linke Grenzprozessor des aktuellen Knotens die beiden bereits korrekt bestimmten Grenzpfade an die zwei Grenzprozessoren des linken Sohnes des aktuellen Knotens; die korrekten Pfade grenzen den aktuellen Bereich von links und rechts ein (dieser Pfad kann auch leer sein, nämlich am linken und rechten Rand der Eingabe). Der rechte Grenzprozessor führt diesen Vorgang analog für den rechten Sohn aus. So fährt man für jede weitere Baumebene fort, so daß die korrekten Grenzpfade im Baum nach unten wandern und alle Bereiche so weit einengen, bis jeder einzelne Prozessorabschnitt korrekt eingegrenzt ist.

Es folgen die algorithmische Darstellung sowie der Korrektheitsbeweis und die Analyse im CGM-Modell.

Algorithmus 3.22 *Deterministische parallele Berechnung der Delaunay-Triangulation einer planaren Punktmenge*

Eingabe: Menge S von n Punkten in der Ebene, CGM(n, p), $\frac{n}{p} \geq p^{1+\epsilon}$, $\epsilon > 0$, $p = 2^k$, $k \in \mathbb{N}$.

Ausgabe: Delaunay-Triangulation DT(S), wobei jeder Prozessor $O(\frac{n}{p})$ Knoten und Kanten von DT(S) speichert.

Die ersten vier Schritte sind nahezu identisch zum *Top-down*-Verfahren:

1. Die Eingabemenge S wird parallel in p Abschnitte S_i $(0 \leq i \leq p-1)$ zerlegt. Dabei gilt: $\forall i \in \{1, ..., p-1\} : min_x\{S_i\} \geq max_x\{S_{i-1}\}$, das heißt S wird in p x-disjunkte Intervalle aufgeteilt. Abschnitt S_j und die darin enthaltenen Punkte werden dann Prozessor p_j zugewiesen. Falls erforderlich, wird eine gleichmäßige Lastverteilung durchgeführt.

2. Jeder Prozessor p_j versendet den Punkt $m_j \in S_j$ mit dem kleinsten x-Wert an alle anderen Prozessoren.

3. Jeder Prozessor sortiert mit einem schnellen sequentiellen Verfahren seine lokalen Punkte aufsteigend gemäß der y-Koordinate.

4. Jeder Prozessor konstruiert seinen Pfad im Bearbeitungsbaum von der Wurzel bis zu seinem Blatt auf Baumebene log p. So wird vorberechnet, wann bei der Ausführung der folgenden

Schleifen welche Prozessoren Teil der aktuellen Prozessorgruppe bzw. deren Grenzprozessoren sind und welcher Punkt Grenzpunkt ist.

5. FOR i := (log p) - 1 DOWNTO 0 DO /* *Bottom-up* */

   (a) Durch Abfrage im Pfad des Bearbeitungsbaumes werden die aktuelle Prozessorgruppe, der aktuelle Grenzprozessor und der aktuelle Grenzpunkt m_a bestimmt.

   (b) Falls p_j Grenzprozessor: p_j integriert für diese Iteration die ggf. in der vorigen Iteration erhaltenen Punkte in S_j und berechnet seinen lokalen Pfad bezüglich m_a, das heißt er berechnet die Transformation $S_j'' := \{(p_y - m_{a_y}, \|p - m_a\|^2 \,|\, p \in S_j\}$, LowerCH($S_j''$) und deren Rücktransformation in die x-y-Ebene. Diese bildet einen *lokalen Pfad* H_j.

   (c) Falls p_j Grenzprozessor: p_j versendet seinen lokalen Pfad an den Nachbarprozessor, der an der durch m_a erzeugten Grenze liegt und empfängt dessen lokalen Pfad.

   (d) Falls p_j Grenzprozessor: Der im vorigen Schritt empfangene lokale Pfad wird von p_j mit dem eigenen zu einem Pseudopfad H_P gemischt. Dann wird dieser Pseudopfad in die y-z-Ebene zu H_P'' wie oben transformiert und der transformierte Gruppenpfad H_G'' :=LowerCH(H_P'') berechnet. Letzterer wird zu H_G in die x-y-Ebene rücktransformiert. Punkte aus H_G, die noch nicht in S_j enthalten sind, werden darin unter Beibehaltung der Sortierung eingefügt.

   (e) Ist p_j linker Grenzprozessor, versendet er H_G an den Grenzprozessor des Vaters des aktuellen Knotens.

6. FOR i := 0 TO (log p) - 1 DO /* *Top-down* */

   (a) Falls p_j Grenzprozessor: p_j sendet den korrekt berechneten Grenzpfad an die beiden Grenzprozessoren des linken (bzw. rechten) Sohnes des aktuellen Knotens, wenn p_j linker (bzw. rechter) Grenzprozessor ist.

   (b) Jetzt erfolgt der Abstieg im Bearbeitungsbaum unter Bestimmung der aktuellen Prozessorgruppe, des aktuellen Grenzprozessors und des aktuellen Grenzpunktes m_a durch jeden Prozessor.

   (c) Falls p_j nun Grenzprozessor: p_j empfängt vom linken (bzw. rechten) Grenzprozessor des Vaterknotens (s. Schritt 6a) dessen korrekt berechneten Grenzpfad, wenn p_j Grenzprozessor eines linken (bzw. rechten) Knotens im Bearbeitungsbaum ist.

   (d) Falls p_j Grenzprozessor: p_j transformiert die beiden empfangenen Grenzpfade und mischt sie mit dem jeweiligen eigenen transformierten Gruppenpfad der aktuell zu bearbeitenden Grenze. Danach berechnet p_j die untere konvexe Hülle dieses Mischergebnisses und transformiert es zurück in die x-y-Ebene zum Grenzpfad H_L bzw. H_R.

7. Jeder Prozessor berechnet lokal die Delaunay-Triangulation $DT(S_j \cup H_L \cup H_R)$ innerhalb der berechneten Grenzpfade. Zur Vermeidung der mehrfachen Speicherung von Grenzpfadkanten löscht jeder Prozessor die Kanten seines linken Grenzpfades; der nullte Prozessor braucht dies nicht zu tun.

Analyse

Die Begrenzung des lokalen Speichers auf $O(\frac{n}{p})$ Datensätze pro Prozessor kann dann problematisch werden, wenn ein Grenzpfad mehr Punkte enthält als in den Speicher passen. Es sind zwar theoretisch Eingaben denkbar, bei denen alle (oder fast alle) Punkte auf einem Grenzpfad liegen, eine solche Verteilung ist jedoch aus praktischer Sicht höchst unwahrscheinlich. Blelloch et al. zeigen in ihrer Arbeit [15, S. 261f.] experimentell für vier verschiedene Verteilungen, daß die Anzahl der Punkte auf den Grenzen nicht zu groß wird. Es wird daher im folgenden ohne wesentliche Einschränkung davon ausgegangen, daß kein Grenzpfad mehr als $O(\frac{n}{p})$ Punkte umfaßt. Gleiches gilt für die im Verlauf des Algorithmus berechneten Gruppenpfade.

Es ist nun zu zeigen, daß die berechneten Grenzpfade korrekt sind. Dazu wird zunächst bewiesen, daß dies auch für die Zwischenergebnisse des aktuellen Baumknotens beim *Bottom-up*-Durchlauf in Bezug auf den korrespondierenden Eingabebereich (aber eben nicht für die gesamte Eingabe) gilt.

Lemma 3.23 *Die beim* Bottom-up*-Durchlauf erstellten Gruppenpfade H_G sind für den Bereich des aktuellen Baumknotens korrekt berechnet.*

Beweis: Die Ebene der Blätter im vollständigen binären Bearbeitungsbaum habe im Gegensatz zur normalen Zählweise die Nummer 0, die Ebene darüber die Nummer 1 usw. Die Behauptung wird durch vollständige Induktion über die Nummer i der Baumebene bewiesen.

Induktionsanfang, i = 1: Die beiden Grenzprozessoren berechnen gemeinsam den Grenzpfad, der aufgrund der in Abschnitt 3.1 dargelegten Ergebnisse korrekt ist.

Wir setzen nun voraus, daß die Behauptung für alle $i_0 \leq i$ stimmt.

Induktionsschluß, i→ i+1: Die beiden Grenzprozessoren eines Knotens der Baumebene i+1 haben in der Iteration i den jeweiligen dort korrekt berechneten Grenzpfad erhalten. Nach Induktionsvoraussetzung und gemäß der geometrischen Grundlagen "kennen" die beiden Grenzprozessoren daher sowohl ihre lokalen Punkte als auch solche Punkte im vom aktuellen Knoten repräsentierten Teil der Eingabe links bzw. rechts ihres lokalen Abschnitts, die Einfluß auf den Grenzpfad nehmen, also auf dem Pfad liegen könnten. Sie sind daher in der Lage, mit dem angegebenen Verfahren einen für den aktuellen Abschnitt korrekten Grenzpfad zu berechnen. □

Aus diesen Ergebnissen läßt sich leicht folgern, daß der erste im Verlauf des Algorithmus garantiert richtig berechnete Grenzpfad derjenige ist, der durch den Grenzpunkt der Wurzel des Bearbeitungsbaumes verläuft, weil der von der Wurzel repräsentierte Abschnitt die gesamte Eingabe umfaßt. In der zweiten Schleife werden die Gruppenpfade zu korrekten Grenzpfaden aktualisiert:

Lemma 3.24 *Die beim* Top-Down*-Durchlauf erstellten Grenzpfade sind korrekt berechnet.*

Beweis: Die Numerierung der Ebenen des Bearbeitungsbaums fange nun wie gewohnt wieder bei der Wurzel mit 0 an, ansonsten ist der Induktionsbeweis über die Baumebenennummer i recht ähnlich zu dem von Lemma 3.23.

Induktionsanfang, i = 0: Es ist bereits aus Lemma 3.23 gefolgert worden, daß der Grenzpfad der Wurzel (also der Grenzpfad durch den Grenzpunkt der Wurzel) korrekt berechnet worden ist.

Wir setzen voraus, daß die Behauptung für alle $i_o \leq i$ stimmt.

Induktionsschluß, $i \rightarrow i+1$: Die beiden Grenzprozessoren eines Knotens der Baumebene i+1 haben nach Schritt 6a den in Iteration i berechneten Grenzpfad erhalten. Zusätzlich ist ihnen auch der Grenzpfad zugesendet worden, der den aktuellen Abschnitt von der anderen Seite begrenzt (dieser kann leer sein). Die beiden Grenzprozessoren kennen daher sowohl die Punkte des aktuellen Abschnitts, die auf dem Grenzpfad liegen könnten (das sind die Punkte des Gruppenpfades) als auch die Punkte links und rechts außerhalb des aktuellen Abschnitts, die den Grenzpfad beeinflussen können. Sie sind daher in der Lage, den korrekten Grenzpfad ihres Knotens zu berechnen. □

Somit ist gezeigt, daß der Algorithmus die Grenzpfade zwischen den Prozessorbereichen korrekt berechnet. Daraus folgt, daß er dann auch die korrekte Delaunay-Triangulation der Eingabemenge S berechnet. Mit welcher Zeit- und Raumkomplexität dies geschieht, wird nun für jeden Schritt einzeln beschrieben.

1. Mit dem Sortieralgorithmus von Goodrich [37] für das BSP-Modell und seine Varianten kann man diesen Schritt sehr effizient durchführen. Das Verfahren sortiert nämlich $O(n)$ Datensätze, die auf einem BSP-/CGM-Computer mit p Prozessoren gleichmäßig verteilt gespeichert sind, in optimaler Zeit bei $O(\frac{log\, n}{log\, h+1})$ Kommunikationsrunden (h-Relationen), wobei $h = \Theta(\frac{n}{p})$ gilt. Da für $\frac{n}{p} \geq p^{\epsilon}$, $\epsilon > 0$, $O(\frac{log\, n}{log\, h+1}) = O(1)$ gilt, darf man bei dem hier vorausgesetzen Effizienzquotienten von einer konstanten Zahl von Kommunikationsrunden ausgehen. Das Verfahren benötigt nicht mehr lokalen Speicher als vorhanden, nämlich $O(\frac{n}{p})$. Die optionale Lastverteilung benötigt $O(1)$ Kommunikationsrunden und $O(\frac{n}{p})$ lokale Berechnungsschritte.

2. Jeder Prozessor sendet ein Datum an alle anderen Prozessoren, somit sind dies $O(p)$ gesendete und empfangene Daten pro Prozessor.

3. Lokales Sortieren hat eine Zeitkomplexität von $O(\frac{n\, log\, n}{p})$.

4. Bei einer Baumhöhe von $O(log\, p)$ und $O(p)$ Daten pro Knoten kann der Pfad des Bearbeitungsbaumes in $O(p\, log\, p)$ Schritten erstellt werden.

5. Die erste Schleife wird insgesamt $O(log\, p)$-mal durchlaufen.

   (a) Der Zugriff auf den Baumknoten erfolgt bei einer entsprechenden Vorbereitung der Datenstruktur in konstanter Zeit. Das Auslesen der Daten geschieht in $O(p)$ Schritten.

   (b) Dieser Schritt ist wegen der Sortierung von S_j in linearer Zeit durchführbar, also unter der Annahme, daß $|S_j| = O(\frac{n}{p})$ ständig gilt, in $O(\frac{n}{p})$ Schritten.

   (c) Ist die Größe eines lokalen Pfades gemäß unserer Annahme durch $O(\frac{n}{p})$ beschränkt, so erfordert dieser Schritt pro Grenzprozessor das Versenden und Empfangen von $O(\frac{n}{p})$ Daten.

(d) Mischen, transformieren und die Bestimmung der konvexen Hülle haben lineare Komplexität wegen der Sortierung. Das Einfügen der neuen Punkte kann ebenfalls durch das Mischen zweier sortierter Folgen geschehen. So werden für diesen Superstep $O(\frac{n}{p})$ Schritte benötigt, wenn kein Pfad mehr Punkte beinhaltet.

(e) Wie Schritt 5c.

6. Auch die zweite Schleife hat $O(log\, p)$ Iterationen.

   (a) Das Versenden und Empfangen des Grenzpfades erfordert das Versenden einer Nachricht der Größe $O(\frac{n}{p})$.

   (b) Die Aktualisierung des Bearbeitungsstatus durch den Abstieg im Baum erfordert $O(p)$ Berechnungsschritte. Je nach Implementierung ist für das Aufteilen der Prozessorgruppen das Versenden und Empfangen einer Nachricht der Größe $O(p)$ vonnöten.

   (c) Wie Schritt 6a.

   (d) Transformieren, das Bestimmen der unteren konvexen Hülle und die Rücktransformation sind wegen der Sortierung in Linearzeit möglich, so daß der Schritt die Zeitkomplexität $O(\frac{n}{p})$ hat.

7. Die lokale Vervollständigung der Lösung läuft nach dem Mischen der Grenzpfade mit S_j in Linearzeit wie in Schritt 6 des *Top-down*-Schemas in optimaler Zeit von $O(\frac{n\, log\, n}{p})$.

Satz 3.25 *Sei S eine planare Punktmenge in allgemeiner Lage, deren Grenzpfade nicht mehr als $O(\frac{n}{p})$ Punkte beinhalten. Dann berechnet der obige Algorithmus auf einem grobkörnigen Multicomputer CGM(n, p) mit $\frac{n}{p} \geq p^{1+\epsilon}$, $\epsilon > 0$, eine verteilt gespeicherte Delaunay-Triangulation von S mit $O(\frac{n}{p})$ lokalem Speicher pro Prozessor mit einer optimalen Berechnungskomplexität von $O(\frac{n\, log\, n}{p})$. Dazu werden $O(log\, p)$ Kommunikationsschritte, bei denen $O(\frac{n}{p})$ Daten pro Schritt versendet werden, und gleich viele Supersteps benötigt.*

Beweis: Die aufwendigsten Schritte haben eine jeweilige Zeitkomplexität von $O(p\, log\, p)$, $O(\frac{n}{p} \cdot log\, p)$ bzw. $O(\frac{n}{p} \cdot log(\frac{n}{p})) = O(\frac{n\, log\, n}{p})$. Diese oberen Schranken können bei $\frac{n}{p} \geq p^{1+\epsilon}$ alle durch den angestrebten Wert $O(\frac{n\, log\, n}{p})$ nach oben abgeschätzt werden. Die übrigen Aussagen ergeben sich aus der vorausgehenden Analyse. □

Wie bereits erwähnt worden ist, bedeutet die Annahme über die Längenbeschränkung der Pfade keine wesentliche Einschränkung, da Blelloch et al. [15, S. 262ff.] bereits experimentell die Funktionsfähigkeit des Verfahrens auch für ungünstige Punktverteilungen gezeigt haben. Man könnte zusätzlich Bedingungen an die Beschaffenheit der Eingabe knüpfen, unter denen das Längenproblem nachweislich nicht auftreten kann. Erste Ansätze deuten dabei jedoch auf eine zu starke Restriktion der Eingabe hin.

Der Algorithmus verbessert unter der getroffenen Annahme die Leistungsmerkmale des bisher einzigen deterministischen Algorithmus, der im Kontext des CGM-Modells publiziert worden ist (von Diallo et al. [28]), da letzterer $O(log\, p)$ Supersteps mit *jeweils* $O(\frac{n\, log\, n}{p})$ lokalen Berechnungsschritten erfordert. Die randomisierten Algorithmen von Dehne et al. [24] bzw. Kühn [49]

benötigen zwar nur $O(1)$ Supersteps (und somit auch nur $O(1)$ Kommunikationsschritte), sind in dieser Kategorie also um den Faktor $log\, p$ besser, aber das für diese Arbeit entwickelte Verfahren ist deterministisch und zudem wesentlich leichter zu implementieren.

Die grundlegenden Ideen des Zerlegungsverfahrens durch Delaunay-Pfade beruhen auf den zitierten Arbeiten von Blelloch et al. [15] bzw. Lee et al. [52], eine Formalisierung dieses Prinzips für ein grobkörniges paralleles Modell wie CGM ist bis dato aber nicht publiziert worden.

3.6 Fazit

In diesem Kapitel sind die Grundlagen der Zerlegungsmethode erläutert und bewiesen worden, danach wurden zwei Kommunikationsschemata für den parallelen Zerlegungsalgorithmus entworfen und analysiert. Die Bedenken hinsichtlich vermuteter Flaschenhälse bei den beiden Schemata *Top-down* und *Sende-an-alle* sind durch experimentelle Daten bestätigt worden. Ausgehend von diesen Erkenntnissen, konnte ein Alternativalgorithmus für das CGM-Modell formuliert werden, der in einem zweifachen Bearbeitungsbaumdurchlauf $O(log\, p)$ Kommunikationsrunden und ebenso viele Supersteps benötigt sowie für $\frac{n}{p} \geq p^{1+\epsilon}$ optimal hinsichtlich der Kosten für Kommunikation und lokale Berechnung arbeitet.

Diese Laufzeiten setzen allerdings voraus, daß die Grenzpfade, die während des Algorithmus berechnet werden, niemals mehr als $O(\frac{n}{p})$ Punkte enthalten. Experimentell ist bereits gezeigt worden [15, S. 262ff.], daß diese Voraussetzung für verschiedene wichtige Verteilungen erfüllt ist; eine detaillierte theoretische Analyse steht allerdings noch aus und könnte Bestandteil weiterer Forschungen sein. Gleiches gilt für den Nachweis der praktischen Performanz des neu entwickelten Kommunikationsschemas. Es ist davon auszugehen, daß sich seine Vorteile nicht bereits bei einer kleinen Anzahl von Prozessoren zeigen, weil hinter der O-Notation die Konstanten des zweifachen Baumdurchlaufs verborgen bleiben. Für größere Prozessorzahlen, bei denen die beiden anderen Varianten schlecht skalieren, dürfte sich jedoch eine deutliche Verbesserung ergeben.

Kapitel 4

Parallelverarbeitung im Kontext höherer Ordnungen

Nachdem gezeigt worden ist, wie man die gewöhnliche Delaunay-Triangulation für viele Punktverteilungen effizient auf einem grobkörnigen Multicomputer berechnen kann, werden in diesem Kapitel darauf aufbauend Delaunay-Triangulationen und Voronoi-Diagramme höherer Ordnung betrachtet. Dazu wird untersucht, wie sich diese Objekte parallel berechnen lassen, wenn die gewöhnliche Delaunay-Triangulation bzw. das gewöhnliche Voronoi-Diagramm Teil der Eingabe ist, also eines dieser Objekte bereits parallel berechnet worden ist und auf den Prozessoren verteilt gespeichert vorliegt. Zur geeigneten Speicherung empfehlen sich z. B. doppelt verkettete Kantenlisten [13, S. 31ff.], abgekürzt DCEL (von engl.: *doubly-connected edge list*) oder im Falle der Triangulation eine TIN-Datenstruktur [48, S. 40f.], die beide für hier benötigte Nachbarschaftsoperationen nur konstante Zeit benötigen.

Der erste Abschnitt beschreibt ein Verfahren zur Berechnung von Delaunay-Triangulationen erster Ordnung, mit denen man einige wichtige Kritierien in einem Netz optimieren kann. Danach wird in Abschnitt zwei eine parallele Version des Algorithmus von D. T. Lee [51] zur Berechnung von Voronoi-Diagrammen höherer Ordnung entworfen. Obwohl Lees und weitere sequentielle Verfahren schon lange bekannt sind (Lees Arbeit datiert bspw. von 1982), existierte bislang noch kein paralleler Algorithmus zur Lösung des Problems. Das hier entworfene Verfahren wird außerdem in den beiden anschließenden Abschnitten für parallele Anwendungen im Kontext von Delaunay-Triangulationen höherer Ordnung genutzt.

Die Algorithmen dieses Kapitels benötigen teilweise für jeden Prozessor die Kenntnis der Punkte in der unmittelbaren Nachbarschaft seines Eingabebereichs. Gemeint ist damit, daß Teile von fremden Prozessorbereichen, die direkt an den eigenen Grenzpfaden liegen, bekannt gemacht werden müssen. Dies schließt die Grenzpfade selbst natürlich mit ein, weshalb im Unterschied zur Verfahrensweise im vorigen Kapitel die Grenzpfade hier auf beiden ihnen angrenzenden Prozessoren abgespeichert sein sollen. Es ergibt sich dadurch keine asymptotische Verschlechterung der Raumkomplexität.

4.1 Delaunay-Triangulationen erster Ordnung

Wie bereits in Abschnitt 1.2.1 gezeigt worden ist, kann man eine Delaunay-Triangulation erster Ordnung (1-ODT) aus einer gewöhnlichen Delaunay-Triangulation erhalten, indem man Kantenflips unabhängiger Kanten durchführt. Will man dies parallel unter Verwendung einer verteilt gespeicherten gewöhnlichen Delaunay-Triangulation machen, müssen sich die Prozessoren in geeigneter Form darüber abstimmen, welche am Rand ihres lokalen Bereichs liegenden Kanten zu wechseln sind und ob diese unabhängig sind oder ob es zu einer Kollision kommt.

Definition 4.1 *Eine* Kollision *tritt auf, wenn zwei abhängige Kanten gewechselt werden sollen und es dadurch zu einer Überschneidung kommt.*

Da ein Kantenwechsel die Struktur einer Triangulation nur innerhalb eines Vierecks verändert, ist die Annahme sinnvoll, daß die Entscheidung, ob eine Kante gewechselt werden soll, auch nur von dem Viereck abhängt, in dem die Kante liegt.

Beobachtung 4.2 *Zwei Kanten, die von verschiedenen Prozessoren gespeichert werden und beide keine Grenzpfadkanten sind, sind voneinander unabhängig, weil sie nicht zu demselben Dreieck gehören können.*

Folgerung 4.3 *Will ein Prozessor nur Kanten wechseln, die keine Grenzpfadkanten sind, so können durch seine Kantenwechsel keine Kollisionen mit Kanten anderer Prozessoren ausgelöst werden.*

Anders gesagt, beeinflußt ein Kantenwechsel keine Dreiecke auf anderen Prozessoren, wenn die zu wechselnde Kante keine Grenzpfadkante ist. Nichtsdestotrotz kann eine solche Operation in Konflikt mit dem Wechsel einer Grenzpfadkante, veranlaßt durch einen benachbarten Prozessor, geraten. Kennen jedoch beide Prozessoren die sich möglicherweise beeinflussenden Bereiche, können sie ohne erneute Kommunikation konfliktfrei eine 1-ODT herstellen.

4.1.1 Algorithmus

Definition 4.4 *Die linke (bzw. rechte)* Dreiecksfolge *eines Grenzpfads besteht aus den Dreiecken, von denen mindestens eine Kante Teil des Grenzpfads ist und deren übrige(n) Kante(n) links (bzw. rechts) des Grenzpfads liegen.*

Lemma 4.5 *Um aus einer Delaunay-Triangulation eine 1-ODT durch Flipping aufgrund der Struktur des jeweiligen Vierecks kollisionsfrei herzustellen, braucht jeder Prozessor neben seinen lokalen Daten zusätzlich nur die Dreiecksfolgen, die an den eigenen aus Delaunay-Kanten bestehenden Grenzpfaden liegen, und die daran angrenzenden Dreiecke zu kennen.*

Beweis: Erhält ein Prozessor die fremden Dreiecksfolgen, sind ihm dadurch alle Kanten bekannt, die mglw. mit seinen lokalen Kanten in Konflikt stehen könnten. Kennt er außerdem die Vierecke, die die Kanten der fremden Dreiecksfolgen umgeben, kann er für diese bestimmen, ob sie gewechselt werden müssen, und so Konflikte selbständig auflösen. □

Das eigentliche Problem besteht somit darin, zu jeder Grenzpfadkante das Dreieck auf der fremden Seite und dessen benachbarte Dreiecke zu kennen. Um zunächst die Dreiecksfolgen zu bestimmen, muß man die Kanten der eigenen Grenzpfade identifizieren und von anderen Prozessoren den dritten Punkt anfordern, der mit der jeweiligen Kante das Dreieck auf der fremden Seite

bildet. Hierbei kann jedoch die Lage des Punktes jenseits des Grenzpfades ohne Zusatzkenntnisse nicht eingegrenzt werden, und ein Prozessor weiß auch nicht, welcher andere Prozessor das gesuchte Dreieck speichert. Deshalb wird zu jeder Kante folgendermaßen ein Anfragepunkt erstellt, der garantiert im fremden Dreieck der aktuellen Kante liegt:

Sei α der kleinste Innenwinkel aller Dreiecke. Dann wird der Punkt q auf der Strecke r, die den Kreismittelpunkt mit dem Kreisrand verbindet und die Grenzpfadkante $e = \overline{uv}$ im rechten Winkel schneidet, derart plaziert, daß die Winkel $\measuredangle uvq$ und $\measuredangle vuq$ jeweils die Größe $\frac{\alpha}{2}$ haben (s. Abb. 4.1). Dies hat zur Folge, daß q definitiv im gesuchten Dreieck liegen muß.

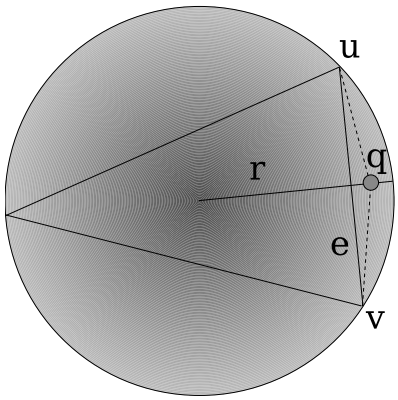

Abbildung 4.1: Berechnung des Anfragepunktes q zur Kante e

Noch vor der Berechnung aller Anfragepunkte werden die Umkreismittelpunkte aller Dreiecke bestimmt und mit den Endpunkten ihres Dreiecks verknüpft. Man wendet dann den Multipunktlokalisierungsalgorithmus von Diallo et al. [28] an, um die gesamten Anfragepunkte zu lokalisieren. Da die von diesem Verfahren zurückgegebenen Umkreismittelpunkte die Information über die Eckpunkte ihres Dreiecks mit sich führen, kann so das jeweils gesuchte Dreieck vollständig bestimmt werden. Danach erstellt man für dessen soeben erhaltene Kanten neue Anfragepunkte und berechnet analog zum obigen Verfahren die angrenzenden Dreiecke.

Algorithmus 4.6 *Parallele Umwandlung einer Delaunay-Triangulation in eine 1-ODT durch Flipping*

Eingabe: CGM(n, p) mit $\frac{n}{p} \geq p$; gleichmäßig auf p Prozessoren verteilt gespeicherte Delaunay-Triangulation einer n-elementigen planaren Punktmenge S.

Ausgabe: Gleichmäßig auf p Prozessoren verteilt gespeicherte Delaunay-Triangulation erster Ordnung von S.

1. Berechne den kleinsten Innenwinkel aller lokalen Dreiecke. Jeder Prozessor sendet seinen Minimalwert an alle anderen Prozessoren und bestimmt aus allen empfangenen p Werten das globale Minimum α.

2. Bestimme die Kanten des linken und des rechten Grenzpfads. Berechne zu jedem Dreieck den Umkreismittelpunkt und zu jeder Grenzpfadkante wie oben beschrieben den Anfragepunkt q. Verwende für jeden Umkreismittelpunkt einen Verbund, der zusätzlich die drei Eckpunkte des Dreiecks speichert.

3. Führe für die Anfragepunkte den Punktlokalisierungsalgorithmus von Diallo et al. [28] aus. Dieser liefert zu jedem Anfragepunkt q den Umkreismittelpunkt des Dreiecks, in dem q liegt. Bestimme aus dem zugehörigen Verbund den gesuchten Endpunkt des Dreiecks.

4. Integriere die neuen Punkte und Kanten der Dreiecksfolgen in die lokale Datenstruktur und berechne zu jeder dieser Kanten mit dem obigen Verfahren einen Anfragepunkt q.

5. Wiederhole Schritt 3 mit den neu erstellten Anfragepunkten und integriere wiederum die so erhaltenen Kanten.

6. Jeder Prozessor berechnet für jede durch ein Viereck eingeschlossene Kante anhand desselben lokalen Kriteriums innerhalb dieses Vierecks, ob sie geflippt werden muß. Kollisionen werden gesucht, dann lokal nach einem für alle Prozessoren festen Kriterium aufgelöst und die übrigbleibenden Flips durchgeführt.

Analyse

Da nach dem Ende des Algorithmus auf jedem Prozessor das umgebende Viereck jeder Kante des Konfliktbereichs bekannt ist und die Kantenwechsel sowie Kollisionen überall auf die gleiche Weise bestimmt und aufgelöst werden, entsteht gemäß der Ergebnisse dieses Abschnitts und von Gudmundsson et al. [39, S. 92-95] eine korrekte und verteilt gespeicherte 1-ODT. Der Zuwachs an Speicherbedarf ist linear in Bezug auf die Grenzpfadlängen und somit unproblematisch.

Die Bestimmung des kleinsten Innenwinkels sowie der Grenzpfadkanten kann durch einen Durchlauf durch die Datenstruktur in linearer Zeit erfolgen. Für die Kommunikation der Minimalwerte sind das einmalige Versenden eines Wertes und Empfangen und Durchsuchen von $O(p)$ Werten erforderlich, was für $\frac{n}{p} \geq p$ unproblematisch ist.[19] Pro Dreieck bzw. Kante benötigt man für die Berechnung der Anfrage- und Umkreismittelpunkte konstante Zeit, insgesamt daher höchstens $O(\frac{n}{p})$. Der Punktlokalisierungsalgorithmus erfordert $O(1)$ Kommunikationsrunden und $O(\frac{n\,log\,n}{p})$ lokale Berechnungsschritte [28, S. 332]. Zur Integration einer Kante in die lokale Datenstruktur wird die passende Stelle in der Zeit $O(log(\frac{n}{p}))$ lokalisiert und die Kante konstanter Zeit eingefügt.

Da es bei $O(\frac{n}{p})$ Dreiecken und Kanten auch nur $O(\frac{n}{p})$ Vierecke geben kann, in denen ggf. ein Kantenwechsel durchgeführt werden muß, kann es auch höchstens ebenso viele Kollisionen geben. Ist das lokale Flipkriterium - wie hier angenommen - in konstanter Zeit pro Kante berechenbar, benötigt der letzte Schritt daher Linearzeit bezüglich der Anzahl der Kanten. Es folgt unmittelbar:

Satz 4.7 *Der Algorithmus 4.6 berechnet bei Verwendung eines Flipkriteriums, das auf dem umgebenden Viereck basiert und in konstanter Zeit pro Kante berechenbar ist, auf einem grobkörnigen Multicomputer CGM(n, p) mit $\frac{n}{p} \geq p$ aus einer verteilt gespeicherten gewöhnlichen Delaunay-Triangulation eine verteilt gespeicherte Delaunay-Triangulation erster Ordnung in $O(\frac{n\,log\,n}{p})$ Zeit bei $O(1)$ Kommunikationsrunden und Supersteps.*

[19] Hier wird im Hinblick auf ein grobkörniges Modell auf das Binärbaumparadigma zur Berechnung des Minimums verzichtet, um Kommunikation zu sparen.

Der Vorteil dieses Algorithmus besteht in dem Versenden und Empfangen weniger großer Nachrichten. Anstatt für jede mögliche Kollision einzelne kleine Nachrichten auszutauschen, werden alle Daten, die für eine mögliche Kollisionsauflösung nötig sein können, in $O(1)$ Kommunikationsrunden bestimmt. Dies erspart im grobkörnigen Umfeld Kosten durch Latenzen und Nachrichtenaufbereitung (man erinnere sich an das in Abschnitt 1.3.3 erwähnte BSP*-Modell, in dem aus diesem Grunde kleine Nachrichten "bestraft" werden).

4.1.2 Netzoptimierungen mit Delaunay-Triangulationen erster Ordnung

Mit dem Algorithmus lassen sich durch 1-OD-Triangulationen mehrere nützliche Kriterien parallel optimieren.

Definition 4.8 *Sei ein TIN T gegeben. Ein Eingabepunkt p von T ist ein* lokales Minimum *(bzw.* lokales Maximum*) genau dann, wenn die Höhenwerte der mit p durch Kanten verbundenen Punkte höher (bzw. niedriger) sind als der Höhenwert von p.*

Die Ergebnisse von Gudmundsson et al. [39, S. 93ff.] besagen, daß man mit 1-OD-Triangulationen die Anzahl lokaler Minima - und damit auch die Anzahl künstlicher Dämme - sowie die Anzahl lokaler Extrema (Minima *und* Maxima) minimieren kann. Gleiches gilt für weitere Netzeigenschaften. Daraus kann man schließen:

Folgerung 4.9 *Für eine auf einem grobkörnigen Multicomputer CGM(n, p) mit* $\frac{n}{p} \geq p$ *gleichmäßig verteilt gespeicherte Delaunay-Triangulation einer Menge S von n Punkten in der Ebene kann man eine optimale 1-OD-Triangulation durch Kanten-Flips unabhängiger Delaunay-Kanten in* $O(\frac{n \log n}{p})$ *Zeitschritten bei* $O(1)$ *Supersteps und Kommunikationsrunden für jedes der folgenden Kriterien erhalten:*

1. Minimierung der Anzahl lokaler Minima, 2. Minimierung der Anzahl lokaler Extrema, 3. Minimierung des Dreiecks mit maximaler Fläche, 4. Minimierung des maximalen Winkels, 5. Maximierung des minimalen Radius eines Umkreises, 6. Maximierung des minimalen Radius eines einschließenden Kreises, 7. Minimierung der Summe der Inkreisradien, 8. Minimierung der Gesamtkantenlänge.

Man kann daher bereits für k = 1 mehrere Kriterien optimieren, die für die Qualität eines Netzes von Bedeutung sind. Diese Optimierung läßt sich bei gegebener Delaunay-Triangulation darüberhinaus effizient parallel berechnen.

4.2 Voronoi-Diagramme höherer Ordnung parallel berechnen

In einem gewöhnlichen Voronoi-Diagramm wird jede Zelle von genau einem Punkt der Eingabemenge S induziert, denn jede Zelle entspricht dem zusammenhängenden Ort von Punkten der Ebene mit demselben nächsten Nachbarn aus S. Das Konzept der Ordnung k erweitert diese Anschauung, indem jede Zelle den Ort der Punkte darstellt, die dieselben *k* nächsten Nachbarn aus S haben. Das gewöhnliche Voronoi-Diagramm hat somit die Ordnung 1 (im Unterschied dazu hat die gewöhnliche Delaunay-Triangulation die Ordnung 0). Zur Veranschaulichung zeigen die Abbildungen 4.2 und 4.3 zu einer planaren Punktmenge ihre Voronoi-Diagramme der Ordnungen 1 bis 4.[20]

[20] Die Berechnung dieser Diagramme erfolgte mit einem Java-Applet von Barry Schaudt [61].

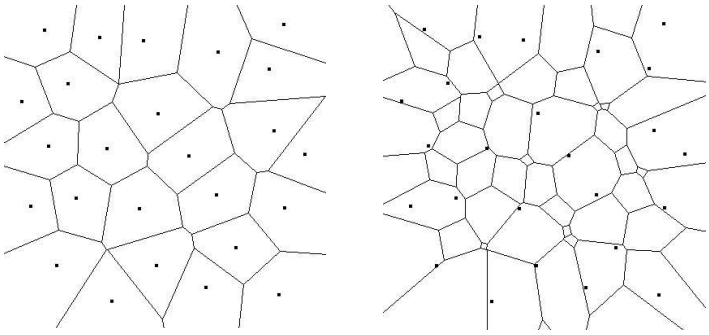

Abbildung 4.2: Voronoi-Diagramme der Ordnungen 1 und 2

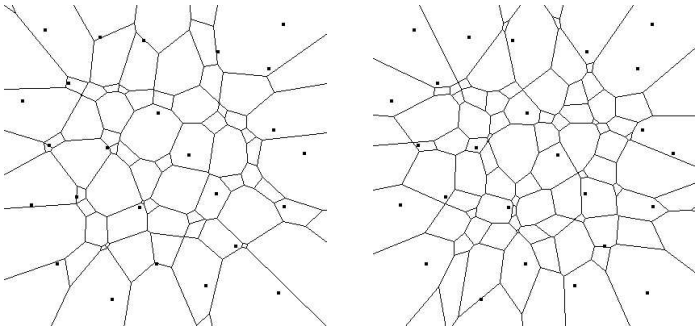

Abbildung 4.3: Voronoi-Diagramme der Ordnungen 3 und 4

4.2.1 Bisherige Arbeiten

Details zu diesem Thema, die hier nicht aufgeführt werden, weil sie den Rahmen dieser Arbeit sprengen, finden sich u. a. bei Agarwal et al. [1], Boisonnat et al. [16], Chan [19], Okabe et al. [56] und im vergleichsweise alten Artikel aus dem Jahre 1982 von D. T. Lee [51], der dort neben grundlegenden Eigenschaften dieser Diagramme auch einen Algorithmus zu ihrer Berechnung beschreibt.

Es handelt sich bei diesem Algorithmus um ein iteratives Verfahren, das aus dem Voronoi-Diagramm (k-1)-ter Ordnung jenes der Ordnung k konstruiert. Die ursprüngliche Laufzeit des Verfahrens beträgt bei $O(k)$ Iterationen, in denen man die gesamte Familie aller Voronoi-Diagramme der Ordnung $\leq k$ berechnen kann, insgesamt $O(k^2 n \log n)$. Diese Schranke wurde jedoch durch den Linearzeitalgorithmus zur Bestimmung des Voronoi-Diagramms eines konvexen Polygons von Aggarwal et al. [2] zu $O(n \log n + nk^2)$ verbessert. Für kleine k ($k < \sqrt{n/\log n}$), wie sie hier betrachtet werden, gibt es bislang keinen besseren deterministischen Algorithmus. Die Raumkomplexität beträgt $O(k^2(n-k))$, weil zu jeder der $O(k(n-k))$ Voronoi-Regionen eines $VD_k(S)$ die k induzierenden Punkte gespeichert werden.

Der Algorithmus basiert auf der Idee, daß man jede Voronoi-Region der Ordnung k-1, induziert durch ein $H' \subset S$ mit $|H'| = k-1$, in Teilregionen zerlegt. Diese Teilregionen sollen jeweils den Ort der Punkte mit denselben k nächsten Nachbarn bilden. Da die Punkte aller dieser Teilregionen bereits dieselben k-1 Nachbarn (nämlich die Punkte aus H') haben, muß zu jeder dieser Teilregionen nur noch der nächste Nachbar aus $S \backslash H'$ gefunden werden. Es genügt also im we-

sentlichen, jede Region von $VD_{k-1}(S)$ auf geeignete Art und Weise anhand von $VD_1(S\backslash H')$ zu zerlegen.

4.2.2 Parallelisierung von Lees Algorithmus

Aus Lees Ergebnissen [51] läßt sich schließen:

Lemma 4.10 *Für die beschriebene Zerlegung einer Voronoi-Region (k-1)-ter Ordnung in Teilregionen der Ordnung k ist nur die Kenntnis der angrenzenden Voronoi-Regionen der Ordnung k-1 nötig.*

Beweis: Lee zeigt in seinem Lemma 5 [51, S. 482], daß von den k-1 nächsten Nachbarn zweier aneinandergrenzenden Voronoi-Regionen (k-1)-ter Ordnung k-2 übereinstimmen. Man betrachte daher eine solche Region R mit s Nachbarregionen R_m, $1 \leq m \leq s$: Die Kanten (auch zu Punkten degenerierte Kanten) zwischen R und diesen Regionen sind Teile von Bisektoren. Punkte auf diesen Bisektoren haben dieselben (k-1) + (k-1) - (k-2) = k nächsten Nachbarn (jeweils k-1 aus den angrenzenden Regionen minus die k-2 doppelt gezählten). Dies sind die Eingabepunkte, die R induzieren, sowie derjenige Punkt, der zwar nicht R, aber das benachbarte R_m mitinduziert.
Also müssen die Punkte aus S, die für Punkte aus R die k-nächsten Nachbarn sein können, zu denen gehören, die ein R_m, aber nicht R mitinduzieren. Eine beliebige Voronoi-Region R' (k-1)-ter Ordnung außerhalb von $\bigcup_{1\leq m\leq s} R_m$ braucht nicht betrachtet zu werden: Wird sie vom gesuchten k-nächsten Nachbarn induziert, gilt dies bereits auch für ein R_m, weil dieses R_m näher an R liegt als R'.

Die s Punkte, die $\bigcup_{1\leq m\leq s} R_m$, aber nicht R induzieren, sind somit die einzigen Punkte aus $S\backslash H'$, die einen Beitrag zu $VD_1(S\backslash H')$ innerhalb von R leisten können, wodurch die Behauptung bewiesen ist. □

Diese Lokalitätseigenschaft kann man sich für eine parallele Umsetzung von Lees Algorithmus zunutze machen. Lees Schema folgend, besteht das Problem darin, jede Voronoi-Region R der Ordnung k-1 in Teilregionen der Ordnung k zu zerlegen, zusammengehörige Teilregionen zu mischen und, falls erforderlich, die so entstandenen neuen Regionen gleichmäßig auf die Prozessoren zu verteilen.

Für die Zerlegung muß ein Prozessor lediglich die Punkte kennen, die die an R grenzenden Voronoi-Regionen induzieren. Diese Kenntnis ist nur für Regionen nicht vorhanden, die nah an den Grenzen zwischen den Bereichen verschiedener Prozessoren liegen. Aus diesem Grunde müssen die erforderlichen Daten im Verlauf des Algorithmus kommuniziert werden. Die Aufgabe besteht nun darin, diese Kommunikation zu optimieren und zu zeigen, daß durch den Algorithmus die gesetzten Schranken bei Raum- und Zeitkomplexität nicht überschritten werden. Letzteres wird durch die folgenden Lemmata erreicht.

Bezeichnung 4.11 *Die Vereinigung der Voronoi-Kanten, die zwischen zwei benachbarten Prozessorbereichen liegen, wird* Voronoi-Grenzpfad *(abgekürzt:* VGP*) genannt. Die Voronoi-Regionen, von denen mindestens eine Kante Teil des Voronoi-Grenzpfades ist, heißen* Grenzregionen.

Lemma 4.12 *Die Grenzregionen am linken (bzw. rechten) Voronoi-Grenzpfad eines Prozessors sind zusammenhängend; ihr dualer Graph ist ein Pfad (genannt:* Zellpfad*), im gewöhnlichen Voronoi-Diagramm der Ordnung 0 ist dies ein Delaunay-Pfad.*

Beweis: Wir zeigen die Behauptung o. B. d. A. für den linken Voronoi-Grenzpfad eines Prozessors p_j, $j > 0$. Ein beliebiger Voronoi-Knoten dieses Pfades habe den Grad d (bei allgemeiner Lage gilt $d = 3$). Zwei dieser d Kanten eines Knotens des Voronoi-Grenzpfades gehören zum Grenzpfad, die übrigen verlaufen entweder links oder rechts des Pfades. Verlaufen d' Kanten davon links, also auf der fremden Seite des Voronoi-Grenzpfades, verlaufen $d_R := d - d' - 2$ rechts. Letztere bilden für $d_R > 0$ eine Grenze zwischen zwei benachbarten und somit zusammenhängenden Voronoi-Regionen am Voronoi-Grenzpfad. Ihre duale Verbindung ist eine Kante (bei Ordnung 0 ist sie eine Delaunay-Kante), wegen ihres fortlaufenden Zusammenhangs entsteht ein Pfad (bei Ordnung 0 ein Delaunay-Pfad). Im Fall $d_R = 0$ existiert keine solche Grenze, die Voronoi-Region setzt sich an diesem Knoten fort. □

Lemma 4.13 *Ein Prozessor erstellt beim Übergang von $VD_{i-1}(S_j)$ zu $VD_i(S_j)$ höchstens $O(\frac{i(n-i)}{p})$ Voronoi-Regionen i-ter Ordnung.*

Beweis durch vollst. Induktion über i: Induktionsanfang, i = 1: In einem gewöhnlichen Voronoi-Diagramm (Ordnung 1) speichert jeder Prozessor $O(\frac{n}{p}) = O(\frac{n-1}{p})$ Voronoi-Regionen, so daß die Behauptung für i = 1 stimmt. Wir setzen nun voraus, daß sie für alle $i_0 \leq i$ gilt.
Induktionsschritt, i→i+1: Ein Prozessor speichert bei einem Voronoi-Diagramm i-ter Ordnung nach Voraussetzung $O(\frac{i(n-i)}{p})$ Regionen. Diese Regionen kann man als völlig eigenständiges Teildiagramm ansehen, das allerdings von jeweils höchstens $O(\frac{i(n-i)}{p})$ Regionen zu beiden Seiten beeinflußt wird. Die Gesamtzahl dieser Regionen beträgt aber immer noch $O(\frac{i(n-i)}{p})$, so daß nach Lee [51, S. 484] das aus dem lokalen Bereich entwickelte Teildiagramm (i+1)-ter Ordnung $O(\frac{(i+1)(n-(i+1))}{p})$ Regionen haben muß. □

Folgerung 4.14 *Ein Zellpfad der Ordnung i auf Prozessor p_j besteht aus höchstens $O(\frac{i(n-i)}{p})$ Voronoi-Regionen der Ordnung i, weil er nicht mehr Regionen haben kann als auf p_j gespeichert sind.*

Zur Eingabe des Algorithmus gehört das gewöhnliche Voronoi-Diagramm von S. An dieses wird die Bedingung gestellt, daß es *x-disjunkt* ist. Damit ist eine verteilte Speicherung des Diagramms derart gemeint, daß die induzierenden Punkte auf einem Prozessor ein Intervall bilden, das x-disjunkt ist zu dem entsprechenden Intervall jedes anderen Prozessors. Auf diese Weise läßt sich bei Kenntnis der Grenzpunkte durch binäre Suche ermitteln, von welchem Prozessor ein Punkt gespeichert wird.

Algorithmus 4.15 *Parallele Berechnung des Voronoi-Diagramms k-ter Ordnung*

Eingabe: CGM(n, p) mit $\frac{n}{p} \geq p$ und $O(\frac{k^2(n-k)}{p})$ lokalem Speicher pro Prozessor; n-elementige planare Punktmenge S und ihr auf den p Prozessoren verteilt gespeichertes x-disjunktes Voronoi-Diagramm, wobei jeder Prozessor $O(\frac{n}{p})$ Voronoi-Regionen und Voronoi-Kanten speichert; Ordnung k des gewünschten Voronoi-Diagramms.

Ausgabe: Voronoi-Diagramm k-ter Ordnung $VD_k(S)$, das derart auf p Prozessoren verteilt ist, daß jeder Prozessor $O(\frac{k(n-k)}{p})$ Regionen mit jeweils k induzierenden Punkten speichert.

1. Die Prozessoren kommunizieren untereinander ihre Grenzpunkte, das sind die x-Maxima ihres lokalen Bereichs.

2. Jeder Prozessor bestimmt die Prozessoren, die zu seinem lokalen Bereich adjazente Voronoi-Regionen speichern. Dazu errechnet er die induzierenden Punkte dieser adjazenten Regionen und durch binäre Suche in den Grenzpunkten den korrekten Prozessorbereich.

3. FOR i := 2 TO k DO

   (a) Jeder Prozessor sendet die Grenzregionen (i-1)-ter Ordnung, die an seinem linken bzw. rechten Voronoi-Grenzpfad liegen (also den linken bzw. rechten Zellpfad), an die jeweils angrenzenden Nachbarprozessoren (sofern vorhanden).

   (b) Mit Lees Algorithmus [51, S. 485f.] wandelt jeder Prozessor p_j seinen lokalen Bereich des $VD_{i-1}(S)$ (also den Bereich innerhalb der aktuellen Voronoi-Grenzpfade) in das Voronoi-Diagramm i-ter Ordnung seines lokalen Bereichs um. An beiden VGP eines Prozessors entstehen neue Regionen, die den aktuellen VGP überdecken.

   (c) Da die einen VGP überlappenden Regionen auf beiden angrenzenden Prozessoren berechnet, aber nur auf einem weiter benutzt werden sollen, erfolgt folgende Aufteilung: Ist i gerade, verläuft der neue Voronoi-Grenzpfad an den links des alten VGP liegenden Kanten der überdeckenden Regionen i-ter Ordnung. Ist i ungerade, verläuft er an ihren rechts des alten VGP liegenden Kanten.

Erläuterungen und Analyse

Die weiter oben zusammengefaßten Ergebnisse dieser Arbeit ergeben zusammen mit denen Lees, daß jeder Prozessor aus seinem lokal gespeicherten Bereich durch zusätzliche Kenntnis der angrenzenden Zellpfade ein korrektes Voronoi-Diagramm der nächsthöheren Ordnung erstellen kann. Es bleibt hier daher nur noch zu zeigen, daß diese lokalen Diagramme derart gespeichert werden, daß sie zusammen ein korrektes globales Diagramm ergeben.

In Abweichung von der üblichen Begrenzung $O(\frac{n}{p})$ der Größe des lokalen Speichers und der Kommunikationspakete, die während eines Supersteps versendet werden können, wird hier als obere Schranke $O(\frac{k^2(n-k)}{p})$ angenommen. Die originäre Begrenzung hat den eigentlichen Sinn, den Speicherbedarf des sequentiellen Algorithmus im parallelen Gesamtsystem mit p Prozessoren asymptotisch nicht zu übersteigen. Durch Anpassung dieser Schranke erfolgt das gleiche hier, da Lees sequentielles Verfahren eine Raumkomplexität von $O(k^2(n-k))$ hat; es stellt daher keine Einschränkung, sondern statt dessen eine Erweiterung des Konzepts dar.

Die Zeit- und Raumkomplexität der einzelnen Schritte ergeben sich wie folgt:

1. Dieser Schritt erfordert eine Kommunikationsrunde, in der $O(p)$ Kopien eines Datums versendet und $O(p)$ Grenzpunkte empfangen werden.

2. Es gibt $O(\frac{n}{p})$ Regionen, zu denen die induzierenden Punkte der Nachbarregionen berechnet werden müssen. Dies geschieht jeweils in konstanter Zeit, da der Bisektor bekannt ist. Die binäre Suche erfordert $O(log\, p)$ Schritte, so daß insgesamt eine Zeit in Höhe von $O(\frac{n\, log\, p}{p})$ für diesen Schritt erforderlich ist.

3. Die Schleife wird $O(k)$-mal ausgeführt.

   (a) Ein Zellpfad hat höchstens die Größe $O(\frac{(i-1)^2(n-(i-1))}{p})$ und kann daher in einem Schritt versendet und empfangen werden. Die Kommunikationspartner dazu sind seit Schritt 1 bekannt.

   (b) Die i-te Iteration von Lees Algorithmus erfordert unter Verwendung der Verbesserung von Aggarwal et al. [2], mit der sich Voronoi-Diagramme konvexer Polygone in Linearzeit berechnen lassen, $O(\frac{i \cdot n}{p})$ Schritte.

   (c) Mit einer geeigneten Datenstruktur wie einer geringfügig abgewandelten DCEL kann man neue Regionen markieren, die den alten VGP überdecken. (Es handelt sich übrigens in der Tat um eine Überdeckung, da keine Kanten des $VD_{i-1}(S)$ in $VD_i(S)$ vorkommen.) Die Datenstruktur erlaubt dann die Bestimmung des neuen VGP in Linearzeit bezüglich der Anzahl seiner Kanten. Diese übersteigt $O(\frac{i(n-i)}{p})$ nicht.
   Die Neueinteilung der Prozessorbereiche, die zwischen je zwei VGP liegen, bewirkt, daß jede Voronoi-Regionen zu genau einem Bereich gehört und die Vereinigung dieser Bereiche das korrekte Voronoi-Diagramm i-ter Ordnung bildet, welches asymptotisch gleichmäßig verteilt gespeichert ist.

Man erhält folgendes Ergebnis:

Satz 4.16 *Sei die Delaunay-Triangulation einer planaren Punktmenge S gleichmäßig verteilt auf einem grobkörnigen Multicomputer CGM(n, p) mit* $\frac{n}{p} \geq p$ *und* $O(\frac{k^2(n-k)}{p})$ *lokalem Speicher pro Prozessor gespeichert. Dann kann man daraus das Voronoi-Diagramm der Ordnung k von S mit einer Zeitkomplexität von* $O(\frac{k^2 n}{p} + \frac{n\, log\, p}{p})$ *bei* $O(k)$ *Supersteps und Kommunikationsrunden, in denen jeder Prozessor höchstens* $O(\frac{k^2(n-k)}{p})$ *Daten sendet und empfängt, berechnen.*

Beweis: Fast alle Schritte des Algorithmus werden durch $O(\sum_{i=1}^{k} \frac{i \cdot n}{p}) = O(\frac{k^2 n}{p})$ dominiert bzw. versenden Daten erlaubter Größe in $O(k) \cdot O(1)$ Kommunikationsrunden, nur Schritt 2 benötigt $O(\frac{n\, log\, p}{p})$ lokale Berechnungsschritte. □

Somit steht ein paralleler Algorithmus zur Verfügung, der ausgehend von einem Voronoi-Diagramm oder einer Delaunay-Triangulation die Familie aller Voronoi-Diagramme der Ordnung $\leq k$ effizient berechnet.

4.3 Parallele Bestimmung der Ordnung einer Triangulation

Um die Ordnung einer Triangulation zu bestimmen, gibt es zwei verschiedene Ansätze, die von Gudmundsson et al. [40, S. 106] beschrieben werden. Davon ist hier aber nur das Verfahren mit

Voronoi-Diagrammen k-ter Ordnung von Interesse, weil dieses für kleine k effizienter ist. In Anlehnung an das sequentielle Verfahren testet dabei jeder Prozessor fortlaufend für jedes lokal gespeicherte Dreieck der Triangulation, ob es die Ordnung k' hat. k' wird dabei solange geeignet erhöht, bis die Bedingung für jedes Dreieck zutrifft.

Ob ein Dreieck t die Ordnung k' hat, bestimmt man durch die Prüfung, ob der k'-nächste Nachbar des Mittelpunktes c des Umkreises von t näher an c liegt als der Kreisrand. Nur wenn c in einer lokal gespeicherten Voronoi-Region des Diagramms der Ordnung k' liegt, läßt sich dieser Vergleich ohne Kommunikation durchführen. Andernfalls muß der Prozessor bestimmt werden, in dessen lokalen Bereich c fällt und der daher auch den k'-nächsten Nachbarn von c kennt. Diese Aufgabe wird durch den planaren Punktlokalisierungsalgorithmus von Diallo et al. [28] gelöst, der auf einem grobkörnigen Multicomputer $O(n)$ Anfragepunkte in einer planaren Subdivision in $O(\frac{n \log n}{p})$ Schritten findet.

Algorithmus 4.17 *Parallele Bestimmung der Ordnung einer verteilt gespeicherten Triangulation*

Eingabe: CGM(n, p) mit $O(\frac{k(n-k)}{p})$ lokalem Speicher; eine gleichmäßig auf p Prozessoren verteilt gespeicherte Triangulation T(S) und das Voronoi-Diagramm VD(S) einer n-elementigen planaren Punktmenge S mit jeweils $O(\frac{n}{p})$ Kanten pro Prozessor.

Ausgabe: Delaunay-Ordnung k von T(S).

1. FOR EACH Dreieck t von T(S) im lokalen Speicher DO
   (a) Berechne den Umkreismittelpunkt von t und speichere ihn in der verwendeten Datenstruktur.
2. k' := 1
3. DO
   (a) Jeder Prozessor berechnet parallel das Voronoi-Diagramm k'-ter Ordnung $VD_{k'}(S)$ und führt ein Preprocessing für Punktlokalisierungen durch. Dazu wird als Ausgangsdiagramm dasjenige verwendet, welches in der vorigen Iteration berechnet wurde bzw. bei k' = 1 in der Eingabe vorliegt.
   (b) Lokalisiere den Umkreismittelpunkt c jedes lokal gespeicherten Dreiecks t von T(S) in $VD_{k'}(S)$ mit Hilfe des Algorithmus von Diallo et al. [28] und bestimme, ob der k'-nächste Nachbar von c im Umkreis von t liegt.
   (c) Jeder Prozessor sendet an alle anderen, ob alle Dreiecke in seinem lokalen Speicher k'-OD-Dreiecke sind.
   (d) k' := 2k'

   WHILE (∃ Dreieck t auf einem beliebigen Prozessor, das kein k'-OD-Dreieck ist)

4. /* Das gesuchte k liegt nun zwischen k' und $2k'$ */
   Führe das Verfahren von Schritt 3 analog einer binären Suche im Intervall $[k', 2k']$ durch und bestimme so die korrekte Ordnung k von T. Benutze als Startdiagramm jeweils $VD_{k'}(S)$, das bereits in Schritt 3 berechnet worden ist.

Analyse

Findet man ein Dreieck, in dessen Umkreis mindestens k' Punkte liegen, wird k' verdoppelt und das Verfahren wiederholt. Nach $O(log\, k)$ Schritten kann man daher ein Intervall eingrenzen, in dem das gesuchte k liegt. Nach weiteren $O(log\, k)$ Schritten kann man durch binäre Suche in diesem Intervall das korrekte k bestimmen. Die Korrektheit des Algorithmus folgt aus der Korrektheit des seriellen Algorithmus, des Punktlokalisierungsalgorithmus von Diallo et al. [28] in Schritt 3b und des Algorithmus zur Bestimmung des Voronoi-Diagramms höherer Ordnung dieser Arbeit.

Gudmundsson et al. [40, S. 106] benutzen in ihrer Arbeit ein Ergebnis von Ramos [60], mit dessen Hilfe sich ein Voronoi-Diagramm k'-ter Ordnung in $O(k'n \log n)$ Schritten herstellen und auf Punktlokalisierungsanfragen vorbereiten läßt. Sie folgern daraus, daß der Gesamtalgorithmus eine serielle Zeitkomplexität von $O(kn \log n \log k)$ hat. Diese obere Schranke läßt sich durch die Verwendung von Lees Algorithmus [51] mit der Verbesserung von Aggarwal et al. [2] zu $O(log\, k(n \log n + nk^2))$ verändern, was für $k = O(log\, n)$ eine Beschleunigung darstellt. Für die Parallelisierung läßt sich feststellen:

Satz 4.18 *Die Ordnung einer planaren Triangulation T(S), die zusammen mit dem Voronoi-Diagramm VD(S) verteilt auf einem grobkörnigen Multicomputer CGM(n, p) gespeichert vorliegt, läßt sich mit einer Zeitkomplexität von* $O(log\, k \cdot \frac{nk^2 + n \log n}{p})$ *in* $O(log\, k)$ *Supersteps und Kommunikationsrunden berechnen.*

Beweis: Die ersten beiden Schritte haben offensichtlich eine Zeitkomplexität von $O(\frac{n}{p})$ bzw. $O(1)$. Die erste Schleife wird $O(log\, k)$-mal ausgeführt, Schritt 3a benötigt in jeder Iteration $O(\frac{k^2 n + n \log p}{p})$ Zeit. Demgegenüber hat Schritt 3b in jeder Iteration eine Zeitkomplexität von $O(\frac{n \log n}{p})$. Die beiden letzten Teilschritte der Schleife werden hinsichtlich lokaler Berechnung von den ersten beiden dominiert, es ist aber eine Kommunikationsrunde pro Iteration notwendig. Da die zweite Schleife dieselbe Zeitkomplexität wie die erste hat und die Berechnung des Voronoi-Diagramms k'-ter Ordnung schrittweise erfolgen kann, ergibt sich eine Gesamtkomplexität von $O(log\, k \cdot \frac{nk^2 + n \log n}{p})$ für lokale Berechnung und von $O(log\, k)$ für die Anzahl der Supersteps und Kommunikationsrunden. □

Nach einer Verbesserung des seriellen Algorithmus für $k = O(log\, n)$ erreichen wir hiermit auch eine effiziente Parallelisierung, wenn das Voronoi-Diagramm Teil der Eingabe ist. Daraus folgt zusätzlich, daß das Problem ohne vorliegendes Voronoi-Diagramm optimal parallelisiert werden kann, wenn es einen optimalen CGM-Algorithmus zur Berechnung des Voronoi-Diagramms gibt.

4.4 Parallel nützliche k-OD-Kanten bestimmen

Das Verfahren, aus allen k-OD-Kanten die nützlichen herauszufiltern, unterscheidet sich im Grundsatz nicht sehr stark von der Bestimmung der Ordnung einer Triangulation. Zu jeder Kante werden zwei Eingabepunkte bestimmt, die mit der Kante zwei Dreiecke ergeben. Dann wird analog zum vorigen Abschnitt mit Hilfe des Voronoi-Diagramms k-ter Ordnung berechnet, ob diese Dreiecke k-OD-Dreiecke sind. Zur Bestimmung dieser beiden Punkte zu einer k-OD-Kante $\overline{uv}$ muß allerdings noch etwas Vorarbeit geleistet werden, nämlich die Bestimmung der Delaunay-Kanten, die von $\overline{uv}$ geschnitten werden. Liegen beide Endpunkte von $\overline{uv}$ nicht in demselben Prozessorbereich, ist dieser Schritt mit zusätzlicher Kommunikation verbunden. Im grobkörnigen Parallelrechnermodell ohne gemeinsamen Speicher führt dies zu einem Algorithmus, dessen Laufzeit nur schwer nach oben abgeschätzt werden kann, was für das PRAM-Modell nicht gilt:

Algorithmus 4.19 *Parallele Bestimmung nützlicher k-OD-Kanten im PRAM-Modell*

Eingabe: Planare Punktmenge S mit n Punkten, Delaunay-Triangulation von S, alle k-OD-Kanten von S.

Ausgabe: Alle nützlichen k-OD-Kanten von S.

1. Preprocessing: Berechne parallel das Voronoi-Diagramm k-ter Ordnung von S.
2. FOR EACH Kante $\overline{uv}$, die lokal gespeichert ist PARDO
   (a) Lokalisiere die zugehörigen Punkte u und v in der Delaunay-Triangulation.
   (b) Traversiere von u nach v entlang $\overline{uv}$ von Delaunay-Dreieck zu Delaunay-Dreieck und speichere dabei alle geschnittenen Delaunay-Kanten. Falls dies mehr als 2k-1 oder auf einer Seite mehr als k sind, ist $\overline{uv}$ nicht nützlich.
   (c) Bestimme s_1 und s_2 wie in Lemma 1.24 definiert. Berechne, wie viele Punkte - höchstens k (nützlich) oder doch mehr (nicht nützlich) - in den Umkreisen $C(u, s_1, v)$ und $C(u, s_2, v)$ liegen. Benutze dafür das Voronoi-Diagramm k-ter Ordnung, um den k-test-nächsten Nachbarn zu finden und dessen Abstand zum Mittelpunkt des aktuellen Kreises mit dem Kreisradius zu vergleichen.

Erläuterungen und Analyse

Die Berechnung des gewöhnlichen Voronoi-Diagramms bei gegebener Delaunay-Triangulation ist in $O(1)$ Zeitschritten bei $O(n)$ Work möglich. Die Umwandlung des Diagramms von der Ordnung 1 zur Ordnung k erfordert eine Abwandlung des CGM-Algorithmus: In jeder Iteration wird jede Voronoi-Region gleichzeitig in eine Region der nächsthöheren Ordnung umgewandelt, wozu die Berechnung des Voronoi-Diagramms von $S \backslash H'$ nötig ist. Hierfür verwenden wir den parallelen Algorithmus von Amato et al. [4], der auf einer EREW-PRAM in $O(log^2 n)$ Zeitschritten und $O(n \log n)$ Work das Voronoi-Diagramm einer n-elementigen Punktmenge bestimmt.

Zur Ausführung des Algorithmus ist der gleichzeitige lesende Zugriff auf Eingabeobjekte erforderlich, ein gleichzeitiger schreibender Zugriff jedoch nicht, weshalb eine CREW-PRAM als

ausführendes Modell gewählt werden kann. Die Korrektheit des Algorithmus folgt aus der Korrektheit seiner seriellen Variante und der eingesetzten Verfahren zur Bestimmung des Voronoi-Diagramms k-ter Ordnung. Man kann schließen:

Satz 4.20 *Hat man alle k-OD-Kanten einer planaren Punktmenge S aus n Punkten und ihre Delaunay-Triangulation gegeben, lassen sich alle nützlichen k-OD-Kanten von S in $O(k \cdot log^2(kn))$ Zeitschritten bei $O(k^2 n\, log(kn))$ Gesamtoperationen auf einer CREW-PRAM berechnen.*

Beweis: Die Umwandlung des Voronoi-Diagramms der Ordnung i-1 in eines der Ordnung i erfordert mit dem o. a. Verfahren $O(log^2(kn))$ Zeit und $O(kn\, log(kn))$ Work pro Iteration, weil $O(k(n-k)) = O(kn)$ Voronoi-Regionen zu modifizieren sind und diese $O(kn)$ Kanten haben können. Multipliziert mit $O(k)$ Iterationen ergibt sich das Resultat des Satzes. Die übrigen Schritte werden hiervon dominiert: Das Lokalisieren der Punkte erfolgt in $O(log(n))$ Zeit bei $O(kn\, log(kn))$ Work, das Traversieren der Kanten benötigt $O(k)$ Zeit und $O(k^2 n)$ Work und der letzte Schritt erfordert schließlich $O(log\, n)$ Zeit und $O(kn\, log\, n)$ Work. □

In einer CGM-Variante dieses Verfahrens trifft man auf das Problem der Verteilung der k-OD-Kanten. Da die Delaunay-Triangulation dann nicht in einem gemeinsamen Speicher vorliegt, müßte jede Kante $\overline{uv}$ zu ihrer Traversierung an die Prozessoren versendet werden, deren lokaler Bereich der Delaunay-Triangulation zumindest teilweise von $\overline{uv}$ geschnitten wird. Es ist bisher nicht gelungen zu zeigen, daß kein Bereich während der Schleife mehr als $O(\frac{kn}{p})$ Punkte speichern muß. Diese Speichergrenze spielt bei gemeinsamem Speicher naturgemäß keine Rolle, sollte aber bei einem CGM-Algorithmus schon wegen der Lastverteilung eingehalten werden. Der Beweis einer nichttrivialen oberen Schranke bleibt Bestandteil weiterer Forschungen. Hierfür müßte man zeigen, daß sich die k-OD-Kanten gleichmäßig innerhalb der vorliegenden Prozessorteilbereiche der Delaunay-Triangulation verteilen.

4.5 Fazit

In diesem Kapitel ist ein CGM-Algorithmus für die Berechnung von Delaunay-Triangulationen erster Ordnung angegeben worden, mit dessen Hilfe sich mehrere Eigenschaften von Netzen optimieren lassen. Weiterhin wurde der erste dem Autor bekannte parallele Algorithmus zur Berechnung von Voronoi-Diagrammen höherer Ordnung entworfen. Die zunächst ausführlich beschriebene CGM-Variante dieses Verfahrens wurde im letzten Teilkapitel außerdem in einen Algorithmus für eine CREW-PRAM umgesetzt. Im Bereich der Anwendungen im Kontext höherer Ordnungen konnte die Bestimmung der Ordnung einer Triangulation effizient für das CGM-Modell parallelisiert werden und der sequentielle Algorithmus gegenüber der Originalarbeit für kleine Werte k verbessert werden.

Im letzten Abschnitt bleibt bei der Bestimmung aller nützlichen k-OD-Kanten die Frage nach einer konkreten oberen Schranke für die Laufzeit eines CGM-Algorithmus offen. Während sich für eine CREW-PRAM eine effiziente Parallelisierung angeben läßt, bleibt die Umsetzung ins CGM-Modell Thema weiterer Forschungsarbeit.

Kapitel 5

Paralleles Preprocessing zur Visualisierung von TINs

Für die Darstellung eines TINs am Bildschirm möchte man die vorher bei der parallelen Berechnung und verteilten Speicherung erzielten Vorteile der Parallelverarbeitung natürlich nicht aufgeben. Dieser Abschnitt soll daher zeigen, wie man das in der Computergrafik bekannte Verfahren der Tiefensortierung (engl.: *depth order*) durch Parallelverarbeitung beschleunigen kann.

5.1 Grundlagen

Zur Darstellung eines zweieinhalbdimensionalen TINs auf einem üblichen zweidimensionalen Bildschirm müssen die darzustellenden Objekte aus dem Raum auf die Sichtebene projiziert werden. Es kommt dabei vor, daß Objekte im Raum andere Objekte verdecken, weil diese vom Standpunkt des Betrachters aus gesehen vor den verdeckten Objekten liegen. Bei der Visualisierung muß dies berücksichtigt werden, weil sonst kein realistisches Bild der dargestellten Szene entsteht.

Ein wichtiger Algorithmus zum Entfernen verdeckter Oberflächen (engl.: *hidden-surface removal*) bedient sich der Tiefensortierung. Dabei wird jedem Objekt ein Tiefenwert zugeordnet, anhand dessen bestimmt wird, in welcher Reihenfolge die Objekte gezeichnet werden müssen, um ein korrektes Ergebnis zu erhalten. Je weiter hinten vom Betrachter aus gesehen die Objekte liegen, desto früher werden sie auf den Bildschirm gebracht. Da dieses Verfahren der Arbeitsweise eines Malers ähnelt, wird es *Maleralgorithmus* genannt. Es gibt zwar den Fall der zyklischen Überlappung, in dem eine Tiefensortierung nicht möglich ist und der Maleralgorithmus ohne zusätzliche Techniken versagt, solch einen Fall kann man aber bei der Darstellung von Dreiecksnetzen recht einfach ausschließen (s. de Berg [12, S. 84]).

Das Verfahren zur Bestimmung einer Tiefenreihenfolge der Objekte bei gegebener Blickrichtung des Betrachters beruht auf der Erkenntnis, daß ein Strahl im Raum zwei Dreiecke in genau der Reihenfolge schneidet wie der in die x-y-Ebene projizierte Strahl die ebenfalls projizierten Dreiecke schneidet, falls die geschnittenen Dreiecke sich nicht überlappen (vgl. de Berg [12, S. 84]). Weil diese Bedingung bei Dreiecksnetzen erfüllt ist, genügt es, die Tiefenreihenfolge für die

Projektion der Dreiecke und des Strahls der Blickrichtung zu bestimmen.

Definition 5.1 *Seien d die in die x-y-Ebene projizierte Blickrichtung des Betrachters und t und t' in die x-y-Ebene projizierte Dreiecke eines Dreiecksnetzes T. Dann liegt Dreieck t vor Dreieck t', wenn es eine Gerade in der Ebene mit Richtung d gibt, die erst t und dann t' schneidet.*

5.2 Algorithmus

Die Tiefenreihenfolge für die einzelnen Dreiecke eines anhand von vertikalen Grenzpfaden verteilt gespeicherten TINs läßt sich ohne viel Kommunikationsaufwand parallel bestimmen, wenn die Blickrichtung des Betrachters winkelmäßig nicht sehr stark von der horizontalen Achse abweicht. Wenn nämlich die Blickrichtung nicht steiler als die steilste Kante auf allen Grenzpfaden zwischen zwei Prozessoren ist, liegen die Dreiecke eines Prozessors alle vor den Dreiecken seines rechten Nachbarn.

Algorithmus 5.2 *Parallele Bestimmung der Tiefenreihenfolge(T, d)*

Eingabe: CGM(n,p), TIN T mit n Stützstellen als $T = \bigcup_{0 \leq j \leq p-1} T_j$ verteilt auf p Prozessoren gespeichert, die Aufteilung erfolgt durch vertikale Delaunay-Pfade an den Prozessorgrenzen; Blickrichtung d.

Ausgabe: Verteilt gespeicherte Tiefensortierung der Dreiecke von T anhand von Prioritätsnummern für jedes Dreieck.

1. Jeder Prozessor (außer dem äußerst linken) berechnet für jede Kante seines linken Grenzpfades ihren Winkel zur x-Achse. Ist das Maximum dieser Winkel auf einem Prozessor größer als d, wird die Eingabe zurückgewiesen und dieses Maximum ausgegeben.

2. Lokal wird auf Prozessor p_j der duale Graph $G_j = (V_j, E_j)$ von T_j konstruiert. G ist zunächst ungerichtet, wird aber hinsichtlich der "liegt vor"-Relation folgendermaßen gerichtet: Eine Kante $\overline{ab}$ wird zu $\vec{ab}$ gerichtet, falls Dreieck a vor Dreieck b liegt, sie wird zu $\vec{ba}$ gerichtet, falls b vor a liegt und gelöscht, falls keiner der beiden Fälle gilt.

3. Jeder Prozessor führt lokal auf G_j eine topologische Sortierung durch und vergibt für jedes Dreieck eine Prioritätsnummer.

4. Global wird die parallele Präfix-Summe über die Anzahl der auf den Prozessoren gespeicherten Dreiecke berechnet.

5. Die Prioritätsnummern der Dreiecke werden um das für den eigenen Prozessor geltende Ergebnis der Präfix-Summe erhöht.

Erläuterungen und Analyse

Die letzten beiden Schritte können entfallen, wenn jeder Prozessor die Aufgabe übernimmt, seine lokalen Daten an die Bildschirmausgabe zu übermitteln. Es genügt dann, jedes Dreieck neben der Priorität mit der zugehörigen Prozessornummer zu behaften und sie zunächst in der Reihenfolge der Prozessoren und erst dann der (lokalen) Priorität auf den Bildschirm zu bringen.

Das Berechnen der Steigung der Grenzpfadkanten ist in Linearzeit bezüglich der Pfadlänge zu bewältigen. Die eventuelle Rückgabe des zulässigen Maximalwertes für d erfordert bei einer Kommunikationsrunde $O(p)$ lokale Berechnungsschritte. Ist das TIN in einer geeigneten Datenstruktur (etwa in einer doppelt verketteten Kantenliste) gespeichert, so ist der duale Graph sehr einfach zu konstruieren. Man benötigt dafür nur $O(\frac{n}{p})$ Schritte, da nach Folgerung 2.2 das gesamte TIN höchstens $2n - 4$ Dreiecke (und G somit Knoten) hat. Das Richten der Kanten ist ebenfalls in Linearzeit möglich. Die topologische Sortierung hat eine Zeitkomplexität von $O(|V_j| + |E_j|) = O(\frac{n}{p})$. Die globale Präfix-Summe kann durch das Versenden und Addieren von $O(p)$ Daten bei einer Kommunikationsrunde in $O(p)$ Zeitschritten erfolgen.[21] Die Aktualisierung der Prioritäten benötigt dann noch $O(\frac{n}{p})$ lokale Schritte. Es folgt nun direkt:

Satz 5.3 *Der Algorithmus zur Bestimmung der Tiefenreihenfolge in einem TIN bei zulässiger Blickrichtung d benötigt auf einem grobkörnigen Multicomputer CGM(n, p) mit* $\frac{n}{p} \geq p$ $O(1)$ *Kommunikationsrunden und hat eine optimale Berechnungskomplexität von* $O(\frac{n}{p})$.

5.3 Fazit

In diesem Kapitel ist ein optimaler CGM-Algorithmus angegeben worden, mit dessen Hilfe eine Tiefensortierung eines TINs bei gegebener Blickrichtung berechnet werden kann. An diese Blickrichtung wird eine Vorbedingung gestellt, sie muß "möglichst waagrecht" sein, um jeden vertikalen Delaunay-Pfad genau einmal zu durchstoßen. Diese Einschränkung könnte man aufheben, wenn es einen grobkörnigen parallelen Algorithmus zur Berechnung der topologischen Sortierung eines Graphen gäbe.

Für das PRAM-Modell gibt es ein solches Verfahren von Thomas H. Spencer [65], das mit Hilfe spezieller Listen und der transitiven Hülle eines Graphen dessen topologische Sortierung berechnet. Unter den hier gegebenen Bedingungen benötigt dieser Algorithmus auf einer EREW-PRAM eine Ausführungszeit von $O(\frac{n}{p}\, log^2 p)$ und $O(np^2 log\, p)$ Operationen. Zur Lösung unseres Problems der Tiefensortierung bei beliebiger Blickrichtung müßte also nur der duale Graph von T anhand der "liegt-vor"-Relation gebildet und für ihn mit diesem PRAM-Algorithmus eine topologische Sortierung berechnet werden. Wir sind allerdings an einer grobkörnigen Lösung interessiert, deren Entwicklung das Ziel zukünftiger Forschungsarbeit ist.

[21] Hier wird im Hinblick auf ein grobkörniges Modell auf das Binärbaumparadigma zur Berechnung der Präfix-Summe verzichtet, um Kommunikation zu sparen.

Schlußbemerkungen

In dieser Arbeit konnte gezeigt werden, daß sich das Konzept der Delaunay-Triangulation höherer Ordnung und verwandte Verfahren in wesentlichen Teilen durch Parallelverarbeitung beschleunigt werden können. Als geeignetes paralleles Rechenmodell ist hierfür nach dem Vergleich dreier Modelle das des grobkörnigen Multicomputers (CGM) gewählt worden, weil es realitätsnah und vergleichsweise einfach bei Entwurf und Analyse ist. Die Algorithmen, von denen eine effiziente Parallelisierung entworfen worden ist, konstruieren die gewöhnliche Delaunay-Triangulation und jene erster Ordnung sowie Voronoi-Diagramme höherer Ordnung und bestimmen die Ordnung einer Triangulation sowie eine Tiefensortierung von geeigneten Dreiecksnetzen. Von diesen Verfahren sind besonders die Algorithmen für die gewöhnliche Delaunay-Triangulation und die Voronoi-Diagramme höherer Ordnung hervorzuheben, da sie eine nichttriviale Verbesserung bzw. eine vollständige Neuerung darstellen und die Grundlage für die anderen Methoden bilden.

Eine Gelegenheit für weitere Forschungen bieten die drei folgenden theoretischen Probleme: erstens ein Beweis, für welche konkreten Punktverteilungen sich effizient bzw. nicht effizient mit dem entworfenen Algorithmus die Delaunay-Triangulation berechnen läßt, zum zweiten ein Beweis, daß sich die k-OD-Kanten beim Nützlichkeitstest günstig verteilen (lassen) und schließlich ein grobkörniger Algorithmus, der für jede Eingabe eine topologische Sortierung eines gerichteten und azyklischen Graphen berechnen kann. Zudem kann man in weiteren Arbeiten die praktische Performance des theoretisch verbesserten Delaunay-Algorithmus untersuchen und mit den für diese Arbeit implementierten und auf zwei Clusterrechnern getesteten und evaluierten Varianten vergleichen.

Literaturverzeichnis

[1] Agarwal, P. K.; de Berg, M; Matousek, J.; Schwarzkopf, O.: Constructing Levels in Arrangements and Higher Order Voronoi Diagrams. In: Proc. 10th Symp. on Computational Geometry (1994), S. 67-75.

[2] Aggarwal, A.; Guibas, L. J.; Saxe, J.; Shor, P. W.: A Linear-Time Algorithm for Computing the Voronoi Diagram of a Convex Polygon. In: Discrete & Comput. Geometry **4** (1989), S. 591-604.

[3] Akl, S.; Lyons, K.: Parallel Computational Geometry. Prentice Hall, 1993.

[4] Amato, N. M.; Goodrich, M. T., Ramos, E. A.: Parallel algorithms for higher-dimensional convex hulls. In: Proc. 35th Annual IEEE Symp. on Foundations of Computer Science (1994), S. 683-694.

[5] Aurenhammer, F.: A New Duality Result Concerning Voronoi Diagrams. In: Discrete & Computational Geometry **5** (1990), S. 243-254.

[6] Aurenhammer, F.: Voronoi Diagrams - A Survey of a Fundamental Geometric Data Structure. ACM Comput. Surv. **23** (1991), Nr. 3, S. 345-405.

[7] Aurenhammer, F.; Klein, R.: Voronoi Diagrams. In: Handbook of Computational Geometry. Hg. von J.-R. Sack u. J. Urrutia. Elsevier Science, 1999. S. 201-290.

[8] Aurenhammer, F.; Schwarzkopf, O.: A Simple On-Line Randomized Incremental Algorithm for Computing Higher Order Voronoi Diagrams. In: Proc. 7th Symp. on Computational Geometry (1991), S. 142-151.

[9] Aurenhammer, F.; Xu, Y.-F.: Optimal triangulations. In: Encyclopedia of Optimization, Bd. 4. Hg. von P. M. Pardalos u. C. A. Floudas. Kluwer Academic Publishing, 2000. S. 160-166.

[10] Bäumker, A.; Dittrich, W.; Meyer auf der Heide, F.: Truly Efficient Parallel Algorithms: c-Optimal Multisearch for an Extension of the BSP Model. In: Proc. 3rd ESA (1995), S. 17-30.

[11] Beowulf.org: Frequently Asked Questions. Webseite abrufbar unter http://www.beowulf.org/overview/faq.html, Stand: 2004, Abruf: 27.08.2004.

[12] Berg, M. de: Visualization of TINs. In: Algorithmic Foundations of Geographic Information Systems. Lecture Notes in Computer Science 1340. Hg. von M. J. van Kreveld, J. Nievergelt, T. Roos u. P. Widmayer. Springer-Verlag, 1997. S. 79-97.

[13] Berg, M. de; Kreveld, M. J. van; Overmars, M.; Schwarzkopf, O.: Computational Geometry. Algorithms and Applications. 2. Aufl. Springer-Verlag, 2000.

[14] Bern M.; Eppstein, D.: Mesh Generation And Optimal Triangulation. In: Computing in Euclidean Geometry. 2. Aufl. Hg. von D.-Z. Du u. F. Hwang. World Scientific, 1995. S. 47-123.

[15] Blelloch, G. E.; Hardwick, J. C.; Miller, G. L.; Talmor, D.: Design and Implementation of a Practical Parallel Delaunay Algorithm. In: Algorithmica **24** (1999), S. 243-269.

[16] Boissonnat, J.-D.; Devillers, O.; Teillaud, M.: A Semi-Dynamic Construction of Higher Order Voronoi Diagrams and its Randomized Analysis. In: Algorithmica **9** (1993), S. 329-356.

[17] Brouns, G.; Wulf, A. de; Constales, D.: Multibean data processing: Adding and deleting vertices in a Delaunay triangulation. In: The Hydrographic Journal **101** (2001), S. 3-9.

[18] Chan, T. M.: Output-sensitive results on convex hulls, extreme points, and related problems. In: Proc. 11th Symp. on Computational Geometry (1995), S. 10-19.

[19] Chan, T. M.: Random Sampling, Halfspace Range Reporting, and Construction of (<= k)-Levels in Three Dimensions. SIAM J. Comput. **30** (2000), Nr. 2, S. 561-575.

[20] Chazelle, B.; Edelsbrunner, H.: An Improved Algorithm for Constructing *k*th-Order Voronoi Diagrams. In: IEEE Transactions on Computers **C-36** (1987), Nr. 11, S. 1349-1354.

[21] Chin, F.; Wang, C. A.: Finding the Constrained Delaunay Triangulation and Constrained Voronoi Diagram of a Simple Polygon in Linear Time. SIAM J. Comput. **28** (1998), Nr. 2, S. 471-486.

[22] Cignoni, P.; Montani, C.; Perego, R.; Scopigno, R.: Parallel 3D Delaunay Triangulation. Comput. Graph. Forum **12** (1993), Nr. 3, S. 129-142.

[23] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.: Introduction to Algorithms. MIT Press, 1990.

[24] Dehne, F.; Deng, X.; Dymond, P.; Fabri, A.; Khokhar, A. A.: A Randomized Parallel Three-Dimensional Convex Hull Algorithm for Coarse-Grained Multicomputers. In: Theory of Computing Systems **30** (1997), Nr. 6, S. 547-558.

[25] Dehne, F.; Fabri, A.; Rau-Chaplin, A.: Scalable parallel geometric algorithms for coarse grained multicomputers. In: Proc. 9th Symp. on Computational geometry (1993), S. 298-307.

[26] Dehne, F.; Fabri, A.; Rau-Chaplin, A.: Scalable parallel computational geometry for coarse grained multicomputers. In: Int. J. Comput. Geometry **6** (1996), Nr. 3, S. 379-400.

[27] Delaunay, B.: Sur la sphre vide. A la memoire de Georges Voronoi. In: Izv. Akad. Nauk SSSR, Otdelenie Matematicheskih i Estestvennyh Nauk 7 (1934), S. 793-800. Literaturverweis nach Aurenhammer und Klein [7].

[28] Diallo, M.; Ferreira, A.; Rau-Chaplin, A.: A Note On Communication-Efficient Deterministic Parallel Algorithms for Planar Point Location and 2d Voronoi Diagram. In: Parallel Processing Letters **11** (2001), Nr. 2-3, S. 327-340.

[29] Diallo, M.; Ferreira, A.; Rau-Chaplin, A.; Ubéda, S.: Scalable 2D Convex Hull and Triangulation Algorithms for Coarse Grained Multicomputers. In: J. Parallel Distrib. Comput. **56** (1999), Nr. 1, S. 47-70.

[30] Dwyer, R. A.: A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations. In: Algorithmica **2** (1987), S. 137-151.

[31] Estivill-Castro, V.; Lee, I.: AMOEBA: Hierarchical Clustering Based on Spatial Proximity Using Delaunay Triangulation. Techn. Bericht 99-05 (1999), Dep. of Computer Science and Software Engineering, The University of Newcastle. Erhältlich unter ftp://ftp.cs.newcastle.edu.au/pub/techreports/tr99-05.ps.Z, Abruf: 11.08.2004.

[32] Fortune, S.: A Sweepline Algorithm for Voronoi Diagrams. Algorithmica **2** (1987), S. 153-174.

[33] Fortune, S.: Voronoi Diagrams and Delaunay Triangulations. In: Euclidean Geometry and Computers. Hg. von D. A. Du u. F. K. Hwang. World Scientific Publishing, 1992. S. 193-233.

[34] Fortune, S.: Voronoi Diagrams and Delaunay Triangulations. In: Handbook of Discrete and Computational Geometry. Hg. von J. O'Rourke u. J. E. Goodman. CRC Press, 1997. S. 377-388.

[35] Gemma Frisius, R.: Libellus de locorum describendorum ratione, [et] de eorum distantijs inveniendis. In: Cosmographicus liber Petri Apiani. Antwerpen, 1533.

[36] Gerbessiotis, A. V.; Valiant, L. G.: Direct-Bulk-Synchronous Parallel Algorithms. In: J. of Parallel and Distrib. Computing **22** (1994), Nr. 2, S. 251-267.

[37] Goodrich, M. T.: Communication-Efficient Parallel Sorting. In: SIAM J. on Computing **29** (1999), Nr. 2, S. 416-432.

[38] Graham, R. L.: An Efficient Algorithm For Determining the Convex Hull of a Planar Set. In: Information Processing Letters **1** (1972), Nr. 4, S. 132-133.

[39] Gudmundsson, J.; Hammar, M.; Kreveld, M. J. van: Higher Order Delaunay Triangulations. In: Computational Geometry - Theory and Appl. **23** (2002), Nr. 1, S. 85-98.

[40] Gudmundsson, J.; Haverkort, H.; Kreveld, M. J. van: Constrained Higher Order Delaunay Triangulations. In: Proc. 19th Europ. Workshop on Comp. Geom. (2003), S. 105-108.

[41] Guibas, L.; Stolfi, J.: Primitives for the manipulation of general subdivisions and the computation of Voronoi Diagrams. In: ACM Transactions on Graphics **4** (1985), Nr. 2, S. 74-123.

[42] Hambrusch, S. E.: Models for Parallel Computation. In: Proc. ICPP Workshop on Challenges for Parallel Processing (1996), S. 92-95.

[43] Hill, J. M. D.; Donaldson, S. R.; Skillicorn, D. B.: Portability of performance with the BSPlib communications library. In: Proc. Massively Parallel Programming Models (1997), S. 33-42.

[44] JaJa, J.: Introduction to Parallel Algorithms. Addison-Wesley, 1992.

[45] Jungnickel, D.: Graphen, Netzwerke und Algorithmen. 3. Aufl. B. I. Wissenschaftsverlag, 1994.

[46] Juurlink, B. H. H.; Wijshoff, H. A. G.: Communication primitives for BSP computers. In: Information Processing Letters **58** (1996), S. 303-310.

[47] Klein, R.: Algorithmische Geometrie. Addison-Wesley-Longman, 1997.

[48] Kreveld, M. J. van: Digital Elevation Models and TIN Algorithms. In: Algorithmic Foundations of Geographic Information Systems. Lecture Notes in Computer Science 1340. Hg. von M. J. van Kreveld, J. Nievergelt, T. Roos u. P. Widmayer. Springer-Verlag, 1997. S. 37-78.

[49] Kühn, U.: Lokale Eigenschaften in der algorithmischen Geometrie mit Anwendungen in der Parallelverarbeitung. Dissertation Westfälische Wilhelms-Universität Münster. 1998. 116 S.

[50] Lanthier, M.; Maheshwari, A.; Sack, J.-R.: Approximating Shortest Paths on Weighted Polyhedral Surfaces. In: Algorithmica **30** (2001), Nr. 4, S. 527-562.

[51] Lee, D.T.: On k-nearest neighbor Voronoi diagrams in the plane. In: IEEE Transactions on Computers **31** (1982), Nr. 6, S. 478-487.

[52] Lee, S.; Park, C.-I.; Park, C.-M.: An Improved Parallel Algorithm for Delaunay Triangulation on Distributed Memory Parallel Computers. In: Parallel Processing Letters **11** (2001), Nr. 2/3, S. 341-352.

[53] Mehlhorn, K.; Näher, S.: LEDA. A platform for combinatorial and geometric computing. Cambridge University Press, 1999.

[54] Moore, G. E.: Cramming More Components Onto Integrated Circuits. In: Electronica **38** (1965), Nr. 8.

[55] Mulmuley, K.; Schwarzkopf, O.: Randomized Algorithms. In: Handbook of Discrete and Computational Geometry. Hg. von J. O'Rourke u. J. E. Goodman. CRC Press, 1997. S. 633-652.

[56] Okabe, A.; Boots, B.; Sugihara, K.: Spatial tesselations. Concepts and applications of Voronoi diagrams. Wiley, 1992.

[57] O'Rourke, J.: Computational Geometry in C. Cambridge University Press, 1994.

[58] Preparata, F.; Hong, S.: Convex hulls of finite sets of points in two and three dimensions. In: Commun. ACM **20** (1977), S. 87-93.

[59] Preparata, F. P.; Shamos, M. I.: Computational Geometry - An Introduction. 2. Aufl., Springer-Verlag, 1988.

[60] Ramos, E. A.: On range reporting, ray shooting and k-level construction. In: Proc. 15th Symp. on Computational Geometry (1999), S. 390-399.

[61] Schaudt, B.: Higher Order Voronoi Diagrams: A Java Applet. Webseite abrufbar unter http://www.msi.umn.edu/~schaudt/voronoi/voronoi.html, Stand: April 1998, Abruf: 28.07.2004.

[62] Seidel, R.: Convex hull computations. In: Handbook of Discrete and Computational Geometry. Hg. von J. O'Rourke u. J. E. Goodman. CRC Press, 1997. S. 361-375.

[63] Shi, H.; Schaeffer, J.: Parallel sorting by regular sampling. In: Journal of Parallel and Distributed Computing **14** (1992), Nr. 4, S. 361-372.

[64] Snir, M.; Otto, S.; Huss-Lederman, S.; Walker, D.; Dongarra, J.: MPI: The Complete Reference. Bd. 1, 2. Aufl. MIT Press, 1998.

[65] Spencer, T. H.: Time-Work Tradeoffs for Parallel Algorithms. In: Journal of the ACM **44** (1997), Nr. 5, S. 742-778.

[66] Su, P.; Drysdale, R. L. S.: A Comparison of Sequential Delaunay Triangulation Algorithms. In: Proc. 11th Annual Symp. on Computational Geometry (1995), S. 61-70.

[67] Valiant, L. G.: General Purpose Parallel Architectures. In: Handbook of Theoretical Computer Science, Bd. A: Algorithms and Complexity. Hg. von J. van Leeuwen. Elsevier and MIT Press, 1990. S. 943-972.

[68] Valiant, L. G.: A Bridging Model for Parallel Computation. In: Communications of the ACM **33** (1990), Nr. 8, S.103-111.

[69] Weiss, M. A.: Data Structures and Algorithm Analysis in C++. 2. Aufl. Addison-Wesley, 1999.

Abbildungsverzeichnis

1.1 Planare Punktmenge und ihre konvexe Hülle 7
1.2 Delaunay-Triangulation und Voronoi-Diagramm einer planaren Punktmenge . . . 9
1.3 Konvexes Viereck mit zwei möglichen Diagonalen 10
1.4 Künstlicher Damm in einem TIN (links) und seine Auflösung durch einen Kantenwechsel (rechts) 12
1.5 $C(u, v, s)$ enthält p 13
1.6 Hülle einer k-OD-Kante 14
1.7 Wenn $\triangle us_1v$ kein k-OD-Dreieck ist, ist $\overline{uv}$ nicht nützlich. 15
1.8 Jede nützliche 1-OD-Kante schneidet höchstens eine nützliche 1-OD-Kante 17
1.9 Schema einer PRAM 21
1.10 Schema eines Baum-Broadcasts 25

2.1 Paraboloid der Form $z = x^2 + y^2$ 34

3.1 Prinzip der Eingabeaufteilung anhand eines Bearbeitungsbaumes 52
3.2 Jeweilige Grenzprozessoren der Baumknoten 52
3.3 Graphische Ausgabe der Lösung bei 8 Prozessoren und insgesamt 512 Eingabepunkten 62
3.4 Speedupwerte auf dem Beowulf-Cluster 63
3.5 Speedupwerte auf dem *hpcLine*-Cluster 64
3.6 *Bottom-up*- vor *Top-down*-Durchlauf im Bearbeitungsbaum 65

4.1 Berechnung des Anfragepunktes q zur Kante e 73
4.2 Voronoi-Diagramme der Ordnungen 1 und 2 76
4.3 Voronoi-Diagramme der Ordnungen 3 und 4 76

Selbständigkeitserklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, 31. August 2004 Henning Meyerhenke