Vergleich parallelisierter Fast Marching Algorithmen

Moritz Christopher Kappes

Geboren am 24. August 1998 in Brühl

30. September 2021

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Alexander Effland

Zweitgutachter: Dr. Thomas Pinetz

Institut für Angewandte Mathematik

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Inhaltsverzeichnis

Einleitung

Man stelle sich eine Grenze zwischen zwei Regionen vor. Im Zweidimensionalen eine Kurve, in drei Dimensionen eine Oberfläche. Diese Grenze bewegt sich nun in Richtung der äußeren Normale der Kurve/Oberfläche mit einer bekannten Geschwindigkeitsfunktion F.
Das Ziel ist es, die Bewegung dieser Grenzfläche über die Zeit zu verfolgen.
Da die Grenzfläche über Zeit problematische Stellen entwickeln kann, wird eine geeignete schwache Lösung für das oben beschriebene Problem benötigt. Als Beispiel kann man eine Cosinus-Kurve betrachten, die sich mit Geschwindigkeit $F = 1$ in Richtung der Normalen, nach oben, bewegt. Diese entwickelt in dem „Tal“ über die Zeit einen sogenannten Schwalbenschwanz und die Grenzfläche ist nicht mehr klar als eine solche zu erkennen. Vor allem ist sie keine wohldefinierte Funktion mehr.

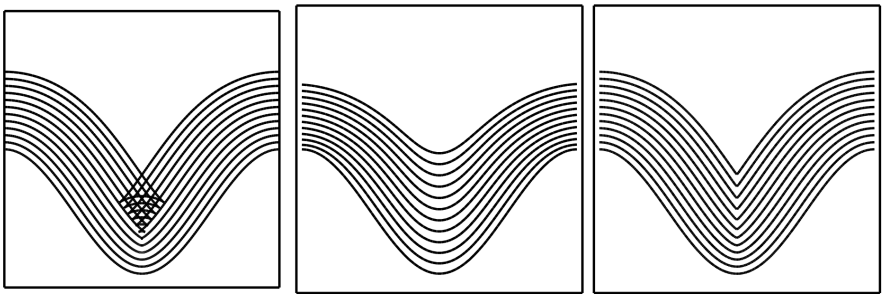


Abbildung 0.1.: Links: Mit $F = 1$ entwickelt sich die Grenzfläche fehlerhaft
Mitte, rechts: Der Übergang zur und die erwünschte Lösung
Grafik entnommen aus [1]

Diese Art der schwachen Lösung wird im Theoriekapitel sukzessiv mathematisch eingeführt und definiert.
Im Folgenden wird der Einfachheit halber der zweidimensionale Fall betrachtet. Die Anschauung überträgt sich direkt auf den dreidimensionalen Fall.
Die Grenzfläche wird Γ genannt, wobei Γ eine abgeschlossene Kurve in $\mathbb{R}^2$ ist. In den sogenannten Niveaumengenmethoden wird Γ als Niveaumenge von 0 einer unbekannten Funktion ϕ betrachtet. Demnach ist $\phi(x, t = 0) = \pm d_x$ wobei d_x der Abstand von einem Punkt x zu Γ ist. und Γ_0 die gegebene Startposition von Γ Das Vorzeichen ist + oder - je nachdem, ob x außer- oder innerhalb von Γ liegt.
Mit der Kettenregel kann man eine Evolutionsgleichung für die Grenzfläche Γ erzeugen.

$$\begin{cases} \phi_t + F|\nabla\phi| = 0 \\ \phi(x, t = 0) = \Gamma_0 \end{cases} \tag{0.1}$$

Dies ist eine partielle Differentialgleichung in einer um eins höheren Dimension als das Anfangsproblem und wird *Level-Set-Gleichung* genannt.

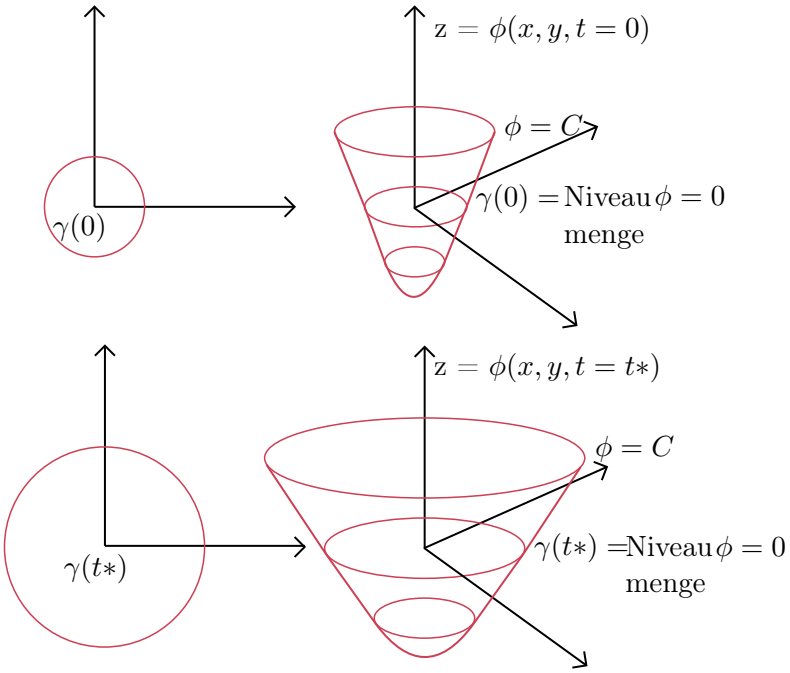


Abbildung 0.2.: Beispiel eines sich mit $F = 1$ ausbreitenden Kreises Γ

Für dieses Problem gibt es bereits numerische Algorithmen, allerdings sind diese Aufgrund der höheren Dimension sehr rechenaufwändig.
Eine starke Reduzierung des Rechenaufwandes erreicht man in dem Spezialfall, in dem $F = F(x, y, z)$ entweder strikt positive oder strikt negative Werte annimmt. Dann breitet sich die Grenzfläche monoton nach außen oder innen aus. Nun wird die Entwicklung der Grenzfläche Γ, also der Niveaumenge von 0 zum Zeitpunkt $t = 0$, über die Zeit betrachtet.
Sei $T(x, y)$ der Zeitpunkt, in dem die sich ausbreitende Grenzfläche den Punkt (x, y) das erste Mal berührt. Die Oberfläche erfüllt nun:

$$|\nabla T|F = 1 \tag{0.2}$$

Die obige Gleichung ist ein Spezialfall einer Eikonalgleichung. Anschaulich bedeutet das, dass die Ankunftszeit invers proportional zur Geschwindigkeit der Grenzfläche ist. Die erneute Umwandlung in ein stationäres Problem und die Positivität der Geschwindigkeitsfunktion ermöglichen es einen effizienten

Algorithmus für die Lösung des oben beschriebenen Problems zu finden, die Fast-Marching-Methode.
Anwendungen von Eikonalgleichungen finden sich unter anderem in der computerunterstützten Bildanalyse, geometrischer Optik und stetigen kürzeste-Pfade Problemen.
Da in diesen Anwendungsbereichen meistens sehr große und komplexe Problemstellungen behandelt werden, sind parallele Algorithmen gefordert, um eine annehmbare Berechnungszeit zu erreichen. Für die Fast-Marching-Methode gab es lange keinen parallelen Algorithmus. Zusätzlich wurden für die Lösung der Eikonalgleichung andere Algorithmen entwickelt und parallelisiert. Außerdem sind wenige parallele Löser öffentlich verfügbar, weshalb das Programmieren eines parallelen Programms von hoher Bedeutung ist.

Aufbau der Arbeit

Die Arbeit ist in vier Kapitel unterteilt. In Kapitel 1 werden zuerst die theorethischen Grundlagen für den Algorithmus aufgearbeitet. Insbesondere wird auf Differentialgleichungen, schwache Lösungen und numerische Diskretisierung eingegangen. Zusätzlich wird der Algorithmus vorgestellt und auf die Frage der Parallelisierung eingegangen.
In Kapitel 2 wird die Implementierung des sequentiellen Algorithmus erläutert. Zusätzlich wird das entwickelte Programm auf Speicherverbrauch und Laufzeit analysiert.
In Kapitel 3 wird auf Details der Implementierung des parallelen Algorithmus eingegangen. Außerdem wird das Programm in verschiedenen Fällen auf Geschwindigkeit und parallele Effizienz getestet.
In Kapitel 4 befindet sich eine kurze Zusammenfassung der Arbeit und ein Fazit.

Eigenbeitrag

In dieser Arbeit befinden sich folgende wesentliche Beiträge:

- Sammlung und Zusammenfassung der relevanten Theorie
- Implementierung und sukzessive Verbesserung der sequentiellen Fast-Marching-Methode
- Implementierung des parallelen Algorithmus [2]
- Ausführliches Testen und Auswertung des parallelen Programms

Danksagung

Zuallererst möchte ich mich bei Prof. Dr. Alexander Effland für dieses interessante Thema und die Betreuung bedanken. Auch bedanke ich mich für die Zurverfügungstellung des Servers für das Messen der Laufzeiten. Außerdem danke ich Dr. Thomas Pinetz für die Übernahme der Zweitkorrektur. Nicht zuletzt gilt mein herzlicher Dank meinen Freunden und meiner Familie.

1. Mathematischer Hintergrund

In diesem Kapitel werden zunächst die mathematischen Grundlagen betrachtet. Dabei wird auf die Theorie der Differentialgleichungen eingegangen. Insbesondere wird eine Art der schwachen Lösung, die sogenannte Viskositätslösung, ausführlich theoretisch motiviert. Für bestimmte Gleichungssysteme wird eine explizite Darstellung der Lösung erarbeitet. Dann wird die Verbindung zur numerischen Diskretisierung erläutert und letztere vorgestellt. Danach wird die Fast-Marching-Methode detailliert besprochen. Im letzten Abschnitt werden allgemeine Probleme der Parallelisierung, und deren Lösungsmöglichkeiten behandelt. Die behandelte Theorie basiert zum Großteil auf [3] und alle Beweise finden sich dort. Außerdem wird für manche alternativen Ideen [4] benutzt, wo sich auch eine ausführlichere Besprechung der mathematischen Grundlagen findet.

1.1. Differentialgleichungen und deren Lösungen

- Eine partielle Differentialgleichung (PDG) der Ordnung $k \geq 1$, ist eine Gleichung einer unbekannten Funktion u und ihrer Ableitungen. In diesem Kapitel wird sich auf Differentialgleichungen Ordnung $k \leq 2$ beschränkt.

$$F(x, u, Du, D^2u) = 0, \qquad x \in \Omega \subset \mathbb{R}^n$$
$$F : \Omega \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^{n \times n} \to \mathbb{R}$$

  Ω ist dabei meistens eine offene Menge mit angemessener Randbedingung. Im Folgenden wird die mit [3] kohärente Notation $F(x, u, Du, D^2u) = F(x, z, p, M)$ benutzt.

- Eine PDG der Ordnung k heißt *linear*, falls sie die Form

$$Lu = f(x)$$

  hat, wobei L ein linearer Differentialoperator der Ordnung k ist.

Bekannte PDG sind zum Beispiel:

1. Die *Laplace Gleichung*
   $\Delta u = 0$
2. Die *Wärmeleitungsgleichung*
   $u_t - \Delta u = 0$
3. Die *Eikonal Gleichung*
   $|Du| - f(x) = 0$
4. Die *Hamilton-Jacobi-Gleichung*
   $u_t + H(x, u, Du) = 0, \mathbb{R}^n \times (0, \infty)$

In diesem Abschnitt geht es vor allem um die letzte Gleichung. Die sogenannte Hamilton-Funktion $H : \Omega \times \mathbb{R} \times \mathbb{R}^n$ ist stetig und oft konvex in p (Also dem Term der ersten Ableitung). Insbesondere sieht man, dass die in der Einführung erwähnte Level-Set Gleichung (0.1) eine Hamilton-Jacobi-Gleichung mit Hamilton-Funktion $H(x, p) = -\frac{1}{f(x)}|p|$ ist. Sie ist außerdem konvex in p, dem Gradiententeil.

1.1.1. Klassische und schwache Lösungen

Für Differentialgleichungen gibt es hauptsächlich zwei Formulierungen.
I *Dirichlet- (Randwert-) Probleme*

$$\begin{cases} F(x, u, Du, D^2u) = 0 & x \in \Omega \\ u(x) = g(x) & x \in \partial\Omega \end{cases} \tag{1.1}$$

wobei $\Omega \subset \mathbb{R}^n$ offen und beschränkt ist und g die gegebene, stetige Randwertbedingung ist. II *Cauchy- (Anfangswert-) Probleme*

$$\begin{cases} u_t + F(x, u, Du, D^2u) = 0 & (t, x) \in (0, \infty) \times \Omega \subset \mathbb{R}^n \\ u(0, x) = g(x) & x \in \Omega \end{cases} \tag{1.2}$$

g ist diesmal die stetige, gegebene Anfangswertbedingung

Definition 1.1.1. Wenn eine partielle Differentialgleichung der Ordnung $k \geq 1$ ist, wird $U : \Omega \mapsto \mathbb{R}^n$ eine *klassische Lösung* genannt, falls $u \in C^k(\Omega)$ ist und die Gleichung an jedem $x \in \Omega$ bzw. $x \in \Omega \times [0, \infty)$ löst.

Anmerkung 1.1.2. Unter passenden Anfangs-, beziehungsweise Randwerten sind die Lösungen meistens eindeutig, aber Existenz kann nicht immer gezeigt werden. Ein Beispiel folgt später.

Da das Zeigen der Existenz von zentraler Bedeutung ist, gibt es verschiedene Arten von sogenannten *schwachen Lösungen*, die Annahmen in der klassischen Lösung lockern.
Es gibt einige Konzepte für schwache Lösungen, sehr bekannt sind zum Beispiel distributionelle Lösungen. Diese Lösungen existieren meistens und sind oft auch eindeutig. Im Kontext der Hamilton-Jacobi-Gleichung führt deren nicht-Linearität zu Problemen für Existenz und Eindeutigkeit der distributionellen Lösung.
Lions und Crandall [5] haben in 1983 eine andere Art der schwachen Lösung, die *Viskositätslösung* eingeführt. Diese funktioniert vor allem im Kontext von Hamliton-Jacobi-Gleichungen, aber auch für viele andere Differentialgleichungen erster und zweiter Ordnung, sehr gut. Außerdem können Existenz und Eindeutigkeit gezeigt werden. Für einen ausführlichen Vergleich der zwei Lösungen sei auf [6] verwiesen.

1.2. Prinzip der nachlassenden Viskosität

Als Beispiel für eine solche Lösung kann man als Spezialfall einer stationären Hamilton-Jacobi-Gleichung eine einfache Eikonal-Gleichung betrachten:

$$\Omega = (-1, 1) \subset \mathbb{R} \text{ und } f(x) = 1$$

Es soll also

$$\begin{cases} |u'(x)| = 1 & \text{für } x \in (-1, 1) \\ u(x) = 0 & \text{für } x = \pm 1 \end{cases} \tag{1.3}$$

gelöst werden. Bei näherer Betrachtung fällt auf, dass es keine klassische Lösung für dieses Problem geben kann. Da am Rand jeweils der Wert Null angenommen wird, die Ableitung jedoch im Betrag konstant Eins ist, muss die Ableitung der Lösung mindestens einen Sprung machen.
Im nächsten Schritt könnte man nach lipschitzstetigen Lösungen suchen, die fast überall unsere Gleichung lösen. Nun stellt man aber schnell fest, dass es dann sogar überabzählbar viele Lösungen gibt

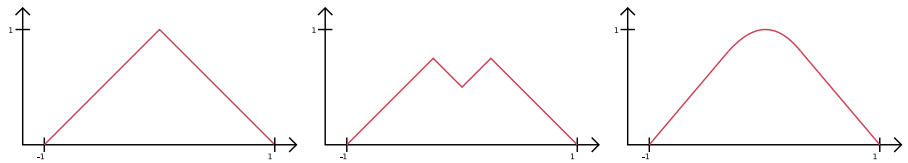


Abbildung 1.1.: Links, Mitte: Zwei mögliche lipschitzstetige Lösungen
Rechts: Lösung des Problems mit Viskositäts-Term

Die Herausforderung ist es nun, die „richtige“ Lösung zu wählen. Die Idee ist es nun durch das Addieren eines weiteren, kleinen Terms die Gleichung eindeutig lösbar zu machen. Dieser Term heißt *Viskositäts-Term*. In diesem Fall werden Lösungen u_ε der Form

$$\begin{cases} -\varepsilon(u_\varepsilon)'' + |(u_\varepsilon)'(x)| = 1 & \text{für } x \in (-1,1) \\ u_e(x) = 0 & \text{für } x = \pm 1 \end{cases} \tag{1.4}$$

betrachtet, wobei Gleichung (1.4) genau Gleichung (1.3) für den Fall $\varepsilon = 0$ entspricht. Im Gegensatz zu vorher lässt sich Gleichung (1.4) eindeutig lösen. Die Lösung ist gegeben durch:

$$\begin{cases} u_\varepsilon(-x) = u_\varepsilon(x) & \text{für } x \in [-1,1] \\ u_\varepsilon(x) = x + 1 - \varepsilon(e^{\frac{x}{\varepsilon}} - e^{\frac{-1}{\varepsilon}}) & \text{für } -1 \leq x \leq 0 \end{cases}$$

Nach Nehmen des Grenzwert in ε wird folgende Lösung erhalten:

$$\lim_{\varepsilon \mapsto 0} u_\varepsilon(x) = 1 - |x| = u_0(x)$$

Diese Lösung u_0 heißt *Viskositätslösung* der Gleichung (1.3)
Diese Methode nennt sich die *Methode der nachlassenden Viskosität* und kann für Hamilton-Jacobi-Gleichungen verallgemeinert werden.

1.3. Viskositätslösungen für Hamilton-Jacobi-Gleichungen

Analog zum letzten Abschnitt wird aus der Hamilton-Jacobi-Gleichung

$$\begin{cases} u_t + H(x, Du) = 0 & (x,t) \in \Omega \subset \mathbb{R}^n \times (0,\infty) \\ u(x, t=0) = g(x) & x \in \Omega \end{cases} \tag{1.5}$$

die Gleichung

$$\begin{cases} u_t^\varepsilon + H(x, Du^\varepsilon) - \varepsilon \Delta u^\varepsilon = 0 & (x,t) \in \mathbb{R}^n \times (0,\infty) \\ u^\varepsilon(x, t=0) = g(x) & x \in \mathbb{R}^n \end{cases} \tag{1.6}$$

Unabhängig von dieser Methode wird eine Viskositätslösung wie folgend definiert.

Definition 1.3.1. Angenommen u ist beschränkt und gleichmäßig stetig auf $R^n \times [0,T]$ für jedes $T > 0$. Dann ist u eine *Viskositätslösung*, wenn folgende drei Bedingungen erfüllt sind:

1. $u = g$ auf $\mathbb{R}^n \times \{t = 0\}$ und
2. Für jede Testfunktion $v \in C^\infty(\mathbb{R}^n \times (0, \infty))$ gilt:
   Falls $u - v$ ein lokales Maximum in einem Punkt$(x_0, t_0) \in \mathbb{R}^n \times (0, \infty)$ hat, dann gilt:

$$v_t(x_0, t_0) + H(Dv(x_0, t_0), x_0) \leq 0 \tag{1.7}$$

   (u ist eine Viskositäts-Sublösung)

3. Und falls $u - v$ ein lokales Minimum in einem Punkt$(x_0, t_0) \in \mathbb{R}^n \times (0, \infty)$ hat, dann gilt:

$$v_t(x_0, t_0) + H(Dv(x_0, t_0), x_0) \geq 0 \tag{1.8}$$

   (u ist eine Viskositäts-Superlösung)

Evans zeigt in seinem Buch auch, dass eine Lösung, die durch die Methode der nachlassenden Viskosität produziert wurde, diese Eigenschaften erfüllt. Allerdings gibt es auch komplett andere Ansätze, die zu einer Viskositätslösung führen können.

Anmerkung 1.3.2. Da der Wert von $v(x_0)$ für die Ungleichungen irrelevant ist, kann man immer $u(x_0) = v(x_0)$ annehmen.
Außerdem kann man durch das Austauschen von $\phi(x)$ durch $\psi(x) = \phi(x) \pm (x - x_0)^4$ immer annehmen, dass das lokale Extremum strikt ist.

1.3.1. Konsistenz

Eine wichtige Frage ist nun natürlich, ob klassische Lösungen überhaupt Viskositätslösungen sind und unter welchen Umständen der Umkehrschluss gilt. Dies wird durch die folgenden Sätze geklärt.

Satz 1.3.3. (Klassische Lösungen sind Viskositätslösungen)
Sei $u \in C^1(\mathbb{R}^n \times [0, \infty))$. Falls u beschränkt und gleichmäßig stetig ist, ist u eine Viskositätslösung.
Beweis: Wo $u - v$ ein Extremum annimmt, erhält man über die Bedingung erster Ordnung, dass

$$\begin{cases} Du(x_0, t_0) = Dv(x_0, t_0) \\ u_t(x_0, t_0) = v_t(x_0, t_0) \end{cases}$$

Nun kann man direkt sehen, dass die zweite Bedingung mit „ = “ erfüllt ist.

Für die andere Richtung muss zunächst ein Lemma eingeführt werden.

Lemma 1.3.4. (Berühren durch eine C^1-Funktion)
Sei $u : \mathbb{R}^n \mapsto \mathbb{R}$ stetig und auch am Punkt x_0 differenzierbar. Dann existiert eine Funktion $v \in C^1(\mathbb{R}^n)$ sodass:

$$u(x_0) = v(x_0)$$

und

$$u - v \text{ hat ein striktes lokales Maximum in } x_0$$

Beweis: Siehe [3], Seiten 584-585

Satz 1.3.5. (Konsistenz von Viskositätslösungen)
Sei u eine Viskositätslösung von (1.5), und u sei zusätzlich differenzierbar in einem Punkt $(x_0, t_0) \in \mathbb{R}^n \times (0, \infty)$, dann gilt:

$$u_t(x_0, t_0) + H(Du(x_0, t_0), x_0) = 0$$

Beweis: Der Beweis geht über das vorherige Lemma und benutzt den Standard-Glätter, siehe [3], Seiten 585-586

Also entspricht insbesondere eine differenzierbare Viskositätslösung einer klassischen Lösung.

1.3.2. Eindeutigkeit

Um die Eindeutigkeit einer Viskositätslösung zu klären, wird als erstes eine Zeit T fixiert und folgendes, leicht generalisiertes Problem betrachtet.

$$\begin{cases} u_t + H(x, Du) = 0 & (x, t) \in \mathbb{R}^n \times (0, T] \\ u(x, t = 0) = g(x) & x \in \mathbb{R}^n \end{cases} \tag{1.9}$$

Nun ist eine beschränkte und gleichmäßig stetige Funktion u wieder eine Viskositätslösung, wenn sie Definition 1.3.1 erfüllt. Nur muss t_0 jetzt in $(0, T]$ statt in $(0, \infty)$ liegen. Zusätzlich wird für den Beweis ein weiteres Lemma für die Behandlung vom Endpunkt T benötigt.

Lemma 1.3.6. (Extrema am Endpunkt)
Angenommen u ist eine Viskositätslösung von (1.5) und $u - v$ nimmt ein lokales Maximum (Minimum) in einem Punkt $(x_0, t_0) \in \mathbb{R}^n \times (0, T]$ an. Dann gilt:

$$v_t(x_0, t_0) + H(Dv(x_0, t_0), x_0) \leq 0 (\geq 0)$$

Insbesondere wird $t_0 = T$ erlaubt.
Beweis: Wird über einen Grenzwert t_ε, der gegen t konvergiert, bewiesen. Siehe [3], Seiten 586-587

Zusätzlich werden für die Eindeutigkeit Lipschitz-Bedingungen für die Hamilton-Funktion benötigt.

$$\begin{cases} |H(p,x) - H(q,x)| \leq C|p-q| \\ |H(p,x) - H(p,y)| \leq C|x-y|(1+|p|) \end{cases} \tag{1.10}$$

für $x, y, p, q \in \mathbb{R}^n$ und eine davon unabhängige Konstante $C \geq 0$

Satz 1.3.7. (Eindeutigkeit von Viskositätslösungen)
Unter Annahme von (1.10) existiert höchstens eine Lösung von (1.5)
Beweis: Im Beweis wird wie üblich angenommen, dass es zwei verschiedene Lösungen gibt. Dies wird dann mit Hilfe von (1.10) zum Widerspruch geführt. Siehe [3], Seiten 587-590

1.3.3. Existenz

Für den Beweis der Existenz von einer Viskositätslösung gibt es verschiedene Methoden:.
Zum einen könnte man versuchen zu zeigen, dass der Viskositäts-Grenzwert (1.6) immer existiert. Dann müsste man gute und gleichmäßige Abschätzungen zur Lösung der ursprünglichen Gleichung finden. Das ist möglich, aber im Allgemeinen sehr kompliziert.
Außerdem kann man mit Hilfe weiterer Lemmata und unter der starken Annahme des Vergleichsprinzip die Existenz über die Methode von Perron zeigen (siehe[4]). Das Problem ist dann jedoch, dass man in jedem Fall zuerst das Vergleichsprinzip beweisen muss, was meistens keine Arbeit erspart.
Im Folgenden wird eine Methode präsentiert, die für eine andere Art der Differentialgleichung Existenz zeigt, und für die zusätzlich eine explizite Lösungsformel gezeigt werden kann. Am Ende wird der Zusammenhang zu den bis jetzt behandelten Hamilton-Jacobi-Gleichungen erläutert.

1.4. Kontrolltheorie und das Prinzip des dynamischen Programmierens

Zuerst wird eine kurze Einführung in Kontrolltheorie gegeben. Danach wird mit Hilfe des sogenannten Prinzips des dynamischen Programmierens eine

Verbindung zu einer anderen Klasse an PDG, den Hamilton-Jacobi-Bellman-Gleichungen, gezogen. Zu dieser Klasse an PDG wird man dann mit den neuen Methoden Existenz zeigen können, und sogar eine explizite Form erhalten.

1.4.1. Einführung in Kontrolltheorie

In Kontrolltheorie soll Kontrolle über die Lösung ausgeübt werden, um bestimmte, gewünschte Lösungen für eine PDG zu bekommen. Dafür wird folgende, ordinäre Differentialgleichung (ODE) betrachtet:

$$\begin{cases} \frac{\mathrm{d}}{\mathrm{d}s}\boldsymbol{y}_x(s) = \boldsymbol{f}(\boldsymbol{y}_x(s), \boldsymbol{\alpha}(s)) & t < s < T \\ \boldsymbol{y}_x(t) = x \end{cases} \tag{1.11}$$

Hier ist $T > 0$ eine feste Endzeit, $x \in \mathbb{R}^n$ ein Startpunkt der Lösung $\boldsymbol{y}_x(\cdot)$ zur Startzeit $t \geq 0$. Ab dann verhält sich $\boldsymbol{y}_x(\cdot)$ entsprechend der ordinären Differentialgleichung (ODE)(1.11), wobei

$$\boldsymbol{f} : \mathbb{R}^n \times A \mapsto \mathbb{R}^n$$

eine gegebene, lipschitzstetige Funktion ist und A eine kompakte Teilmenge von z.B. $\mathbb{R}^m$ ist.
Die Funktion $\boldsymbol{\alpha}(\cdot)$ in (1.11) ist die *Kontrolle*, die in Abhängigkeit von A die PDG beeinflusst. Zunächst wird mit

$$\mathcal{A} \coloneqq \{\boldsymbol{\alpha} : [0,T] \mapsto A | \boldsymbol{\alpha}(\cdot) \text{ ist messbar}\} \tag{1.12}$$

die Menge der *zulässigen Kontrollen* definiert.
Auf Grund der Beschränktheit und Lipschitzstetigkeit, erhält man nun folgende Abschätzungen für $\boldsymbol{f}$:

$$|f(y,a)| \leq C, \quad |f(x,a) - f(y,a)| \leq C|x-y| \quad (x,y \in \mathbb{R}^n, a \in A)$$

Aufgrund dieser Abschätzungen kann die Existenz und Eindeutigkeit einer lipschitzstetigen Lösung $\boldsymbol{y}_x(\cdot) = \boldsymbol{y}_x^{\boldsymbol{\alpha}(\cdot)}(\cdot)$ in dem Intervall $[t,T]$ gezeigt werden. Diese Funktion $\boldsymbol{y}_x^{\boldsymbol{\alpha}}$ löst (1.11) fast überall.
$\boldsymbol{y}_x(\cdot)$ wird die *Antwort* des Systems auf die Kontrolle $\boldsymbol{\alpha}(\cdot)$ und $\boldsymbol{y}_x(s)$ der *Status* des Systems zur Zeit s genannt.
Das Ziel ist es nun, eine „optimale“ Kontrolle $\boldsymbol{\alpha}^*(\cdot)$ zu finden. Dafür muss aber zuerst ein Kriterium für Optimalität eingeführt werden.
Wenn ein Startpunkt $x \in \mathbb{R}^n$ und $0 \leq t \leq T$ gegeben ist, wird für $\boldsymbol{\alpha}(\cdot) \in A$ das *Kostenfunktional*

$$C_{x,t}[\boldsymbol{\alpha}(\cdot)] \coloneqq \int_t^T r(\boldsymbol{y}_x(s), \boldsymbol{\alpha}(s))\mathrm{d}s + g(\boldsymbol{y}_x(T)) \tag{1.13}$$

definiert. Hierbei löst $\boldsymbol{x}(\cdot) = \boldsymbol{x}^{\boldsymbol{\alpha}(\cdot)}(\cdot)$ die ODE (1.11).

$$r : \mathbb{R}^n \times A \mapsto \mathbb{R} \text{ und } \quad g : \mathbb{R}^n \mapsto \mathbb{R}$$

sind gegebene Funktionen.
Geläufig ist es r die *laufenden Kosten pro Zeiteinheit* und g die *Kosten der Terminierung* zu nennen. Zusätzlich wird im Folgenden angenommen, dass

$$\begin{cases} |r(x,a)|, \quad |g(x)| \leq C \\ |r(x,a) - r(y,a)|, \quad |g(x) - g(y)| \leq C(|x-y|) \end{cases} \tag{1.14}$$

für eine Konstante C gilt.
Gegeben einem Startpunkt $x \in \mathbb{R}^n$ und $0 \leq t \leq T$ soll die Gleichung (1.13) bezüglich $\boldsymbol{\alpha}$ minimiert werden. Gesucht ist also die optimale Kontrolle unseres Systems (1.11).
Da T hier endlich ist, wird ein Problem mit einem *finiten Horizont* betrachtet, für einen unendlichen Horizont vergleiche [3], Seite 605.

1.4.2. Dynamic programming principle

Die Methode des *dynamischen Programmierens* versucht nun das obige Problem durch das Betrachten der *Wertefunktion*

$$u(x,t) \coloneqq \inf_{\boldsymbol{\alpha}(\cdot)\in\mathcal{A}} \quad C_{x,t}[\boldsymbol{\alpha}(\cdot)] \qquad (x \in \mathbb{R}^n, 0 \leq t \leq T)$$

zu lösen.
Die Idee ist es, diese geringsten Kosten $u(x,t)$ als Funktion der Variablen x und t zu untersuchen. Dafür wird das Problem der optimalen Kontrolle (1.13) in eine größere Klasse an Problemen, abhängig von den Werten von x und t, eingebettet. Später wird u eine gewisse Hamilton-Jacobi-Gleichung lösen. Im Umkehrschluss wird eine Lösung der Letzteren auch für die optimale Kontrolle von (1.11) helfen.
Seien ab jetzt $x \in \mathbb{R}^n$ und $0 \leq t \leq T$ fest.

Satz 1.4.1. (Optimalitätsbedingungen)
Für jedes $h > 0$ sodass $t + h \leq T$, gilt:

$$u(x,t) = \inf_{\boldsymbol{\alpha}(\cdot)\in\mathcal{A}} \left\{ \int_t^{t+h} r(\boldsymbol{y}_x(s), \boldsymbol{\alpha}(s))\mathrm{d}s + u(\boldsymbol{y}_x(t+h), t+h) \right\} \tag{1.15}$$

wobei $\boldsymbol{y}_x(\cdot) = \boldsymbol{y}_x^{\boldsymbol{\alpha}(\cdot)}(\cdot)$ die ODE (1.11) für eine gegebene Kontrolle $\boldsymbol{\alpha}(\cdot)$ löst.
Beweis: Siehe [3], Seiten 592-594.

Somit ist bereits eine implizite Lösungsdarstellung der Wertefunktion erreicht.

1.4.3. Hamilton-Jacobi-Bellman Gleichungen

Letztendlich ist es das Ziel, eine „infinitesimale Version“ der Optimalitätsbedingungen (1.15) als eine PDG aufzuschreiben. Dafür ist es nötig, zu überprüfen, ob die Wertefunktion u lipschitzstetig und beschränkt ist.

Lemma 1.4.2. (Abschätzungen für die Wertefunktion)
Es existiert eine Konstante C, sodass

$$|u(y,t)| \leq C$$
$$|u(y,t) - u(\hat{y},\hat{t})| \leq C(|y-\hat{y}| + |t-\hat{t}|)$$

für alle $x, \hat{x} \in \mathbb{R}^m, 0 \leq t, \hat{t} \leq T$ gilt.
Beweis: Annahme (1.14) zeigt zusammen mit der Endlichkeit von T Beschränktheit. Für die Lipschitzstetigkeit sei auf [3], Seiten 594-596 verwiesen.

Die *Hamilton-Jacobi-Bellman Gleichung* ist eine Hamilton-Jacobi Gleichung

$$u_t + H(x, Du) = 0 \qquad (x,t) \in \mathbb{R}^n \times (0,T)$$

mit Hamilton-Funktion

$$H(x,p) = \min_{a \in A}\{\boldsymbol{f}(x,a) \cdot p + r(x,a)\}, \qquad (p, x \in \mathbb{R}^n)$$

Die Verbindung zu den vorherigen Ergebnissen ist durch den folgenden Satz gegeben.

Satz 1.4.3. (Eine PDG für die Wertefunktion)
Die Wertefunktion u ist eine Viskositätslösung des folgenden Endwertproblems der Hamilton-Jacobi-Bellman Gleichung:

$$\begin{cases} u_t + \min_{a \in A}\{\boldsymbol{f}(x,a) \cdot Du + r(x,a)\} = 0 & (x,t) \in \mathbb{R}^n \times (0,T) \\ u(x, t=T) = g(x) & x \in \mathbb{R}^n \end{cases} \tag{1.16}$$

Beweis: Siehe [3]

Zusammen mit Gleichung (1.15) und Wahl von $h = T - t$ kann man mit Hilfe des Endwertproblems eine explizite Darstellung der Lösung erhalten.

Anmerkung 1.4.4. Da statt einem Start- jetzt ein Endwertproblem betrachtet wird, muss die Definition der Viskositätslösung für (1.16) angepasst werden, indem man die Ungleichungen in (1.7) und (1.8) umkehrt. Das wird gemacht,

da in dem Fall wo u eine Lösung von (1.16) ist, dann $w(x,t) \coloneqq u(x, T-t)$ das zugehörige Anfangswertproblem

$$\begin{cases} w_t - \min_{a\in A}\{\boldsymbol{f}(x,a)\cdot Dw + r(x,a)\} = 0 & (x,t)\in\mathbb{R}^n\times(0,T) \\ w(x,t=0) = g(x) & x\in\mathbb{R}^n \end{cases}$$

löst. Dabei wechseln dann die Vorzeichen.

Die Korrespondenz zwischen der Wertefunktion u und allgemeineren Hamilton-Jacobi-Gleichungen, die konvex in der Gradientenvariable p sind, lässt sich über einen zu Satz 1.4.3 ähnlichem Satz übertragen. Dieser erfordert jedoch eine weitere Vertiefung in die Lösung von Hamilton-Jacobi-Gleichungen durch die Hopf-Lax-Formel. Dazu sei auf [3], Kapitel 3.3 und Abschnitt 10.3.4 verwiesen. Für die Fast-Marching-Methode genügt eine Form der Hamilton-Jacobi-Bellman-Gleichung.

1.4.4. Hamilton-Jacobi-Bellmann-Gleichung als Randwertproblem

Bis jetzt wurden Hamilton-Jacobi Gleichungen als Anfangswertprobleme betrachtet, die Theorie überträgt sich allerdings in großen Teilen auch auf Randwertprobleme. Dort existiert insbesondere auch eine Korrespondenz zwischen Wertefunktion und Lösung der (jetzt stationären) Hamilton-Jacobi-Bellmann-Gleichung.
Dafür wird ein sogenanntes „Austrittszeit-Problem“ betrachtet. Sei $\Omega\subset\mathbb{R}^n$ gegeben.
Sei dafür $T \coloneqq \min\{s\geq 0 | \boldsymbol{y}_x(s)\in\partial\Omega\}$. Zusätzlich wird angenommen, dass T immer endlich ist. Da nun keine Zeitkomponente t mehr vorhanden sein soll, werden die vorherigen Formeln mit $t=0$ benutzt.

$$u(x) = \inf_{\boldsymbol{\alpha}(\cdot)\in\mathcal{A}} \quad C_{x,0}[\boldsymbol{\alpha}(\cdot)] = \inf_{\boldsymbol{\alpha}(\cdot)\in\mathcal{A}}\left\{\int_0^{\tau\wedge T} r(\boldsymbol{y}_x(s),\boldsymbol{\alpha}(s))\mathrm{d}s + u(\boldsymbol{y}_x(\tau\wedge T))\right\} \tag{1.17}$$

und die Funktion $g(x)$ ist nun die Randwert-, statt der Endwertbedingung. Außerdem ist $\tau\wedge T$ definiert als $\min(\tau,T)$. Die Korrespondenz zwischen Lösung der Wertefunktion und der stationären Hamilton-Jacobi-Bellmann-Gleichung ist nun über folgenden Satz (siehe [7], Seite 123) gegeben.

Satz 1.4.5. Die Wertefunktion von (1.17) ist für alle $\tau\geq 0$ eine Lösung von

$$\begin{cases} \sup_{a\in A}\{-\boldsymbol{f}(x,a)\cdot\nabla u(x) - r(x,a)\} = 0 & x\in\Omega \\ u(x) = g(x) & x\in\partial\Omega \end{cases} \tag{1.18}$$

Insgesamt wurde nun ein Einblick in die zugrundeliegende Theorie der Viskositätslösungen erhalten. Als wichtigste Teile wurden allgemein Eindeutigkeit und Konsistenz gezeigt. Zusätzlich wurde die Existenz der, für die Fast-Marching-Methode relevanten, Hamilton-Jacobi-Bellmann-Gleichung gezeigt. Insbesondere hat man dafür explizite Lösungsformen für Anfangs- und Randwertprobleme gesehen.

1.5. Diskretisierung

Die Diskretisierung des Problems wurde von Rouy und Tourin [8] gelöst.
In der Fast-Marching-Methode soll später eine Eikonalgleichung der Form

$$\begin{cases} |\nabla u(x)| = n(x) > 0 \\ u(x)|_{\partial\Omega} = g(x) \end{cases} \tag{1.19}$$

gelöst werden. Um die Ergebnisse aus dem vorherigen Abschnitt nutzen zu können, muss die Gleichung in eine angemessene Form gebracht werden. Dafür wird folgende Umformung benutzt:

$$|\nabla u(x)| = n(x) \iff \sup_{|\alpha|\leq 1}\{\nabla u(x)\cdot\alpha - n(x)\} = 0 \quad \forall x \in \Omega$$

Entsprechend der Notation im vorherigen Kapitel gilt nun

- Das entsprechende Kontrollsystem ist

$$\begin{cases} \dfrac{\mathrm{d}}{\mathrm{d}s}\boldsymbol{y}_x(s) = -\boldsymbol{\alpha}(s) & s \geq 0 \\ \boldsymbol{y}_x(0) = x \end{cases}$$

- Die Kontrolle $\boldsymbol{\alpha}$ stammt aus

$$\mathcal{A} = \{\boldsymbol{\alpha} : \mathbb{R}_+ \mapsto \mathbb{R}^2 | \boldsymbol{\alpha}(\cdot) \text{ ist messbar und } |\boldsymbol{\alpha}(s)| \leq 1, s \geq 0\}$$

- Und $r(\boldsymbol{y}_x, \boldsymbol{\alpha}) = n(\boldsymbol{y}_x)$

Wie am Ende des letzten Kapitels wird T als die erste Austrittszeit von Ω, also

$$T = \min\{t \geq 0 | \boldsymbol{y}_x(t) \in \partial\Omega\}$$

definiert. Aufgrund der strikten Positivität von n ist diese endlich. Das Kostenfunktional ist nach den Ergebnissen im letzten Kapitel

$$C_x[\boldsymbol{\alpha}(\cdot)] = \int_0^{\tau \wedge T} n(\boldsymbol{y}_x(s))\mathrm{d}s + u(\boldsymbol{y}_x(\tau \wedge T))$$

Die Wertefunktion bleibt unverändert

$$u(x) = \inf_{\boldsymbol{\alpha}(\cdot)\in\mathcal{A}} \quad C_x[\boldsymbol{\alpha}(\cdot)] \qquad x \in \mathbb{R}^n$$

Dann wird Satz 1.4.5 verwendet und für jedes $\tau \geq 0$ folgende Darstellung erhalten:

$$u(x) = \inf_{\boldsymbol{\alpha}(\cdot)\in\mathcal{A}} \left\{ \int_0^{\tau \wedge T} n(\boldsymbol{y}_x(s))\mathrm{d}s + g(\boldsymbol{y}_x(T))\mathbb{1}_{T\leq\tau} + u(\boldsymbol{y}_x(\tau))\mathbb{1}_{T>\tau} \right\} \quad (1.20)$$

1.5.1. Numerisches Schema

Folgend den Ergebnissen aus dem letzten Kapitel, soll nun (1.20) diskretisiert werden, um (1.19) numerisch lösen zu können.
Der Einfachheit halber wird ab jetzt angenommen, dass Ω ein rechteckiges Gebiet $(0, X) \times (0, Y)$ ist. Dies Gebiet ist in x- und y-Richtung durch ein regelmäßiges Gitter diskretisiert.
Ansonsten kann anstatt des beschränkten Gebietes Ω ein umfassendes Rechteck betrachtet werden und die Geschwindigkeit außerhalb von Ω auf sehr kleine, positive Werte ε gesetzt werden. Wenn die Weiten des Gitters in x und y-Richtung, $\Delta x, \Delta y > 0$ gegeben sind, kann man Folgendes definieren:

- $(x_i, y_j) = (i\Delta x, j\Delta y)$, $\{i = 0 \cdots N, j = 0 \cdots M | N = \frac{X}{\Delta x}, M = \frac{Y}{\Delta y}\}$
- Die Werte unserer approximierten Lösung an (x_i, y_j) als U_{ij}
- Die Werte von $n(x, y)$ an (x_i, y_j) als N_{ij}

Durch die Wahl von

$$\Delta t = \frac{\Delta x \Delta y}{\sqrt{\Delta x^2 + \Delta y^2}}$$

erhält man ein Finite-Differenzen Schema für (1.20). Als Diskretisierung des Gebietes wird

$$\begin{aligned}
\Omega_h &= \{(i, j) \in \mathbb{N}^2 | (x_i, y_j) \in \Omega\} \\
\partial\Omega_h &= \{(i, j) \in \mathbb{N}^2 | (x_i, y_j) \in \partial\Omega\} \\
\overline{\Omega_h} &= \{(i, j) \in \mathbb{N}^2 | (x_i, y_j) \in \overline{\Omega}\}
\end{aligned}$$

definiert. Für alle inneren Punkte der Diskretisierung werden klassische Finite-Differenzen Operatoren erster Ordnung genommen.

$$D_x^+ U_{ij} = \frac{U_{i+1j} - U_{ij}}{\Delta x} \qquad D_x^- U_{ij} = \frac{U_{ij} - U_{i-1j}}{\Delta x}$$
$$D_y^+ U_{ij} = \frac{U_{ij+1} - U_{ij}}{\Delta y} \qquad D_y^- U_{ij} = \frac{U_{ij} - U_{ij-1}}{\Delta y}$$

Außerdem wird die Funktion

$$g_{ij}(a,b,c,d) = \sqrt{\max(a^+, b^-)^2 + \max(c^+, d^-)^2} - N_{ij} \tag{1.21}$$

definiert, wobei

$$x^+ = \max(0, x) \quad x^- = -\min(x, 0)$$

Diese Approximierung erfüllt nun

$$\begin{cases} U_{ij} = g(x_i, y_j) & \forall (i,j) \in \partial\Omega_h \\ g_{ij}(D_x^- U_{ij}, D_x^+ U_{ij}, D_y^- U_{ij}, D_y^+ U_{ij}) = 0 & \forall (i,j) \in \Omega_h \end{cases} \tag{1.22}$$

Für nicht gleichmäßige Gitter funktionieren die folgenden Abschnitte mit entsprechender lokaler Anpassung von Δx und Δy ebenfalls. Für unstrukturierte Gitter funktionieren sie mit weiteren Schwierigkeiten auch, siehe [9].
Dies lässt sich auf eine ähnliche Art auch in drei Dimensionen machen, unter der Wurzel wird lediglich das Maximum der Differenzen in z-Richtung hinzugefügt.

$$D_x^+ U_{ijk} = \frac{U_{i+1jk} - U_{ijk}}{\Delta x} \qquad D_x^- U_{ijk} = \frac{U_{ijk} - U_{i-1jk}}{\Delta x}$$
$$D_y^+ U_{ijk} = \frac{U_{ij+1k} - U_{ijk}}{\Delta y} \qquad D_y^- U_{ijk} = \frac{U_{ijk} - U_{ij-1k}}{\Delta y}$$
$$D_z^+ U_{ijk} = \frac{U_{ijk+1} - U_{ijk}}{\Delta z} \qquad D_z^- U_{ijk} = \frac{U_{ijk} - U_{ijk-1}}{\Delta z}$$

$$g_{ijk}(a,b,c,d,e,f) = \sqrt{\max(a^+, b^-)^2 + \max(c^+, d^-)^2 + \max(e^+, f^-)^2} - N_{ijk} \tag{1.23}$$

$$\begin{cases} U_{ijk} = g(x_i, y_j, z_k) & \forall (i,j,k) \in \partial\Omega_h \\ g_{ijk}(D_x^- U_{ijk}, D_x^+ U_{ijk}, D_y^- U_{ijk}, D_y^+ U_{ijk}, D_z^- U_{ijk}, D_z^+ U_{ijk}) = 0 & \forall (i,j) \in \Omega_h \end{cases} \tag{1.24}$$

Satz 1.5.1. Das numerische Schema (1.22) bzw. (1.24) konvergiert mit Rate $\sqrt{\Delta t}$ gegen die Viskositätslösung von (1.19)
Beweis: Siehe [8]

Anmerkung 1.5.2. In von (1.22) hängt die Lösung in einem Punkt nur von Nachbarn mit geringerem Wert ab. Bei genauerer Betrachtung fällt auf, dass bei Nachbarn mit einem größerem Wert der betreffende Finite-Differenzen-Operator nicht verwendet wird. Das numerische Schema ist also monoton.

Mit diesem numerischen Schema sind alle notwendigen Hintergründe für die Fast-Marching-Methode geklärt. Auch viele andere Methoden, um die Eikonalgleichung zu lösen, bauen darauf auf. Vor allem die angesprochene Monotonie ist später von großer Bedeutung.

1.6. Fast-Marching-Methode

Diese Eigenschaft der Monotonie wird in der von J. Sethian [1] erfundenen Fast-Marching-Methode ausgenutzt.
Um die PDG von den bekannten Randpunkten aus lösen zu können, wird $g(x) = 0$ gewählt.
Auf den ersten Blick ist dann

$$\begin{cases} |\nabla u(x)| = \dfrac{1}{f(x)} \\ u(x)|_{\partial\Omega} = 0 \end{cases} \tag{1.25}$$

ein normales Randwertproblem. Allerdings wird wie am Anfang erwähnt meistens die Entwicklung einer Grenzfläche untersucht. Dementsprechend ist die Randbedingung auf einer Grenzfläche Γ gegeben, sodass $\Gamma \subset \Omega$. In diesem Fall wird als Ω meistens ein Quader genommen, der Γ enthält, da dies die Berechnung vereinfacht und man beliebige Formen durch das Modifizieren der Geschwindigkeitsfunktion erreichen kann. Γ ist dann die sogenannte initiale Maske, und die Lösung entwickelt sich „nach außen“. Dementsprechend wird dann folgende PDG gelöst:

$$\begin{cases} |\nabla u(x)| = \dfrac{1}{f(x)} \\ u(x)|_{\Gamma} = 0 \end{cases} \tag{1.26}$$

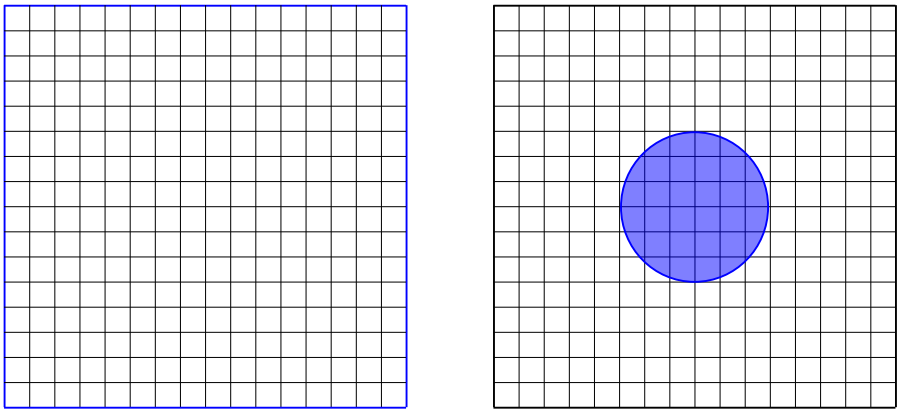


Abbildung 1.2.: Links: Ein normales Randwertproblem, Rechts: Ein Problem mit einer internen Grenzfläche

Um die Lösung sukzessiv berechnen zu können, werden die Punkte in dem diskretisierten Gebiet Ω in drei Kategorien kategorisiert.

- **Akzeptiert** sind die Punkte mit feststehendem Wert
- **In Betrachtung** sind Punkte, die Nachbarn von **akzeptierten** Punkten sind
- **Weit weg** sind die restlichen Punkte

Zu Beginn sind die Punkte in der initialen Maske bekannt, also haben den feststehenden Lösungswert 0 und Status **akzeptiert**. In jedem Schritt wird der Punkt mit dem niedrigsten Wert, der **in Betrachtung** ist, akzeptiert. Es wird genau dieser Punkt ausgewählt, da sich der Wert in diesem Punkt wegen der Monotonie des numerischen Schemas, und der Positivität von f nicht mehr verändert werden kann. Anschließend werden die Nachbarn des neuen Punktes als **in Betrachtung** markiert. Somit existiert in jedem Schritt ein sogenanntes *dünnes Band* um die bekannten Werte, welches sich konsekutiv aus der initialen Maske ausbreitet.
Abhängig davon in wie vielen Richtungen ein Punkt Nachbarn hat die **akzeptiert** sind, wird für die Berechnung der Funktionswerte die drei-, zwei- oder eindimensionale Version von (1.23) benutzt, wobei immer die größtmögliche Lösung ausgewählt wird. Letzteres garantiert, dass beim Ausrechnen der Gleichung der Wert an einem Punkt niemals geringer als der Wert an einem benachbarten akzeptierten Punkt ist.

Algorithmus 1: Klassische Fast-Marching-Methode

Eingabe : Initiale Maske Γ, Geschwindigkeitsfunktion $\frac{1}{f_{ijk}}$
Ausgabe : Die diskrete Lösung von (1.25) auf Ω

```
/* Zuerst wird die initiale Maske initialisiert                    */
```
für *Punkte in* Ω **tue**
  wenn $x_{ijk} \in \Gamma$ **dann**
    $U_{ijk} \longleftarrow 0$;
    $x_{ijk} \longleftarrow$akzeptiert
  sonst
    $U_{ijk} \longleftarrow \infty$;
    $x_{ijk} \longleftarrow$weit weg

für *Punkte in* Ω*, die* akzeptiert *sind* **tue**
  wenn $x_{i,j,k}$ *nicht* akzeptiert *ist und mindestens ein Nachbar* akzeptiert *ist* **dann**
    $x_{ijk} \longleftarrow$in Betrachtung;
    Berechne den neuen Funktionswert anhand von (1.22), wähle dabei die größtmögliche Lösung

```
/* Danach beginnt die Hauptprozedur                                */
```
solange *nicht alle Punkte* akzeptiert *sind* **tue**
  $x_{\min} \longleftarrow$Punkt mit dem kleinsten Wert, der in Betrachtung ist;
  $x_{\min} \longleftarrow$akzeptiert;
  für *Nachbarn* x_{ijk} *von* x_{min} **tue**
    wenn x_{ijk} *nicht* akzeptiert *ist* **dann**
      wenn x_{ijk} weit weg *ist* **dann**
        $x_{ijk} \longleftarrow$in Betrachtung
      Berechne den neuen Funktionswert anhand von(1.22), wähle dabei die größtmögliche Lösung

Der Beweis für die Korrektheit des Algorithmus findet sich im Artikel von Sethian [1]. Außerdem wird dort eine Methode für beliebige Anfangswerte erklärt. Da in jedem Durchlauf der Schleife der Punkt kleinsten Wert bestimmt werden muss, ist es für die Effizienz des Algorithmus von großer Bedeutung, eine effiziente Datenstruktur für diese Aufgabe zu finden. Wie von Sethian[1] diskutiert, eignet sich dafür ein Min-Heap gut. Da es außerdem sein kann, dass Punkte mehrfach einen neuen Wert zugewiesen bekommen, wird zusätzlich eine Lookup-Tabelle benötigt. Mit dieser Tabelle kann man Punkten im Min-Heap neue Werte zuweisen.

1.7. Parallele Implementierungen

Nach der FMM wurden einige andere Algorithmen zur Lösung der Eikonalgleichung entwickelt. Dazu zählen die Fast Sweeping Method (FSM) [10] und die Fast Iterative Method (FIM) [11].
Aufgrund der großen Anzahl an Anwendungsfeldern für die Eikonalgleichung insbesondere auch in Problemen mit vielen Punkten, stieg der Bedarf an schnelleren, parallelen Algorithmen. So berechneten Gillberg et al. [12] die Eikonalgleichung auf einem Gitter mit 139 Millionen Punkten, und in [13] wurde eine Funktion für Turbulenzenmodellierung auf einem Gitter mit 540 Millionen Punkten gelöst.
Diese neuen Algorithmen waren bewusst einfacher zu parallelisieren. Im Gegensatz dazu stand die FMM, welche aufgrund der scheinbar strikten Sequenzialität als schwer parallelisierbar galt. So kommt z.B. die FIM bewusst ohne sequenzielle Teile aus.
Für alle Methoden wurden auch parallele Algorithmen entwickelt. Zum Beispiel [14] für die FMM, [15] für die FIM und [16] für die FSM. Außerdem gibt es kombinierte Methoden wie die Heap-Cell-Method (HCM) [17].
Eine ausführliche Motivation der FMM im Spezifischen kann in [2] nachgelesen werden. Trotz der unterschiedlichen Herangehensweisen teilen die parallelen Umsetzungen ähnliche Herausforderungen, die in diesem Abschnitt besprochen werden.

1.7.1. Domain decomposition

In allen oben genannten parallelen Methoden wird das diskretisierte Gebiet zur Parallelisierung in verschiedene Teilgebiete unterteilt. Im Fall, dass als Gebiet Ω ein Würfel betrachtet wird (was wie in 1.2 gesehen häufig der Fall ist), wäre es eine simple und häufig gemachte Aufteilung, das Gebiet entlang der Achsen regelmäßig aufzuteilen.

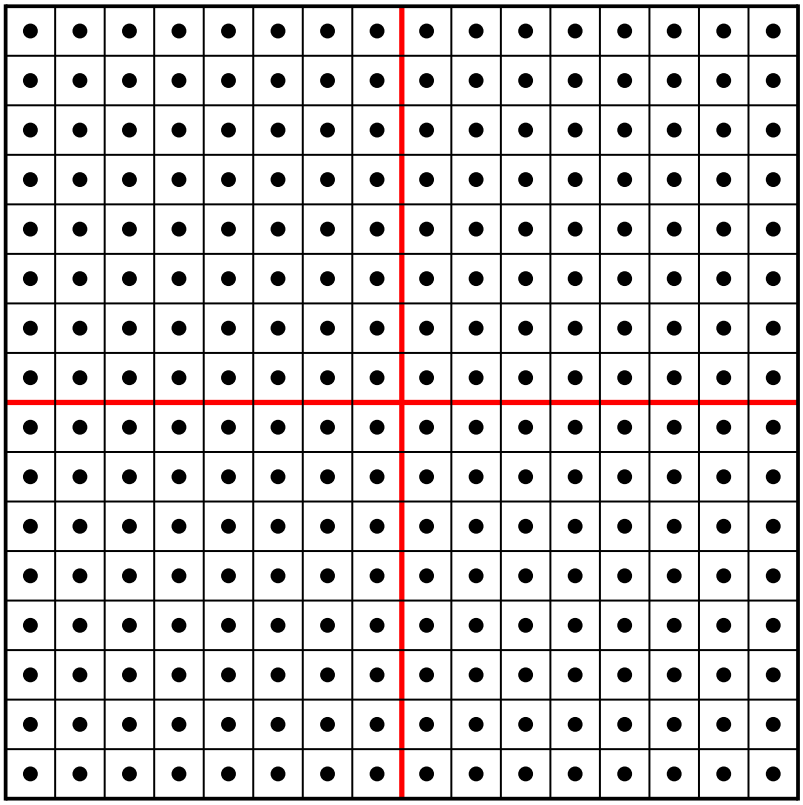


Abbildung 1.3.: Eine regelmäßige Aufteilung in vier Teilgebiete

Danach wird der Algorithmus, meist mit kleinen Anpassungen, auf den Teilgebieten ausgeführt. Da die Lösung an einem Punkt immer von den Nachbarn abhängt, ist es wichtig, dass benachbarte Gebiete mit einander „kommunizieren“. Dafür gibt es verschiedene Ansätze. Zum einen kann die Grenze zwischen den Teilgebieten als sogenanntes „halo“ dafür benutzt werden. Wenn der Wert oder Status in einem Randpunkt aktualisiert wird, teilt dieser dem halo seinen neuen Wert mit. Wenn der entsprechende andere Punkt Informationen von seinem Nachbarn in dem anderen Teilgebiet benötigt, wird der Wert oder Status abgerufen. Zusätzlich gibt es den Ansatz der überlappenden Domain decomposition. Dabei wird um jedes Teilgebiet eine Schicht an sogenannten *Geisterpunkten* Θ hinzugefügt, und das Problem wird auf $\Xi = \Omega \cup \Theta$ gelöst.

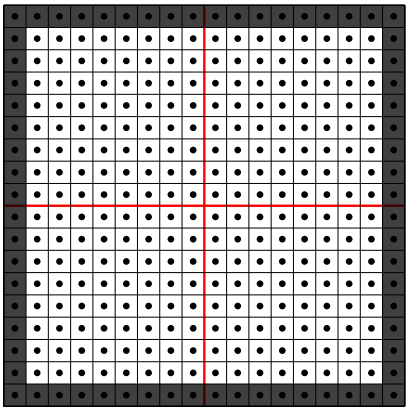


Abbildung 1.4.: Zuerst wird am Rand eine Schicht an Geisterpunkten hinzugefügt

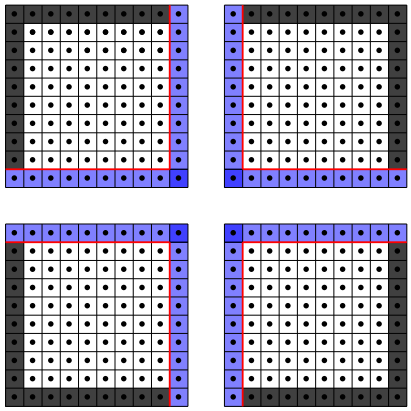


Abbildung 1.5.: Und danach werden die Punkte für den Datenaustausch „angeklebt“

Nun besteht jedes Teilgebiet Ξ_p aus Ω_p, den Punkten aus der originalen Aufteilung, und Θ_p, den Geisterpunkten. Für den Datenaustausch werden dann die Punkte benutzt, die auch in mindestens einem anderen Prozess vorhanden sind. Teilgebiete, die am Rand des ursprünglichen Gebiets Ω waren, bekommen auch eine Schicht an Geisterpunkten, da dann alle Teilgebiete gleich behandelt werden können. Für den Datenaustausch werden alle Punkte benutzt, die in einem benachbarten Teilgebiet Ξ_q liegen.

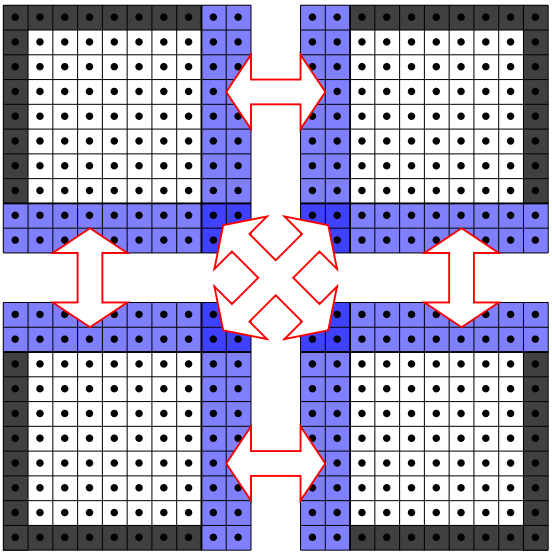


Abbildung 1.6.: In blau: Für den Datenaustausch benutzte Punkte

Die Kommunikation aller Teilgebiete findet dann in einem Schritt statt, in dem Werte und Status der jeweils übereinstimmenden Punkte verglichen werden.

1.7.2. Lastverteilung

Für einen effizienten parallelen Algorithmus ist es von hoher Bedeutung, dass er auf jedem Teilgebiet ähnlich viel Zeit verbringt.
Im Anwendungsfall breitet sich die Geschwindigkeitsfunktion unregelmäßig aus, die initiale Maske ist nicht in der Mitte oder es gibt mehrere verschiedene. Außerdem kann es sein, dass sich die Geschwindigkeitsfunktion so verhält, dass Punkte an der Grenze zu einem benachbarten Gebiet sehr oft kommunizieren müssen.
Für die FIM wurde zum Beispiel eine adaptive Domain decomposition vorgestellt [15]. Dort wird die Berechnung von Blöcken in den Teilgebieten vor jedem Berechnungsschritt auf die Recheneinheiten verteilt.
Außerdem ist es z.B. bei dem Ansatz der überlappenden Domain decomposition nicht einfach, das richtige Maß an Kommunikation zu finden, da diese auch einen signifikanten Teil der Berechnungszeit einnimmt. So will man möglichst lange ohne Kommunikation arbeiten, um möglichst viel rechnen zu können. Gleichzeitig wächst damit aber auch die Gefahr, dass nach einem Kommunikationsschritt viele Werte neu berechnet werden müssen.

2. Sequentielle Fast-Marching-Methode

Das Ziel der Implementierung der sequentiellen Fast-Marching-Methode war es, eine Referenzimplementierung zu haben. Als Arbeitsauftrag sollte darauf geachtet werden, ein möglichst speichereffizientes Programm zu produzieren. Das Programm wurde in C++ geschrieben. Zur Speicherung der Daten wird ein Binärer Min-Heap mit einer Lookup-Tabelle verwendet, um den Wert von bereits eingefügten Punkten zu erneuern. Außerdem wird für das Speichern der Punkte ein simples `struct` mit den Koordinaten und dem (vorläufigen) Wert verwendet. Die Koordinaten werden in der aktuellen Implementierung als `short` gespeichert, da bei dreidimensionalen Problemen auch für sehr viele Gitterpunkte nur wenig Punkte in x-,y-, oder z-Richtung benötigt werden. So entsprechen 1000 Punkte in jeder Richtung schon einer Milliarde Gitterpunkten. Die Eingabedaten sind

- Ein Array aus `bool` der angibt, welche Punkte sich in der initialen Maske befinden
- Ein Array aus `double` für den Wert der Geschwindigkeitsfunktion an jedem Punkt

Für die Berechnung der Gleichung aus dem letzten Kapitel wurde folgender Code verwendet:

Algorithmus 2: LÖSE_QUADRATISCH

Eingabe : Koordinaten l,m,n
Ausgabe : Temporäre Lösung ψ_{temp}
$\psi_{\text{temp}} \longleftarrow \infty$;
`/* Betrachte die x-Richtung um` ψ_1 `und` h_1 `zu errechnen */`
wenn *(l-1,m,n)*$\in \Xi_p$ *ist* **dann**
  wenn $G_{l-1,m,n}$ bekannt *ist und* $\psi_{l-1,m,n} < \psi_{l,m,n}$ **dann**
    $d \longleftarrow -1$;
wenn *(l+1,m,n)*$\in \Xi_p$ *ist* **dann**
  wenn $G_{l+1,m,n}$ bekannt *ist und* $\psi_{l+1,m,n} < \psi_{l,m,n}$ **dann**
    wenn *d=0* **dann**
      $d \longleftarrow 1$;
    sonst wenn $|\psi_{l+1,m,n}| < |\psi_{l-1,m,n}|$ **dann**
      $d \longleftarrow 1$;
wenn $d \neq 0$ **dann**
  $\psi_1 \longleftarrow \psi_{l+d,m,n}$;
  $h_1 \longleftarrow \Delta x^{-1}$;
sonst
  $\psi_1 \longleftarrow 0$;
  $h_1 \longleftarrow 0$;
`/* Betrachte die y-Richtung um` ψ_2 `und` h_2 `zu errechnen */`
$\cdots$;
`/* Betrachte die z-Richtung um` ψ_3 `und` h_3 `zu errechnen */`
$\cdots$;
$a \longleftarrow \sum_i h_i^2$;
$b \longleftarrow -2\sum_i h_i^2 \psi_i$;
$c \longleftarrow \sum_i (h_i^2 \psi_i^2) - F_{l,m,n}^{-2}$;
wenn $(b^2 - 4ac \geq 0)$ **dann**
  $\psi_t \longleftarrow \frac{-b+\sqrt{b^2-4ac}}{2a}$;
  wenn $\psi_1 < \psi_t, \psi_2 < \psi_t, \psi_3 < \psi_t$ **dann**
    $\psi_{\text{temp}} \longleftarrow \psi_t$

Wie im vorherigen Kapitel beschrieben, werden für die Berechnung nur **bekannte** Nachbarn betrachtet und möglichst viele Koordinatenrichtungen verwendet.

2.1. Verschiedene Versionen

In der ersten Version, einer direkten Implementierung der Fast-Marching-Methode, wurden

- Ein Array von `enum` für die Status
- Ein Array aus `double` für den Wert der Geschwindigkeitsfunktion
- Ein Array aus `double` für das Ergebnis

verwendet. Für das Speichern der Punkte im Min-Heap wurde ein `std::vector` benutzt, da immmer nur ein Teil der Punkte **in Betrachtung**, und somit im Min-Heap ist. Der Code dieser Version ist zwar gut lesbar, aber ist speichertechnisch nicht sehr effizient.
In der zweiten Version wurde benutzt, dass in der Fast-Marching-Methode wird zwischen drei Status, **weit weg**, **in Betrachtung** und **akzeptiert**, unterschieden wird, aber man sich diese nicht merken muss.
Bei genauerer Betrachtung fällt nämlich auf, dass der Status eines Punktes bereits aufgrund von schon bekannten Faktoren feststeht:

- **In Betrachtung** sind Punkte, die gegenwärtig im Min-Heap vorhanden sind
- **Akzeptiert** sind diejenigen Punkte, die gegenwärtig nicht im Min-Heap vorhanden sind, aber endlichen Wert besitzen
- **Weit weg** sind die restlichen Punkte

Ob ein Punkt im Min-Heap ist, kann man mit der vorher erwähnten Lookup-Tabelle herausfinden. Um diese Identifizierung nutzen zu können, ist es außerdem zwingend notwendig, dass die Geschwindigkeitsfunktion f in keinem Punkt den Wert 0 annimmt, da ansonsten auch ein Punkt, der **akzeptiert** ist, den Wert unendlich annehmen kann. Dies ist zum Beispiel der Fall, wenn es Hindernisse in dem betrachteten Gebiet gibt (dort kann man alternativ $f = 0.001$ setzen).
Somit wurde ein Array von `Enum` überflüssig gemacht.
In der dritten Version wurde sich folgende Betrachtung zu Nutze gemacht:
Sobald ein neu **akzeptierter** Punkt aus dem Min-Heap entfernt wird, wird dort nie wieder die lokale Geschwindigkeit benutzt. Der Wert in diesem Punkt steht fest und die Geschwindigkeitsfunktion wird lediglich bei der Neuberechnung des Wertes benutzt. Dementsprechend kann das finale Ergebnis in dem Array für die Werte der Geschwindigkeitsfunktion gespeichert werden. Dies führt zu keinen Problemen, da für einen Punkt, der den Status **in Betrachtung** hat,

der neuste temporäre Wert im Min-Heap gespeichert ist.
Außerdem werden für die Berechnung von neuen oder aktualisierten Werten nur Punkte benutzt, die **akzeptiert** sind. Punkte, die nicht **akzeptiert** und nicht im Heap sind, müssen **weit weg** sein. Um unterscheiden zu können, ob ein Punkt **akzeptiert** ist oder nicht(also um zu wissen, ob an dem Eintrag im Array eine Geschwindigkeit oder ein Ergebnis steht), muss man dafür einen Array aus `bool` benutzen.
Somit hat sich der Speicheraufwand auf

- Einen Array aus `bool` um zu gucken ob ein Punkt **akzeptiert** ist
- Den Array aus `double` für die Geschwindigkeiten bzw. das Ergebnis
- Den Min-Heap (mit variabler Größe)

verringert.
Somit existiert eine theoretische Obergrenze von

$N*$(`size_of(bool)`+2*`size_of(double)`+3*`size_of(short)`)+lokale Variablen

N ist dabei die Gesamtzahl aller Punkte. Je nach System entspricht das circa dem 2,5-fachen der Eingabedaten. Zusätzlich ist es wieder möglich an einem Punkt $f = 0$ zuzulassen.

2.2. Ergebnisse

Die sequentielle Fast-Marching-Methode wurde in den folgenden sechs Fällen auf Laufzeit und Speicherverbrauch getestet:

1. Quelle: Ball mit Radius 1/4 um 0.5/0.5/0.5
   Geschwindigkeit: 1
2. Quelle: Punkt in 0.5/0.5/0.5
   Geschwindigkeit: 1+0.5(sin(20πx)sin(20πy)sin(20πz))
3. Quelle:Punkt in 0.5/0.5/0.5
   Geschwindigkeit: 1-0.99(sin(2πx)sin(2πy)sin(2πz))
4. Quelle: 0/0/0
   Geschwindigkeit: 1
5. Quelle: Punkt in 0.5/0.5/0.5
   Geschwindigkeit: 1, in sphärischen Barrieren 0

6. Quellen: Ball mit Radius 1/16 um 0.25/0.25/0.25 und Würfel mit Durchmesser 1/8 um 0.75/0.75/0.75
   Geschwindigkeit: sin(x)^2+cos(y)^2+0.1

Die sphärischen Barrien sind mit $w = \frac{1}{24}$ definiert als

$$(0.15 < R < 0.15 + w) \setminus ((r < 0.05) \cap (z < 0));$$
$$(0.25 < R < 0.25 + w) \setminus ((r < 0.10) \cap (z > 0));$$
$$(0.35 < R < 0.35 + w) \setminus ((r < 0.10) \cap (z < 0));$$
$$(0.45 < R < 0.45 + w) \setminus ((r < 0.10) \cap (z > 0));$$

Die Einzelheiten der Testfälle sind vor allem für den Vergleich mit dem parallelen Algorithmus relevant. Es werden verschieden Große initiale Masken und sich unterschiedlich verhaltende Geschwindigkeitsfunktionen getestet.
Kompiliert wurde mit gcc Version 9.3.0 und cmake 3.16.3 im `release` Modus mit Optimierungsflagge `-O3`. Getestet wurde auf dem calci-Server des Instituts für Angewandte Mathematik Bonn. Der Server besitzt 2 Intel Xeon Platinum 8268 CPUs. Der Speicherverbrauch wurde auf Linux Ubunutu 20.04.1 mit Sysstat [18] erfasst. Ermittlelt wurden Virtual Memory Size (VSZ) und Resident Set Size (RSS). In den Ergebnissen der Laufzeit sieht man, dass die vorhandene Implementation die theoretische $\mathcal{O}(n \log n)$ Rate erfüllt. Die genauen Messungen befinden sich im Anhang A.2
Ab einer Gittergröße von 128^3 kann man auch den im Vergleich zur Eingangsgröße geringen Speicherverbrauch beobachten.

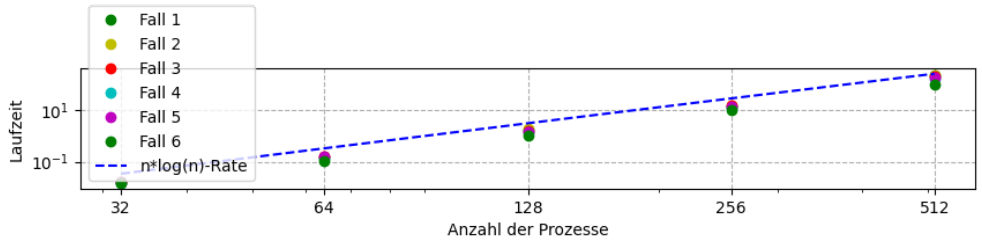


Abbildung 2.1.: Vergleich der durchschnittlichen Laufzeiten

	Eingabegröße	VSZ	RSS	n-faches der Eingabegröße (basierend auf VSZ)
Fall 1:				
64^3	2304	9572	7268	4.15
128^3	18432	33636	30996	1.82
256^3	147456	223076	219648	1.51
512^3	1179650	1726308	1718396	1.46
Fall 2:				
64^3	2304	9812	7340	4.26
128^3	18432	34644	32052	1.88
256^3	147456	227164	224148	1.54
512^3	1179650	1742688	1736488	1.48
Fall 3:				
64^3	2304	9812	7340	4.26
128^3	18432	34644	32052	1.88
256^3	147456	227164	224196	1.54
512^3	1179650	1742688	1738520	1.48
Fall 4:				
64^3	2304	9812	7840	4.26
128^3	18432	34644	32052	1.88
256^3	147456	227164	224148	1.54
512^3	1179650	1726304	1722100	1.46
Fall 5:				
64^3	2304	10340	7508	4.88
128^3	18432	36708	34280	1.99
256^3	147456	251748	138784	1.71
512^3	1179650	1972068	1855252	1.67
Fall 6:				
64^3	2304	9812	7840	4.26
128^3	18432	34644	32052	1.88
256^3	147456	227164	224148	1.54
512^3	1179650	1726304	1722888	1.46

Tabelle 2.1.: Speicherverbrauch der sequentiellen FMM in Kilobytes

3. Parallele Implementierung

Für die parallele Implementierung wurde der Algorithmus von Yang und Stern [2] leicht verändert in C++ programmiert und mit OpenMP parallelisiert. Für die Min-Heaps in den Teilgebieten wurde die selbe Datenstruktur wie im vorherigen Algorithmus genommen.

3.1. Domain decomposition

Wie im Hintergrund-Kapitel besprochen, gibt es verschiedene Ansätze für die Domain decomposition. In dem implementierten Algorithmus wurde der Ansatz der überlappenden Domain decomposition gewählt. Also kriegen alle Teilgebiete, inklusive der Gebiete am Rand, eine Schicht an Geisterpunkten „angeklebt“. In dem Paper wird diese Herangehensweise ausführlich begründet. Ein Hauptargument ist, dass in dem oft vorkommenden Fall, dass die Startoberfläche innerhalb von Ξ ist, diese Punkte für den Algorithmus keine wirklichen Randpunkte sind. Außerdem verursachen die zusätzlichen Randpunkte vergleichsweise wenig Mehraufwand im Gegensatz zur größeren Vereinfachung der Struktur des Codes.
Die überlappende Domain decomposition vereinfacht auch den Code, da normale und Geisterpunkte gleich behandelt werden können und alle Teilgebiete die gleiche Form haben. Zusätzlich werden aufgrund des zugrundeliegenden Problems Daten oft nur in eine Richtung einmal transportiert, sodass nur ein Teil der gesamten geteilten Punkte am Datenaustausch beteiligt ist.
Das Problem der Lastverteilung wird im Algorithmus nicht direkt gelöst. Allerdings kann man je nach Größe von Ξ das Gebiet in mehr Teilgebiete aufteilen, als es Threads bzw. Kerne gibt. In OpenMP gibt es Konstrukte wie z.B. *tasks*, die diese Aufgabe bis zu einem gewissen Maß übernehmen.
So kann es sich auch in einem System mit nur vier Kernen auch lohnen, 32 Teilgebiete zu nehmen.
Im Paper wird das Balancieren der Menge an Kommunikation ausführlich diskutiert.
Im Gegensatz zum Algorithmus im letzten Kapitel werden in jedem Berechnungsschritt nicht nur der Punkt mit dem geringsten Wert betrachtet, sondern bis zu einem Grenzwert alle Punkte, die **in Betrachtung** sind. Dieser Grenzwert

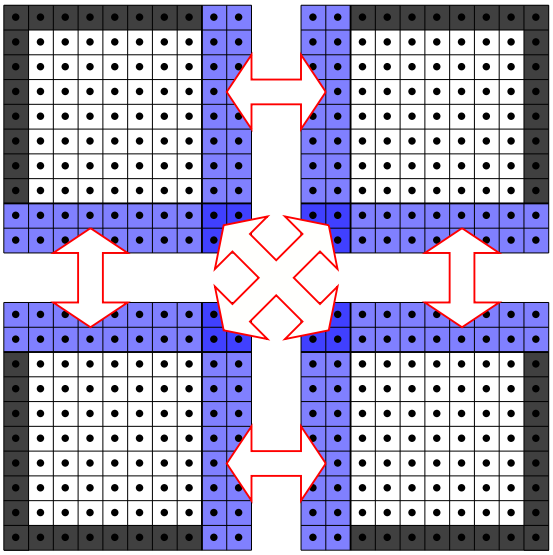


Abbildung 3.1.: Die Teilgebiete mit „angeklebten“ Geisterpunkten. Blau hinterlegte Punkte werden für die Kommunikation benutzt.

hängt vom global geringsten Punkt und einem von der Diskretisierung abhängigen Parameter, im Artikel `stride` genannt, fest. Nach dem darauf folgenden Kommunikationsschritt wird ein weiterer Berechnungsschritt gemacht um die Front auf allen Teilgebieten zum oben genannten Grenzwert zu bringen.
Dieser Ablauf und insbesondere die Wahl `stride` steuern das Verhältnis von Kommunikation und Berechnung im Algorithmus.

3.2. Veränderungen zur sequentiellen FMM

Young und Stern führen für den parallelen Algorithmus weitere Status ein, um sichergehen zu können, dass die Monotonie des numerischen Schemas nicht verletzt wird und um die Kommunikation zwischen den Teilgebieten zu vereinfachen.
Die Kategorie **weit weg** bleibt unverändert. Damit die neues Status auf Deutsch mehr Sinn ergeben, wird **akzeptiert** im Folgenden **bekannt** genannt.

- **Bekannt** wird aufgeteilt zu
  - **Bekannt_fix** für Punkte, die ihren Wert in der Initialisierung bekommen, also Punkte in der initialen Maske.

  - **Bekannt_neu** für Punkte, die aus dem Min-Heap entfernt werden, da sie der Punkt mit dem kleinsten Wert sind.
  - **Bekannt_alt** für Punkte in einem geteilten Gebiet, die **Bekannt_neu** waren und deren Information für den Datenaustausch gesammelt wurde.

- **in Beobachtung** wird aufgeteilt zu
  - **in Beobachtung_neu** für Punkte, die durch Berechnung der Gleichung nicht mehr **weit weg** sind
  - **in Beobachtung_alt** für Punkte in einem geteilten Gebiet, die **in Beobachtung_neu** waren und deren Informationen für den Datenaustausch gesammelt wurden.

Für die Berechnung der Werte der Punkte wird aufgrund der neuen Status eine leicht veränderte Version des Algorithmus Löse_Quadratisch aus dem vorherigen Abschnitt verwendet. Dieser und der restliche Pseudocode befindet sich im Anhang.
Wie oben erwähnt wechselt der Algorithmus zwischen Berechnungs- und Kommunikationsschritten. Zusätzlich wird in jedem Durchlauf das Abbruchkriterium geprüft und der Grenzwert für den Berechnungsschritt neu bestimmt.

3.3. Kommunikation zwischen Threads

Für den Datenaustausch wird eine dreidimensionale Kombination aus `std::array` und `std::vector` benutzt. Die dritte Dimension wird benötigt, um paralleles Schreiben zu erlauben, da `vector.push_back` nicht thread-sicher ist.
In der ersten Dimension werden die neuen Punkte für jedes Teilgebiet gespeichert. In der zweiten Dimension wird dies nach den Nachbarn unterteilt und letztendlich werden in der dritten die Punkte selbst gespeichert.

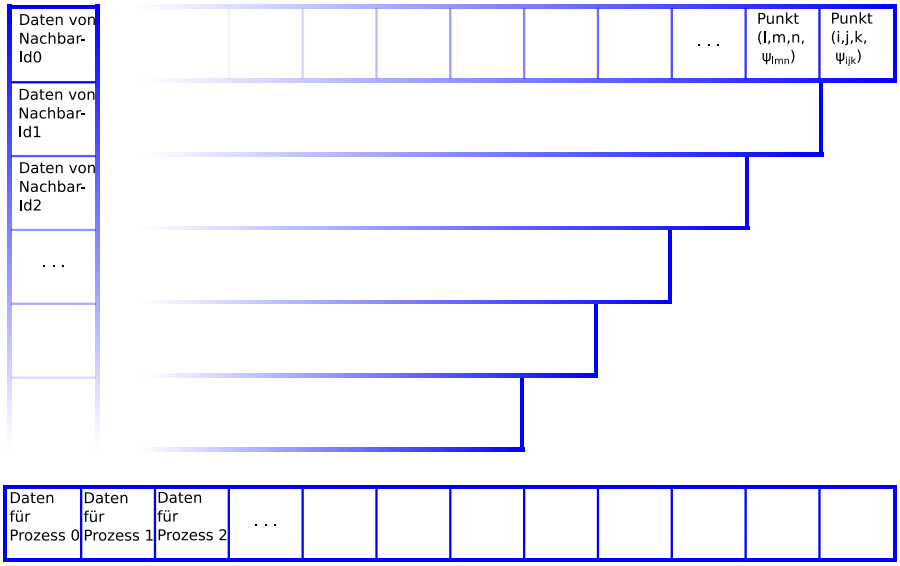


Abbildung 3.2.: Eine Visualisierung der Kommunikation im Algorithmus

Das Auslesen der Daten ist dementsprechend recht einfach.

3.4. Ergebnisse

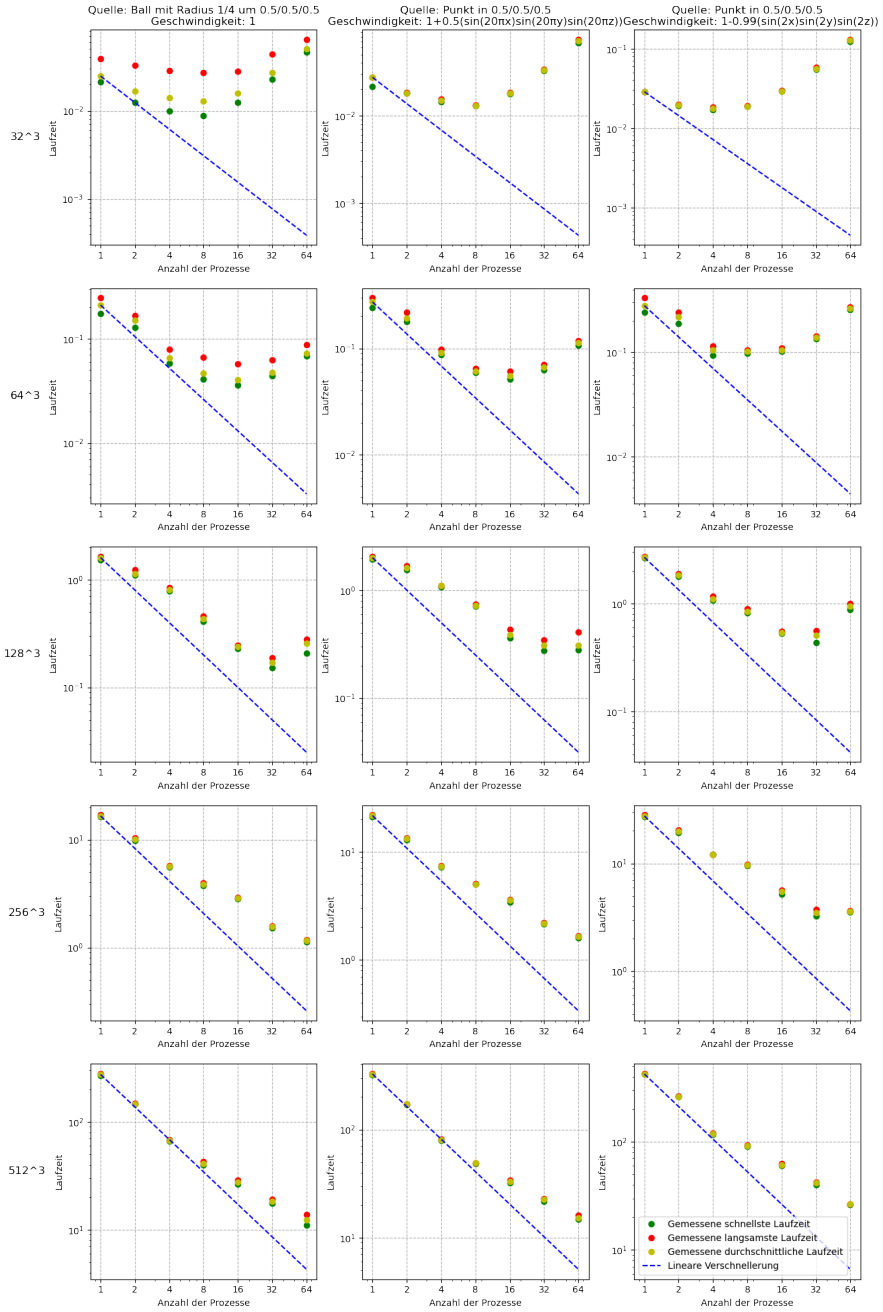
In den folgenden Tests wurden auf einem gleichmäßigem Gitter und Parameter `stride` $= 2 * h$ wobei $h = \Delta x^{-1} = \Delta y^{-1} = \Delta z^{-1}$ durchgeführt. Eine ausführliche Besprechung und Messung verschiedener Parameter findet sich in [2]. Der gewählte Parameter wird dort als allgemein gute Balance zwischen Kommunikation und Berechnung empfohlen.
Es wurden folgende Tests durchgeführt:

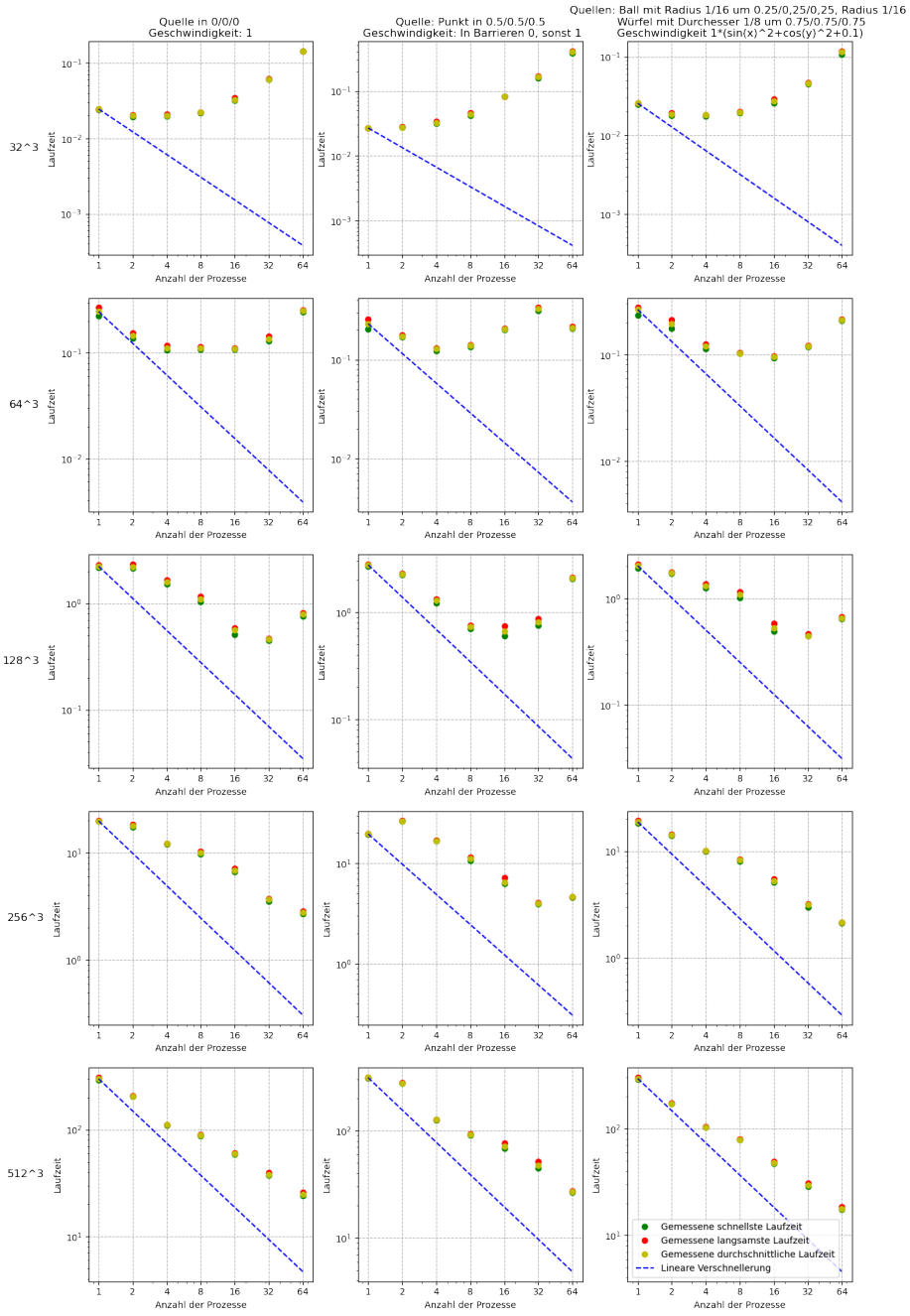
1. Quelle: Ball mit Radius 1/4 um 0.5/0.5/0.5
   Geschwindigkeit: 1
2. Quelle: Punkt in 0.5/0.5/0.5
   Geschwindigkeit: 1+0.5(sin(20πx)sin(20πy)sin(20πz))
3. Quelle: Punkt in 0.5/0.5/0.5
   Geschwindigkeit: 1-0.99(sin(2πx)sin(2πy)sin(2πz))
4. Quelle: 0/0/0
   Geschwindigkeit: 1

5. Quelle: Punkt in 0.5/0.5/0.5
   Geschwindigkeit: 1, in sphärischen Barrieren 0

6. Quellen: Ball mit Radius 1/16 um 0.25/0.25/0.25 und Würfel mit Durchmesser 1/8 um 0.75/0.75/0.75
   Geschwindigkeit: sin(x)^2+cos(y)^2+0.1

Alle Testfälle bis auf Fall Vier und Sechs finden sich auch in [2]. Sie betrachten jeweils eine zentrale initiale Maske mit verschiedenen Geschwindigkeitsfunktionen. Die restlichen Fälle wurden behandelt, um die Leistung des parallelen Algorithmus bei einer stark unbalancierten Verteilung (Fall 4) und bei mehreren initialen Masken (Fall 6) zu testen.
Getestet wurde auf dem calci-Server des Instituts für Angewandte Mathematik Bonn. Der Server besitzt 2 Intel Xeon Platinum 8268 CPUs wobei die Tests jeweils auf 48 Threads beschränkt waren. Kompiliert wurde mit gcc Version 9.3.0 und cmake 3.16.3 im `release` Modus mit Optimierungsflagge `-O3`.
Es wurden pro Fall fünf Durchläufe gemacht. Hier aufgeführt sind der jeweils schnellste und langsamste Durchlauf sowie der Durchschnitt. Die genauen Messdaten finden sich im Anhang A.2.


Quelle: Ball mit Radius 1/4 um 0.5/0.5/0.5
Geschwindigkeit: 1
Quelle: Punkt in 0.5/0.5/0.5
Geschwindigkeit: 1+0.5(sin(20πx)sin(20πy)sin(20πz))
Quelle: Punkt in 0.5/0.5/0.5
Geschwindigkeit: 1-0.99(sin(2x)sin(2y)sin(2z))
32^3
64^3
128^3
256^3
512^3
Laufzeit
Anzahl der Prozesse
Gemessene schnellste Laufzeit
Gemessene langsamste Laufzeit
Gemessene durchschnittliche Laufzeit
Lineare Verschnellerung


Quelle in 0/0/0
Geschwindigkeit: 1
Quelle: Punkt in 0.5/0.5/0.5
Geschwindigkeit: In Barrieren 0, sonst 1
Quellen: Ball mit Radius 1/16 um 0.25/0,25/0,25, Radius 1/16
Würfel mit Durchesser 1/8 um 0.75/0.75/0.75
Geschwindigkeit 1*(sin(x)^2+cos(y)^2+0.1)
32^3
64^3
128^3
256^3
512^3
Laufzeit
Anzahl der Prozesse
Gemessene schnellste Laufzeit
Gemessene langsamste Laufzeit
Gemessene durchschnittliche Laufzeit
Lineare Verschnellerung

Man in den ersten drei Fällen fast lineare Verschnellerung bei bis zu acht Teilgebieten in der höchsten Diskretisierungsstufe.
Auch in den unbalancierten Fällen oder der Geschwindigkeitsfunktion mit Barrieren wird ab 256^3 Punkten eine signifikante Verschnellerung erzielt.
In allen Fällen zahlt sich bei 512^3 Punkten eine Aufteilung in 64 Teilgebiete aus.
Für den Vergleich zu [2] sind die Fälle 1, 2, 3 und 5 zu betrachten. Im Gegensatz zu dieser Arbeit wurde dort auch in Fortran programmiert und für die Parallelisierung MPI statt OpenMP verwendet.
Insgesamt scheint der hier implementierte Algorithmus schneller zu sein. Aufgrund nicht vorhandener absoluter Messwerte für Durchläufe mit mehreren Teilgebieten ist ein genauerer Vergleich jedoch schwer. In dem Fall, wo es nur ein Teilgebiet gibt, ist die Implementation circa doppelt so schnell. Die gemessene parallele Effizienz deckt sich auch mit den hier gemessenen Ergebnissen.
Neben dem Vergleich der parallelen Effizienz des Algorithmus mit sich selbst ist vor allem die Verbesserung bezüglich des sequentiellen Algorithmus interessant. Dort wird in der höchsten Diskretisierungsstufe 512^3 eine Verschnellerung von bis zum Faktor 17.81 in Fall 1 erzielt. Abgesehen von Fall 5 verringert der parallele Algorithmus die Laufzeit mindestens um den Faktor 6. In Fall 5 wird die Laufzeit auch verringert, allerdings deutlich weniger als in allen anderen Fällen.
Auch bei Gittern ab einer Größe von 128^3 kann in allen Fällen eine Verringerung der Laufzeit erreicht werden, allerdings erfordert dies die Wahl von weniger Teilgebieten als 64.

	Anzahl der Teilgebiete						
Fall 1	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3	0.62	0.91	1.09	1.20	0.96	0.56	0.30
64^3	0.67	0.93	2.14	2.98	3.43	2.92	1.92
128^3	1.11	1.56	2.19	4.08	7.38	10.36	6.92
256^3	0.80	1.31	2.33	3.43	4.61	8.47	11.34
512^3	0.80	1.51	3.27	5.40	7.94	12.08	17.81
Fall 2	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3	0.60	0.91	1.11	1.27	0.92	0.49	0.23
64^3	0.57	0.81	1.70	2.56	2.82	2.34	1.39
128^3	0.97	1.21	1.78	2.71	5.04	6.30	6.27
256^3	0.69	1.13	2.04	2.98	4.22	6.88	9.14
512^3	0.66	1.26	2.69	4.42	6.50	9.60	14.07
Fall 3	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3	0.57	0.85	0.93	0.87	0.56	0.29	0.13
64^3	0.55	0.70	1.47	1.50	1.46	1.11	0.58
128^3	0.58	0.84	1.39	1.83	2.87	3.03	1.63
256^3	0.54	0.75	1.23	1.54	2.70	4.25	4.17
512^3	0.48	0.78	1.72	2.23	3.33	4.92	7.72
Fall 4	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3	0.58	0.72	0.70	0.64	0.44	0.23	0.10
64^3	0.51	0.87	1.14	1.13	1.16	0.93	0.51
128^3	0.61	0.61	0.85	1.24	2.42	2.94	1.72
256^3	0.62	0.69	1.02	1.23	1.81	3.36	4.46
512^3	0.54	0.78	1.45	1.80	2.70	4.22	6.45
Fall 5	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3	0.57	0.56	0.48	0.35	0.19	0.09	0.04
64^3	0.48	0.64	0.86	0.80	0.54	0.34	0.52
128^3	0.36	0.43	0.77	1.35	1.49	1.21	0.47
256^3	0.48	0.36	0.56	0.86	1.46	2.37	2.05
512^3	0.32	0.36	0.79	1.09	1.39	2.09	3.68
Fall 6	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3	0.63	0.87	0.89	0.81	0.59	0.35	0.14
64^3	0.58	0.79	1.28	1.49	1.61	1.29	0.72
128^3	0.71	0.83	1.10	1.33	2.73	3.22	2.21
256^3	0.70	0.93	1.32	1.61	2.51	4.20	6.16
512^3	0.59	1.00	1.66	2.18	3.57	5.85	9.63

Tabelle 3.1.: Verhältnis der durchschnittlichen Laufzeiten des parallelen Algorithmus im Vergleich zum sequentiellen Algorithmus

Da der in jedem Schritt ermittelte Grenzwert für die Berechnung von neuen Punkten lediglich vom Gitter und nicht von der Funktion abhängt dauert der parallele Algorithmus selbst auf der in [2] größten getesteten Größe `stride` = $3.5h$ unter Umständen sehr lange. Multipliziert man die Funktion in Fall 6 mit 0.001, sind die Werte der Lösung sehr groß und in jedem Schritt werden nur sehr wenige Punkte erneuert.

Fall 6:
Quellen: Ball an 0.25/0,25/0,25, Radius 1/16 und Würfel an 0.75/0.75/0.75, Durchmesser 1/8
Geschwindigkeit: 0.001*(sin(x)^2+cos(y)^2+0.1)

Test 1: stride = 3.5*h

	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3							
Kürzeste Laufzeit	0.879826809	1.01277466	1.52880226	3.491170232	7.189692072	20.05503698	38.81877616
Längste Laufzeit	1.093094535	1.336536008	1.748234097	3.72645243	7.359152467	20.33629141	39.01279326
Durchschnittliche Laufzeit	1.03305	1.17498	1.64085	3.58372	7.24047	20.2395	38.9093
64^3							
Kürzeste Laufzeit	12.55901366	7.495970327	5.971997006	11.50139813	24.10508272	52.54390196	124.0265057
Längste Laufzeit	14.40115968	9.22722201	7.642826178	14.10236538	26.00828266	55.16604206	125.5287129
Durchschnittliche Laufzeit	12.8702	8.95915	7.59628	12.8575	24.7544	54.0539	124.072
128^3							
Kürzeste Laufzeit	205.758464	121.1799685	74.70921635	55.52880318	68.4556463	75.64254057	294.622047
Längste Laufzeit	211.0105464	123.2709804	77.56862669	59.46156048	71.18818106	81.20389371	300.6901789
Durchschnittliche Laufzeit	208.888	121.578	77.5486	57.1737	70.7613	79.0921	297.657

Tabelle 3.2.: Sehr hohe Laufzeiten schon bei den kleinen Gittergrößen

Dies könnte zum Beispiel gelöst werden, indem `stride` $= \infty$ gewählt wird. Eine andere Idee wäre es `stride` zusätzlich von den Funktionswerten abhängen zu lassen. Da in den Testfällen zuvor der Median der Funktionswerte etwa bei Eins lag, wurde ein Test gemacht wo `stride` mit dem Inversen des Medians aller Funktionswerte multipliziert wurde. In den Ergebnissen zeigt sich, dass `stride` $= \infty$ zwar zu Anfang deutlich schneller ist, aber bei den Gittergrößen 256^3 und 512^3 bedeutend schlechter skaliert. Zusätzlich ist die Laufzeit für `stride` $= \frac{2h}{\texttt{median}}$ ähnlich zu der Laufzeit im ursprünglichen Fall 6.

Test 2: `stride` $= \infty$

	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3							
Kürzeste Laufzeit	0.0175552	0.0270724	0.0203618	0.0192779	0.0213734	0.0280466	0.0498375
Längste Laufzeit	0.0257031	0.0316603	0.0218868	0.0200466	0.0220126	0.029196	0.0583261
Durchschnittliche Laufzeit	0.0199864	0.0295723	0.0209722	0.0198311	0.0216367	0.0285767	0.0544047
64^3							
Kürzeste Laufzeit	0.13612	0.154571	0.136274	0.130261	0.101056	0.088647	0.0633682
Längste Laufzeit	0.140066	0.186351	0.192683	0.156551	0.114293	0.101867	0.0759109
Durchschnittliche Laufzeit	0.137386	0.163561	0.162381	0.14535	0.107774	0.0969732	0.0699897
128^3							
Kürzeste Laufzeit	1.3928	1.21314	0.945593	0.680744	0.47036	0.439918	0.368455
Längste Laufzeit	1.42897	1.27489	0.977457	0.746025	0.574244	0.505228	0.412465
Durchschnittliche Laufzeit	1.41161	1.25378	0.957529	0.721306	0.518061	0.469138	0.388897
256^3							
Kürzeste Laufzeit	14.5993	11.0091	8.41953	5.38523	5.11681	3.3712	2.41481
Längste Laufzeit	15.4968	11.4657	8.52181	5.48124	5.92127	4.02862	3.2017
Durchschnittliche Laufzeit	14.8793	11.1929	8.48565	5.4385	5.445	3.70244	2.6264
512^3							
Kürzeste Laufzeit	221.304	125.307	90.8291	58.797	83.1293	50.7062	32.2593
Längste Laufzeit	225.353	134.59	92.9111	62.537	86.7706	52.5881	34.3086
Durchschnittliche Laufzeit	222.965	132.003	91.5636	60.9681	85.5154	51.472	32.9699

Test 2: `stride` $= \frac{2h}{\texttt{median}}$

	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3							
Kürzeste Laufzeit	0.0256767	0.017363	0.0175406	0.0178531	0.0303322	0.0422577	0.0758629
Längste Laufzeit	0.0377388	0.035973	0.0378959	0.0384354	0.0471112	0.0645394	0.0981017
Durchschnittliche Laufzeit	0.0282956	0.0212934	0.0219123	0.0222009	0.0338019	0.0470678	0.0821744
64^3							
Kürzeste Laufzeit	0.230733	0.179187	0.116901	0.103249	0.100644	0.127809	0.189211
Längste Laufzeit	0.31177	0.257884	0.154469	0.11726	0.118245	0.147644	0.211505
Durchschnittliche Laufzeit	0.266926	0.209503	0.127757	0.107123	0.106077	0.138049	0.201755
128^3							
Kürzeste Laufzeit	1.86719	1.70429	1.29212	0.997401	0.572102	0.403047	0.404378
Längste Laufzeit	1.97827	1.82878	1.43905	1.05689	0.647248	0.515128	0.591415
Durchschnittliche Laufzeit	1.92004	1.77635	1.37964	1.02173	0.612603	0.451918	0.513237
256^3							
Kürzeste Laufzeit	18.133	14.6093	11.3923	8.7901	5.52829	3.35593	2.27859
Längste Laufzeit	19.2157	14.9111	11.6569	9.10739	5.72877	3.52563	2.3639
Durchschnittliche Laufzeit	18.8854	14.7729	11.5342	8.91969	5.62412	3.44796	2.31897
512^3							
Kürzeste Laufzeit	285.697	174.162	109.313	78.7018	47.6471	30.8256	18.2088
Längste Laufzeit	294.59	179.426	110.819	80.2917	49.7633	33.6514	19.1839
Durchschnittliche Laufzeit	289.834	176.559	110.309	79.3869	49.0901	31.7487	18.6779

Tabelle 3.3.: Vergleich zwischen `stride` $= \infty$ und `stride` $= \frac{2h}{\texttt{median}}$

4. Fazit

Nach einer ausführlichen Einführung in die relevante Theorie wurden für diese Arbeit zwei Programme erarbeitet.
Das erste, eine Implementierung der sequentiellen Fast-Marching-Methode, zeichnet sich durch effiziente Speichernutzung aus. Zusätzlich ist die Laufzeit des Algorithmus ähnlich lange für alle verschiedenen Fälle. Somit kann man selbst bei unbekannten Geschwindigkeitsfunktionen eine gewisse Laufzeit erwarten.
Das zweite, eine Implementierung der parallelen Fast-Marching-Methode von Yang und Stern [2], zeichnet sich durch eine gute parallele Effizienz aus. Zusätzlich wurden Fälle mit unbalancierten initialen Masken getestet. Auch dort konnte man eine annehmbare parallele Verschnellerung beobachten. Zusätzlich wurde das Problem des Algorithmus für sehr kleine Funktionswerte dargestellt und ein Lösungsansatz vorgeschlagen.
Da die Fast-Marching-Methode algorithmische Ähnlichkeit zum Algorithmus von Dijkstra hat, lohnt sich gegebenenfalls eine Umstellung auf 3- oder 4- Min-Heaps (siehe [19]). Mit dem Aufstieg von GPUs im paralellen Programmieren, wäre es auch eine Möglichkeit das zweite Programm darauf anzupassen und einen Vergleich mit den hier gemessenen Laufzeiten durchzuführen.
Der gesammelte Code findet sich auch auf https://github.com/MoritzK177/BachelorthesisSourcecode

A. Anhang

A.1. Pseudocode parallele Implementierung

A.2. Absolute Messzeiten der Algorithmen

Algorithmus 3: LÖSE_QUADRATISCH_PARALLEL

Eingabe : Koordinaten l,m,n
Ausgabe : Temporäre Lösung ψ_{temp}
$\psi_{\text{temp}} \longleftarrow \infty$;
`/* Betrachte die x-Richtung um` ψ_1 `und` h_1 `zu errechnen */`
wenn *(l-1,m,n)*$\in \Xi_p$ *ist* **dann**
  wenn $G_{l-1,m,n} \in$ bekannt *ist und* $\psi_{l-1,m,n} < \psi_{l,m,n}$ **dann**
    $d \longleftarrow -1$;
wenn *(l+1,m,n)*$\in \Xi_p$ *ist* **dann**
  wenn $G_{l+1,m,n} \in$ bekannt *ist und* $\psi_{l+1,m,n} < \psi_{l,m,n}$ **dann**
    wenn *d=0* **dann**
      $d \longleftarrow 1$;
    sonst wenn $\psi_{l+1,m,n}| < \psi_{l-1,m,n}$ **dann**
      $d \longleftarrow 1$;
wenn $d \neq 0$ **dann**
  $\psi_1 \longleftarrow |\psi_{l+d,m,n}|$;
  $h_1 \longleftarrow \Delta x^{-1}$;
sonst
  $\psi_1 \longleftarrow 0$;
  $h_1 \longleftarrow 0$;
`/* Betrachte die y-Richtung um` ψ_2 `und` h_2 `zu errechnen */`
$\cdots$;
`/* Betrachte die z-Richtung um` ψ_3 `und` h_3 `zu errechnen */`
$\cdots$;
$nd \longleftarrow$ Anzahl der $h_i \neq 0$;
solange $nd \neq 0$ **tue**
  $a \longleftarrow \sum_i h_i^2$;
  $b \longleftarrow -2 \sum_i h_i^2 \psi_i$;
  $c \longleftarrow \sum_i (h_i^2 \psi_i^2) - F_{l,m,n}^{-2}$;
  wenn *(*$b^2 - 4ac \geq 0$*)* **dann**
    $\psi_t \longleftarrow \frac{-b+\sqrt{b^2-4ac}}{2a}$;
    wenn $\psi_1 < \psi_t, \psi_2 < \psi_t, \psi_3 < \psi_t$ **dann**
      $\psi_{\text{temp}} \longleftarrow$ `Minimum(`ψ_{temp}, ψ_t`)`
  $j \longleftarrow$ Index des maximalen ψ_i;
  $\psi_j \longleftarrow 0$;
  $h_j \longleftarrow 0$;
  $nd \longleftarrow nd - 1$;

Algorithmus 4: Aktualisiert die Werte der Nachbarn eines Punktes, der gerade bekannt geworden ist
AKTUALISIERE_NACHBARN_PARALLEL (i,j,k)

für (l,m,n) *sodass* $(|l-i|+|m-j|+|n-k|)=1$ **tue**
  wenn $(l,m,n) \in \Xi_p$ **dann**
    wenn $G_{l,m,n} \neq$ bekannt_fix *und* $\psi_{l,m,n} > \psi_{i,j,k}$ **dann**
      $\psi_{\text{temp}} \longleftarrow$`LÖSE_QUADRATISCH_PARALLEL(`$l,m,n$`);`
      wenn $\psi_{temp} < \psi_{l,m,n}$ **dann**
        $\psi_{l,m,n} \longleftarrow \psi_{\text{temp}}$;
        $G_{l,m,n} \longleftarrow$ in_Beobachtung_neu;
        wenn `IM_HEAP(`l,m,n`)` **dann**
          `AKTUALISIERE_HEAP(`l,m,n`);`
        sonst
          `EINFÜGEN_HEAP(`l,m,n`);`

Algorithmus 5: MARSCHIERE_DÜNNES_BAND_PARALLEL

Eingabe: Daten des Teilgebiets, $\mathsf{BAND}_{\mathsf{BESCHRÄNKUNG}}$
wiederhole
  wenn `größe_von(`h`)`$=0$ **dann**
    breche ab;
  $(i,j,k) \longleftarrow$ `FINDE_MINIMUM`;
  wenn $|\psi_{i,j,k}| > \mathsf{BAND}_{\mathsf{BESCHRÄNKUNG}}$ **dann**
    breche ab;
  wenn $G_{i,j,k} \neq$bekannt_alt **dann**
    $G_{i,j,k} =$bekannt_neu;
  `ENTFERNE_MINIMUM(`h`);`
  `AKTUALISIERE_NACHBARN(`i,j,k`);`

Algorithmus 6: Sammle alle Daten in den überlappenden Regionen SAMMELN_ÜBERLAPPENDER_DATEN_PARALLEL

Eingabe: Daten des Teilgebiets, Austauschvektor

```
Anzahl_neu ⟵ 0;
für (i,j,k) ∈ ∪_{q∈N_p}(Ξ_p ∩ Ξ_q) tue
    wenn G_{i,j,k} =in_Beobachtung_neu oder G_{i,j,k} =bekannt_neu dann
        Anzahl_neu ⟵ Anzahl_neu +1;
        wenn G_{i,j,k} =in_Beobachtung_neu dann
            G_{i,j,k} =in_Beobachtung_alt;
        sonst
            G_{i,j,k} =bekannt_alt;
        für Prozesse q ∈ N_p tue
            wenn (i,j,k) ∈ Ξ_q dann
                Füge (i,j,k, ψ_exch = ψ_{i,j,k}) Austauschvektor[q][p] hinzu ;
```

Algorithmus 7: Initialisierung des Interface INITIALISIERE_INTERFACE_PARALLEL

Eingabe: Daten des Teilgebiets, Interface Γ

```
ψ ⟵ ∞;
G ⟵weit weg;
für (i,j,k) ∈ Ξ_p ∩ Γ tue
    ψ_{i,j,k} ⟵ 0;
    G_{i,j,k} ⟵ bekannt_fix;
```

Algorithmus 8: Initialisierung des Min-Heaps INITIALISIERE_HEAP_PARALLEL

Eingabe: Daten des Teilgebiets

```
für (i,j,k) ∈ Ξ_p sodass G_{i,j,k} = bekannt_fix tue
    AKTUALISIERE_NACHBARN(i,j,k);
```

Algorithmus 9: Integriere Daten, die von den benachbarten Prozessen erhalten wurden
INTEGRIERE_ÜBERLAPPENDE_DATEN_PARALLEL

```
Eingabe: Daten des Teilgebiets, Austauschvektor[p]
für (l, m, n) ∈ Austauschvektor[p] tue
    Ψ_neu ⟵ ψ_exch;
    wenn ψ_neu < ψ_l,m,n dann
        ψ_l,m,n ⟵ ψ_neu;
        wenn ψ_l,m,n > BAND_BESCHRÄNKUNG dann
            G_l,m,n ⟵ in_Beobachtung_alt;
        sonst
            G_l,m,n ⟵ bekannt_alt;
        wenn IM_HEAP(l,m,n) dann
            AKTUALISIERE_HEAP(l,m,n);
        sonst
            EINFÜGEN_HEAP(l,m,n);
```

Algorithmus 10: Der gesamte Algorithmus PARALLELE_SCHMALBAND_FAST_MARCHING_METHODE

```
INITIALISIERE_INTERFACE_PARALLEL;
INITIALISIERE_HEAP_PARALLEL;
wiederhole
    für Alle Teilgebiete tue
        wenn größe_von(h)=0 dann
            (i, j, k) ⟵ FINDE_MINIMUM;
            MinWert_lokal ⟵ ψ_{i,j,k};
        sonst
            MinWert_lokal ⟵ Weite_Band;
    MinWert_global ⟵ GlobaleReduzierung_MIN(MinWert_lokal);
    Zähler_global ⟵ GlobaleReduzierung_MAX(Zähler_neu);
    wenn MinWert_global ≥ Weite_Band und Zähler_global = 0 dann
        breche ab
    Beschränkung_Band ⟵ min(MinWert_global +stride, Weite_Band);
    MARSCHIERE_DÜNNES_BAND_PARALLEL;
    SAMMELN_ÜBERLAPPENDER_DATEN_PARALLEL;
    INTEGRIERE_ÜBERLAPPENDE_DATEN_PARALLEL;
    MARSCHIERE_DÜNNES_BAND_PARALLEL;
```

Fall 1

Quelle: Ball mit Radius 1/4 um 0.5/0.5/0.5
Geschwindigkeit: 1

	Sequentiell	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3								
Kürzeste Laufzeit	0.0149228	0.0210985	0.0124948	0.00998938	0.00883593	0.0125026	0.0226456	0.0465181
Längste Laufzeit	0.0168535	0.0388093	0.0327078	0.0284455	0.0270033	0.0280882	0.0441277	0.0644641
Durchschnittliche Laufzeit	0.0153255	0.0247099	0.0167984	0.0140729	0.0128035	0.0159029	0.0272146	0.0505655
64^3								
Kürzeste Laufzeit	0.137462	0.174404	0.128679	0.0577605	0.0408075	0.0360612	0.0437954	0.0683731
Längste Laufzeit	0.143116	0.245326	0.166559	0.0785922	0.0659964	0.0572927	0.0622602	0.0872899
Durchschnittliche Laufzeit	0.139261	0.209328	0.148976	0.0649495	0.0467928	0.0406011	0.047661	0.0724795
128^3								
Kürzeste Laufzeit	1.736	1.53371	1.09568	0.780101	0.41147	0.230371	0.151938	0.206455
Längste Laufzeit	1.84519	1.64165	1.22698	0.847627	0.457054	0.24766	0.188665	0.281416
Durchschnittliche Laufzeit	1.77382	1.60098	1.1356	0.810369	0.43524	0.240404	0.171183	0.256417
256^3								
Kürzeste Laufzeit	13.1118	16.3183	9.82403	5.59541	3.77318	2.81687	1.51434	1.13584
Längste Laufzeit	13.457	16.9386	10.4216	5.7025	3.98192	2.89394	1.58816	1.18402
Durchschnittliche Laufzeit	13.2094	16.5763	10.106	5.66152	3.85412	2.86699	1.55912	1.16437
512^3								
Kürzeste Laufzeit	191.559	269.144	145.539	66.3863	39.8943	26.3242	17.6891	11.0594
Längste Laufzeit	286.101	281.152	147.568	68.4896	42.6914	29.0412	19.092	13.7714
Durchschnittliche Laufzeit	220.783	275.168	146.686	67.583	40.8746	27.7969	18.2748	12.3968

Fall 2
Quelle: Ball mit Radius 1/4 um 0.5/0.5/0.5
Geschwindigkeit: 1+0.5(sin(20*π*x) *sin(20πy)*sin(20πz)

	Sequentiell	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3								
Kürzeste Laufzeit	0.0164961	0.0272127	0.0180628	0.0144985	0.0129533	0.017815	0.0329321	0.0679759
Längste Laufzeit	0.0165128	0.0215817	0.0185046	0.015374	0.0131554	0.0182651	0.0338437	0.074417
Durchschnittliche Laufzeit	0.016507	0.0274037	0.018202	0.0148933	0.0130425	0.0180352	0.0334002	0.0720841
64^3								
Kürzeste Laufzeit	0.1544	0.240331	0.179922	0.0866693	0.0591719	0.0516592	0.062578	0.106699
Längste Laufzeit	0.157074	0.298536	0.217442	0.0982031	0.0641242	0.0605342	0.0703711	0.11807
Durchschnittliche Laufzeit	0.155513	0.273585	0.19316	0.0912502	0.0608226	0.0551075	0.0665574	0.112266
128^3								
Kürzeste Laufzeit	1.58838	1.94973	1.54282	1.06585	0.706124	0.361794	0.274214	0.277372
Längste Laufzeit	2.46839	2.05749	1.69134	1.10405	0.742029	0.434834	0.343967	0.410321
Durchschnittliche Laufzeit	1.94652	2.005	1.61073	1.09408	0.718145	0.386421	0.308769	0.310234
256^3								
Kürzeste Laufzeit	15.003	21.0263	13.0012	7.26329	4.99517	3.42321	2.14931	1.60378
Längste Laufzeit	15.0605	21.968	13.4701	7.45856	5.09359	3.65398	2.20353	1.66469
Durchschnittliche Laufzeit	15.0287	21.6572	13.3101	7.36289	5.04642	3.55925	2.18399	1.64454
512^3	214.605	321.51	169.228	79.015	48.2376	32.1518	21.6116	14.8465
Kürzeste Laufzeit	216.462	329.701	172.73	81.3477	49.3518	34.2658	23.0427	16.0613
Längste Laufzeit	215.675	326.189	170.999	80.3176	48.7653	33.1715	22.4754	15.3289
Durchschnittliche Laufzeit								

Fall 3
Quelle: Ball mit Radius 1/4 um 0.5/0.5/0.5
Geschwindigkeit: 1-0.99(sin(2πx)sin(2πy)sin(2πz))

	Sequentiell	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3								
Kürzeste Laufzeit	0.0164749	0.028622	0.0190687	0.0170922	0.0186341	0.0290214	0.0553251	0.123735
Längste Laufzeit	0.0165023	0.0288108	0.0197019	0.0183891	0.0190236	0.0295305	0.0577383	0.130603
Durchschnittliche Laufzeit	0.016483	0.0286996	0.0194296	0.0178058	0.0189172	0.0293581	0.056031	0.127908
64^3								
Kürzeste Laufzeit	0.152854	0.243091	0.187824	0.0941338	0.0980801	0.102587	0.134587	0.257523
Längste Laufzeit	0.155359	0.332314	0.24315	0.114852	0.104785	0.109443	0.143719	0.272116
Durchschnittliche Laufzeit	0.153464	0.280391	0.217922	0.104372	0.102158	0.105189	0.138799	0.2639
128^3								
Kürzeste Laufzeit	1.53888	2.63732	1.79179	1.06768	0.819727	0.528159	0.435623	0.883303
Längste Laufzeit	1.54929	2.72792	1.88152	1.16772	0.890311	0.554896	0.559654	0.999964
Durchschnittliche Laufzeit	1.54323	2.67608	1.84421	1.10694	0.845329	0.538459	0.509661	0.94652
256^3								
Kürzeste Laufzeit	14.7242	27.1717	19.2002	12.0324	9.55822	5.21509	3.23374	3.52821
Längste Laufzeit	15.2062	28.1207	20.4929	12.1708	9.80109	5.60868	3.76701	3.63366
Durchschnittliche Laufzeit	14.8604	27.535	19.8946	12.0774	9.66873	5.49585	3.49456	3.56734
512^3								
Kürzeste Laufzeit	203.82	421.36	260.635	117.766	90.908	60.6228	40.0175	26.1817
Längste Laufzeit	205.152	428.567	263.809	120.011	93.2883	62.6965	42.4323	26.6608
Durchschnittliche Laufzeit	204.468	423.739	262.106	118.953	91.7997	61.4226	41.5388	26.4697

Fall 4:
Quelle: 0/0/0
Geschwindigkeit : 1

	Sequentiell	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3								
Kürzeste Laufzeit	0.0142079	0.023863	0.0193137	0.0197768	0.0219083	0.0316391	0.0607262	0.14199
Längste Laufzeit	0.0142673	0.0247038	0.0203595	0.0206993	0.0223008	0.0342153	0.0613758	0.143635
Durchschnittliche Laufzeit	0.0142319	0.024429	0.0198811	0.0201889	0.0221132	0.0326743	0.0609731	0.143009
64^3								
Kürzeste Laufzeit	0.126091	0.225301	0.137531	0.106476	0.10857	0.108107	0.131065	0.245396
Längste Laufzeit	0.128877	0.269909	0.154024	0.117578	0.11437	0.110631	0.143695	0.254679
Durchschnittliche Laufzeit	0.12723	0.247229	0.145758	0.111479	0.112113	0.109674	0.136757	0.250605
128^3								
Kürzeste Laufzeit	1.3513	2.1605	2.13542	1.51851	1.04023	0.513264	0.451465	0.760051
Längste Laufzeit	1.36124	2.30523	2.34182	1.6492	1.15716	0.588382	0.46874	0.814143
Durchschnittliche Laufzeit	1.35505	2.23262	2.21158	1.59235	1.09329	0.560641	0.461217	0.788111
256^3								
Kürzeste Laufzeit	12.2193	19.6348	17.3809	11.9751	9.75532	6.65277	3.53681	2.70779
Längste Laufzeit	12.3894	19.7973	18.3054	12.2326	10.3005	7.07414	3.71092	2.83991
Durchschnittliche Laufzeit	12.2827	19.7004	17.8262	12.0706	9.9951	6.79508	3.66088	2.75687
512^3								
Kürzeste Laufzeit	160.002	290.625	204.78	110.275	87.9197	59.0292	37.321	23.9695
Längste Laufzeit	161.248	309.701	207.297	111.838	90.3835	60.5295	39.3484	25.784
Durchschnittliche Laufzeit	160.845	299.616	206.125	111.268	89.369	59.6368	38.0932	24.9552

Fall 5
Quelle: Punkt in der Mitte
Geschwindigkeit: In Barrieren 0, sonst 1

	Sequentiell	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3								
Kürzeste Laufzeit	0.0127735	0.026674	0.0275974	0.0313996	0.04192	0.0819718	0.158586	0.38177
Längste Laufzeit	0.0251949	0.0270154	0.0278829	0.0335515	0.0454154	0.0826737	0.168262	0.411148
Durchschnittliche Laufzeit	0.0154046	0.0268522	0.0277461	0.0320973	0.0440648	0.0822599	0.163301	0.397309
64^3								
Kürzeste Laufzeit	0.108223	0.205369	0.170431	0.122447	0.135567	0.200627	0.311881	0.207941
Längste Laufzeit	0.118045	0.255927	0.176823	0.132007	0.141383	0.206254	0.337652	0.215833
Durchschnittliche Laufzeit 0.110609		0.231416	0.173665	0.128684	0.138346	0.20375	0.326989	0.212514
128^3								
Kürzeste Laufzeit	0.987885	2.70667	2.22962	1.2206	0.709233	0.604568	0.753475	2.06946
Längste Laufzeit	0.991855	2.81495	2.30694	1.32793	0.761111	0.74954	0.867861	2.12579
Durchschnittliche Laufzeit	0.989769	2.769	2.2834	1.28721	0.731914	0.664905	0.816036	2.09625
256^3								
Kürzeste Laufzeit	9.47288	19.6348	26.2985	16.7016	10.7563	6.25902	3.94976	4.61734
Längste Laufzeit	9.53147	19.7973	26.6926	16.9721	11.5605	7.21014	4.09578	4.67177
Durchschnittliche Laufzeit	9.5099	19.7004	26.4425	16.8452	11.1194	6.52935	4.00634	4.64829
512^3								
Kürzeste Laufzeit	99.3164	307.2	276.093	125.841	90.7022	67.9101	44.5615	26.572
Längste Laufzeit	99.4957	313.081	278.55	126.694	93.0913	76.7809	51.4858	27.5136
Durchschnittliche Laufzeit	99.415	310.466	277.122	126.263	91.4711	71.2966	47.5114	26.9914

Fall 6:
Quellen: Ball mit Radius 1/16 um 0.25/0,25/0,25 und Würfel mit Durchmesser 1/8 um 0.75/0.75/0.75
Geschwindigkeit: 1*(sin(x)^2+cos(y)^2+0.1)

	Sequentiell	1	2(1x1x2)	4(1x2x2)	8(2x2x2)	16(2x2x4)	32(2x4x4)	64(4x4x4)
32^3								
Kürzeste Laufzeit	0.0160928	0.0248516	0.0179931	0.0176531	0.0194092	0.0259274	0.0451064	0.107677
Längste Laufzeit	0.016112	0.0259174	0.0191923	0.0184147	0.0201328	0.0289169	0.046501	0.118252
Durchschnittliche Laufzeit	0.0161029	0.0256206	0.0184809	0.0180911	0.0198711	0.0270695	0.0461416	0.11526
64^3								
Kürzeste Laufzeit	0.141837	0.232231	0.17617	0.113831	0.102595	0.0932438	0.118542	0.207941
Längste Laufzeit	0.183648	0.275886	0.210869	0.124615	0.104313	0.0973974	0.121785	0.215833
Durchschnittliche Laufzeit	0.153849	0.26426	0.194738	0.120147	0.103203	0.0958263	0.119553	0.212514
128^3								
Kürzeste Laufzeit	1.43068	1.92165	1.70996	1.25025	1.006	0.48835	0.443295	0.64211
Längste Laufzeit	1.45468	2.08845	1.75962	1.36799	1.15117	0.581386	0.460631	0.666447
Durchschnittliche Laufzeit	1.43844	2.01558	1.73688	1.3023	1.07886	0.526862	0.447222	0.652007
256^3								
Kürzeste Laufzeit	13.1084	18.1556	13.9171	9.92907	8.06601	5.1022	2.9927	2.11564
Längste Laufzeit	13.2957	19.2244	14.4287	10.1187	8.44148	5.49868	3.20723	2.16667
Durchschnittliche Laufzeit	13.1993	18.7392	14.2315	10.0126	8.21756	5.26811	3.144	2.14392
512^3								
Kürzeste Laufzeit	171.378	286.066	170.483	102.549	77.9262	47.2495	28.6178	17.5014
Längste Laufzeit	171.651	300.969	171.842	103.645	79.3706	48.9718	30.7482	18.3959
Durchschnittliche Laufzeit	171.458	293.053	171.156	103.187	78.7298	48.0174	29.3318	17.8061

Tabelle A.1.: Absolute gemessene Laufzeiten

Abbildungsverzeichnis

Liste der Algorithmen

Tabellenverzeichnis

Literaturverzeichnis

[1] J A Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[2] Jianming Yang and Frederick Stern. A highly scalable massively parallel fast marching method for the eikonal equation. *Journal of Computational Physics*, 332:333–362, 2017.

[3] Lawrence C. Evans. *Partial differential equations*. American Mathematical Society, Providence, R.I., 2010.

[4] Federica Dragoni. Introduction to viscosity solutions for nonlinear pdes, 2012. http://numerik.mi.fu-berlin.de/wiki/WS_2012/Vorlesungen/NumerikIV_Dokumente/DragoniViscositySolutions.pdf.

[5] Michael G. Crandall and Pierre-Louis Lions. Viscosity solutions of hamilton-jacobi equations. *Transactions of the American Mathematical Society*, 277(1):1–1, January 1983.

[6] Hitoshi Ishii. On the equivalence of two notions of weak solutions, viscosity solutions and distribution solutions. *Funkcialaj Ekvacioj*, 38:101–120, 1995.

[7] Werner H. Schmidt, Knut Heier, Leonhard Bittner, and Roland Bulirsch, editors. *Variational Calculus, Optimal Control and Applications*. Birkhäuser Basel, 1998.

[8] Elisabeth Rouy and Agnès Tourin. A viscosity solutions approach to shape-from-shading. *SIAM Journal on Numerical Analysis*, 29(3):867–884, 1992.

[9] Daniel Ganellari, Gundolf Haase, and Gerhard Zumbusch. A massively parallel eikonal solver on unstructured meshes. *Computing and Visualization in Science*, 19(5-6):3–18, February 2018.

[10] Hongkai Zhao. A fast sweeping method for eikonal equations. *Mathematics of Computation*, 74(250):603–628, May 2004.

[11] Won-Ki Jeong and Ross T. Whitaker. A fast iterative method for eikonal equations. *SIAM Journal on Scientific Computing*, 30(5):2512–2534, January 2008.

[12] Tor Gillberg, Are Bruaset, Øyvind Hjelle, and Mohammed Sourouri. Parallel solutions of static hamilton-jacobi equations for simulations of geological folds. *Journal of Mathematics in Industry*, 4(1):10, 2014.

[13] Shanti Bhushan, Pablo Carrica, Jianming Yang, and Frederick Stern. Scalability studies and large grid computations for surface combatant using CFDShip-iowa. *The International Journal of High Performance Computing Applications*, 25(4):466–487, February 2011.

[14] M Herrmann. A domain decomposition parallelization of the fast marching method. *Center for Turbulence Research Annual Research Briefs*, pages 213–225, 2003.

[15] Sumin Hong and Won-Ki Jeong. A multi-gpu fast iterative method for eikonal equations using on-the-fly adaptive domain decomposition. *Procedia Computer Science*, 80:190–200, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[16] Hongkai Zhao. Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics*, 25(4):421–429, 2007.

[17] Adam Chacon and Alexander Vladimirsky. A parallel heap-cell method for eikonal equations, 2013.

[18] http://sebastien.godard.pagesperso-orange.fr/.

[19] Robert Endre Tarjan. *3. Heaps*, pages 33–43. CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.