Parallel Computing Projekt

Anne Schulte-Kroll

Januar 2021

Inhaltsverzeichnis

1 Einleitung und Problemstellung **3**

2 Stencil Codes **4**

3 Implementierung **5**

4 Evaluation **8**

Literatur **11**

1 Einleitung und Problemstellung

Viele Probleme bestehen aus einer Reihe von Punkten in einem Gitter (im Folgenden auch Matrix genannt), die in aufeinanderfolgenden Iterationen basierend auf den Werten ihrer Nachbarn im selben Gitter aktualisiert werden. Diese Probleme können geometrisch in Blöcke unterteilt werden, die auf verschiedenen Prozessoren berechnet werden[1]. Da für die Berechnung jedes neuen Punkts jedoch die Werte anderer Punkte erforderlich sind, sind diese Berechnungen nicht gänzlich unabhängig voneinander möglich. Insbesondere erfordern die Punkte an den Rändern eines Blocks die Werte von Punkten aus den benachbarten Blöcken [1]. Die allgemeine Fragestellung des Projekts lautet daher: Wie können diese Werte effizient und strukturiert zwischen Prozessen kommuniziert werden?

Die vorliegende Projektarbeit behandelt hierzu die Anwendung der Wärmeleitungsgleichung unter der Verwendung von MPI (Message Passiong Interface) auf ein Gitter. Dabei liegt der Fokus auf der Parallelisierung des Vorgehens. Diese Anwendung stellt ein Beispiel dafür dar, wie große Berechnungen unterteilt werden können. Durch Parallelisierung kann die Gesamtlaufzeit einer solchen Berechnung deutlich verkürzt werden[2].

Die Anwendung der Wärmeleitungleichung ermöglicht die Hinzunahme von Stencil Codes, da die Berechnung als Faltungsproblem aufgefasst werden kann [3]. Es kann dazu ein Stencil zu definiert werden, welcher die Grundlage für die Neuberechnung der Matrix bildet. Dieser ist aufgrund der Wärmeleitungsgleichung ein 5-Punkt-Stencil (siehe (a) in Abbildung 1). Ziel des Projekts ist es, die Differentialgleichung durch ein paralleles Programm zu implementieren, welches die Temperaturen mit einer beliebigen Anzahl von Prozessen unter Verwendung von MPI berechnet.

Die Grundidee besteht darin, das Gitter in Blöcke zu unterteilen und jeden Prozess einen davon aktualisieren zu lassen. Wie bereits erläutert, besteht ein Problem bei diesem Ansatz darin, die Werte an den Grenzen zwischen Blöcken zu berechnen, da diese Werte von einem oder mehreren benachbarten Blöcken erfordern. Das Abrufen der erforderlichen Punkte aus dem Prozess, der den benachbarten Block berechnet, ist dabei keine gute Lösung, da dies zu vielen kleinen Kommunikationsvorgängen mitten in der Berechnung führen würde, was in hohen Latenzkosten resultieren würde [1].

Als Lösungsansatz zu dieser Problemstellung wird bei der Unterteilung des Gitters zusätzlicher Platz für eine Reihe von Geisterzellen an den Rändern jedes Blocks zugewiesen. Bei jeder Iteration tauschen benachbarte Prozesse ihre Randbereiche aus und die empfangenen Werte werden in den Geisterzellen platziert. Die Geisterzellen bilden somit einen Randbereich um jeden Block, der die Werte der Grenzen aller unmittelbaren Nachbarn enthält. Diese Werte in den Geisterzellen dienen dabei dazu, die notwendigen Werte zur Anwendung des Stencils in den Randbereichen eines Blocks zur Verfügung zu stellen (Abbildung 2).

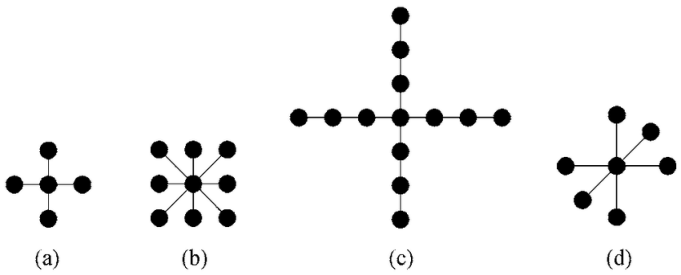

Abbildung 1: Verschiedene Arten von Stencil Codes [4]

2 Stencil Codes

Die Menge benachbarter Punkte, die die Berechnung eines Punktes beeinflussen, wird oft als Stencil bezeichnet. Der Stencil definiert, wie der Wert eines Punktes aus seinem eigenen Wert und den umliegenden Werten berechnet werden soll[3]. Ein solcher Stencil kann viele Formen annehmen und auch Punkte enthalten, die nicht direkt neben dem aktuellen Punkt liegen[1]. Abbildung 1 (a) zeigt einen Fünf-Punkte-Laplace-Operator, bei dem es sich um einen Stencil handelt, mit dem beispielsweise Kanten in einem Bild gefunden werden können. Der Stencil gibt an, dass der Wert eines Punktes in der aktuellen Iteration unter Verwendung der Werte seiner linken, rechten, oberen und unteren Nachbarn aus der vorherigen Iteration berechnet wird [3].

Ein Stencil ist somit eine stilisierte Matrixberechnung, welche einen Satz benachbarter Datenelemente kombiniert, um einen neuen Wert zu berechnen, sodass der Wert jedes Gitterpunkts basierend auf dem Wert des benachbarten Punkts geändert wird [1]. Dies wird häufig in vielen Iterationen ausgeführt, wodurch die Gitterpunkte wiederholt nach einem konstanten Schema geändert werden [2] . Diese Art der Berechnung wird sehr häufig zum Lösen partieller Differentialgleichungen, zur Bildverarbeitung und zur geometrischen Modellierung genutzt. Der Stencil Code aktualisiert alle Array-Elemente in jedem Zeitschritt [3] . In Abbildung 1 sind verschiedene Stencil Codes abgebildet, darunter auch ein 5-Punkt Stencil (a) wie er in diesem Projekt angewendet wird.

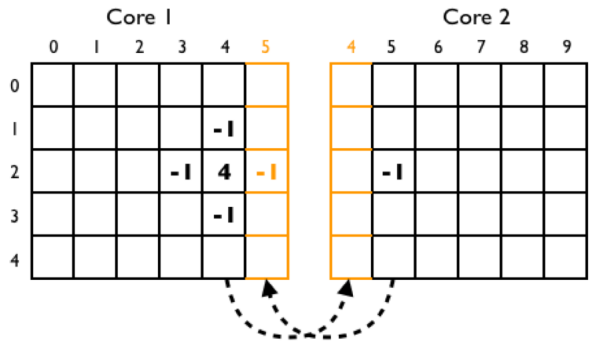

Abbildung 2: Jeder Block empfängt einen Vektor von Geisterzellen von benachbarten Blöcken[1]

3 Implementierung

In der im folgenden beschriebenen Implementierung wird als Anfangsmatrix von einer quadratischen N x N Matrix ausgegangen. Für diese können beim ausführen des Codes Randwerte und Anfangswerte gewählt werden. Als Startgitter wurde für die Auswertung in der vorliegenden Ausarbeitung eine Matrix mit den Randwerten 20 und -20 gewählt. Die Anfangsbedingung aller übrigen Werte ist dabei 0. Es ergibt sich dadurch eine N x N Matrix bei der alle Einträge, mit Ausnahme des Randes null sind.

Die Verwendung von MPI (Message Passing Interface) ermöglicht es, die Berechnung auf mehrere Prozesse aufzuteilen[5]. Dazu wird die Anfangsmatrix in gleich große Blöcke unterteilt, deren Anzahl der Anzahl der gwünschten Prozesse entspricht. Somit kann jedem Prozess ein Teil der Matrix zur Berechnung zugewiesen werden. Die Anzahl der Prozesse kann beim Start des Programms ebenfalls angegeben werden.

Da bei der Berechnung der neuen Gitterpunkte eines, einem Prozess zugewiesenen, Teilgitters allerdings auch Gitterpunkte benötigt werden, die außerhalb von diesem liegen, werden sogenannte Geisterzellen verwendet. Ein Prozess bekommt somit zusätzlich zu seinem Teilgitter auch umliegende Gitterpunkte übertragen. Bei einer Geisterzellenbreite von eins würde ein Prozess ein (N+1)x(N+1) Gitter zugewiesen bekommen, wobei N die eigentliche Größe seines zu berechnenden Teilgitters darstellt. Dabei muss berücksichtigt werden, dass außenliegende Teilgitter der Matrix nicht in jeder Dimension Geisterzellen haben. In Abbildung 2 stellt die gelbe Zone den Bereich dar, in dem die Daten der Prozesse sich überlappen müssen, damit eine Berechnung der Gitterpunkte möglich ist.

Ohne Geisterzellen müsste jede Zelle an der Grenze eines Gitters ihre eigene

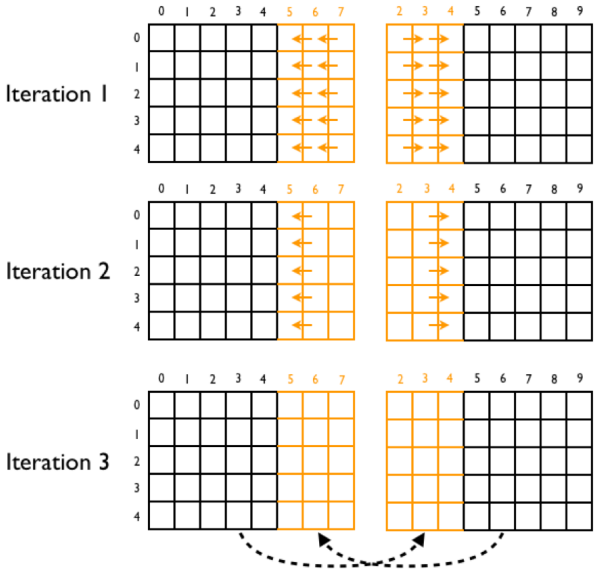

Abbildung 3: Datenautausch jede n-te Iteration [1]

Nachricht an benachbarte Prozesse versenden. Die Verwendung von Geisterzellen ermöglicht es, die Anzahl der Nachrichten zu minimieren, da viele Zellen, die zu den Gittergrenzen eines Prozesses gehören, gleichzeitig mit einer einzelnen Nachricht ausgetauscht werden können. So müssen, wenn jeder Prozess zur Berechnung eines Punktes die anliegenden Punkte (Abstand von 1) benötigt, bei einer Geisterzellenbreite von 1, die Prozesse bereits nach einer Iteration miteinander kommunizieren, wohingegen dies bei einer Geisterzellenbreite von 10 erst nach 10 Iterationen nötig wäre. In der hier beschriebenen Implementierung kann bei der Ausführung des Programms, die Geisterzellenbreite, als auch die Anzahl der Iterationen gewählt werden. Dabei ist zu beachten, dass in dieser Implementierung die wählbare Anzahl der Iterationen multipliziert mit der Anzahl der Geisterzellen, die tatsächliche Anzahl der Iterationen darstellt, da die wählbare Anzahl der Iterationen, die Häufigkeit der Ausführung des Kommunikationsprozesses angibt.

Ausgangspunkt des implementierten Algorithmus stellt ein Hauptprozess dar, welcher das Gitter initialisiert und die Blöcke inklusive Geisterzellen, mit Ausnahme seines eigenen Blocks, zur Berechnung auf andere Prozesse verteilt. Die Anzahl der Berechnungen während eines Kommunikationschritts entspricht der Geisterzellenbreite. Nach der Berechnung werden die berechneten Gitter von den anderen Prozessen wieder an den Hauptprozess versendet und von diesem wieder zusammengefügt. Hierbei erfolgt die Kommunikation zwischen den Prozessen durch Punkt-zu-Punkt Kommunikation. Kollektive Kommunikation

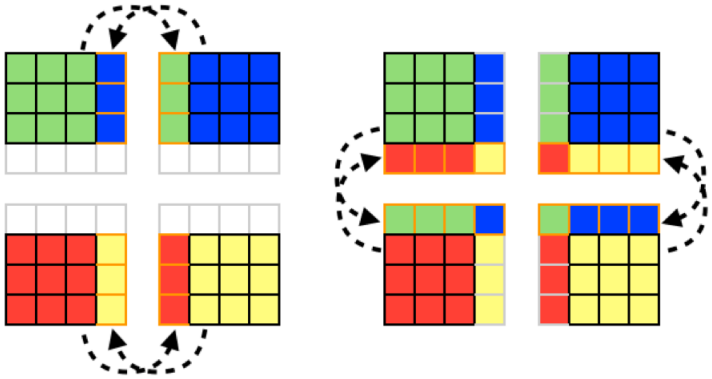

Abbildung 4: Zweidimensionaler Randaustausch [1]

anhand von Scatter und Gather ist hier nicht zum Einsatz gekommen. Beim Zusammenfügen der Blöcke werden die Geisterzellenbereiche wieder abgeschnitten. Das gesamte Vorgehen wird Iterationen/Geisterzellenbreite mal wiederholt.

Bei der Implementierung der Aufteilung des Gitters wurde unterschieden zwischen einer eindimensionalen Unterteilung des Gitters und einer zweidimensionalen. Bei der eindimensionalen Unterteilung wird die Matrix nur zeilenweise in Blöcke unterteilt, welche die Dimension (N/P)xN haben. Dabei stellt P die Anzahl der Prozesse dar. Der letzte Prozess bekommt dabei gegebenenfalls den Rest zugewiesen, falls N/P keine gerade Zahl ergibt.

Die zweidimensionale Implementierung unterteilt das Gitter zeilen- und spaltenweise. Eine Visualisierung des Vorgehens am Beispiel von 4 Prozessen stellt Abbildung 4 dar. In dieser Implementierung bekommen die unteren sowie die rechts liegenden Prozesse den Rest zugewiesen. Je nach Anzahl der Prozesse wird zur Berechnung ein eine der beiden Varianten der Unterteilung verwendet. Wenn die Anzahl der Prozesse eine quaratische Zahl darstellt, wird die zweidimensionale Implementierung verwendet, anderenfalls die eindimensionale.

Zur Implementierung der in der Aufgabenstellung beschriebenen Wärmeleitungsgleichung wurde die Programmiersprache Python mit den Bibliotheken mpi4py, numpy und scipy verwendet. Die Bibliothek scipy mit der Funktion signalconvolve2d wird zur Anwendung des Stencils auf das Gitter verwendet.

4 Evaluation

Bei der Evaluierung der Laufzeiten ist in Abbildung 5 zu erkennen, dass die Gesamtlaufzeit des Programms, wie zu erwarten, mit steigender Anzahl an Prozessen zurück geht und dabei scheinbar ab einer gewissen Anzahl an Prozessen gegen eine minimale Laufzeit konvergiert. Der Rückgang ist begründbar durch den höheren Anteil an parallelen Berechnungen bei einer steigenden Anzahl an Prozessen. Bei steigender Anzahl von Prozessen wird jedoch auch mehr Kommunikation notwendig, welche ebenfalls die Laufzeit erhöhen kann. Es scheint sich beispielsweise bei einer Matrix mit einer Größe von 10000x10000 ab ca. 100 Prozessen keine deutliche Verbesserung der Laufzeit mehr einzustellen. Dies könnte begründet werden, durch die vermehrte Kommunikation zwischen den Prozessen, deren Kosten den Nutzen der Parallelisierung relativieren.

Es ist zudem zu erkennen, dass die hier implementierte zweidimensionale Variante der Unterteilung des Gitters bei einer Geisterzellenbreite von 10 nicht effizienter zu sein scheint als die eindimensionale Implementierung und möglicherweise sogar leicht ineffizienter ist (siehe Abbildung 5). Dies könnte durch den deutlich komplexeren Algorithmus zur Unterteilung des Gitters begründet werden. Gegebenenfalls könnte diese Methode für Matrizen mit einem anderen Format als NxN effizienter sein als die eindimensionale Unterteilung, wenn durch die zeilenweise Unterteilung sehr schmale Blöcke entstehen. Ebenfalls könnte es durchaus möglich sein, die zweidimensionale Unterteilung des Gitters noch effizienter zu programmieren, sodass diese effizienter wird als eine eindimensionale.

Des Weiteren wird deutlich, dass sich die Gesamtlaufzeit des Programms mit steigender Geisterzellenanzahl zunächst verringert. Allerdings steigt diese beim Überschreiten eines optimalen Punks der Geisterzellenbreite wieder an. Bei einem Gitter von 10000x10000 und 32 bzw. 37 Prozessen mit 1D-Implementierung liegt dieser Punkt ungefähr bei einer Geisterzellenbreite von 10. Unter Verwendung der 2D-Implementierung hingegen scheint der optimale Punkt bei ca. 50 zu liegen. Dies lässt darauf schließen, dass der Effekt der Geisterzellenbreite auf die Effizienz von der Art der Gitteraufteilung abhängig ist. Diese Erkenntnisse erklären ebenfalls das Ergebnis von Abbildung 5, wo die 1D-Implementierung effizienter zu sein scheint. Zur Erstellung von Abbildung 5 wurde eine Geisterzellenbreite von 10 bei beiden Implementierungen gewählt. Hier scheint die 1D-Variante noch minimal effizienter zu sein, was mit Abbildung 6 übereinstimmt. Mit einer Geisterzellenbreite ¿ 50 ist die 2D-Implementierung daher, entgegen der obigen Interpretation von Abbildung 5, effizienter als die 1D-Implementierung.

Abbildung 7 veranschaulicht, dass bei einer Geisterzellenbreite von 50, die 2D-Implementierung effizienter ist als die 1D-Implementierung. Bemerkenswert ist hier auch, dass die 1D Implementierung mit 37 Prozessen ebenfalls schlechter abschneidet als die 2D Implementierung mit 36 Prozessen. Es könnte hier

ebenfalls interessant sein, zu evaluieren, ab welcher Anzahl an Prozessen bei der 1D-Implementierung ähnliche Werte wie bei der 2D-Implementierung erreicht werden können.

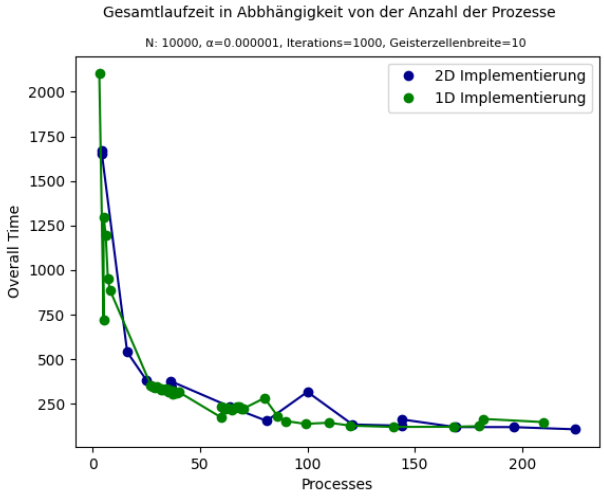

Abbildung 5: Einfluss der Anzahl der Prozesse auf die Laufzeit bei Geisterzellenbreite von 10

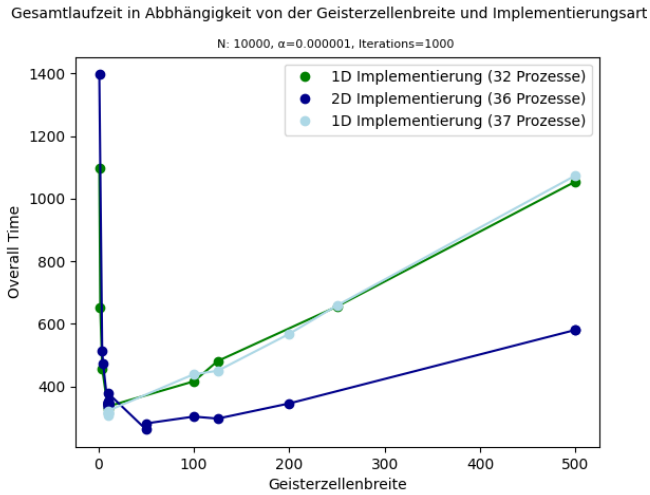

Abbildung 6: Einfluss der Geisterzellenbreite auf die Laufzeit

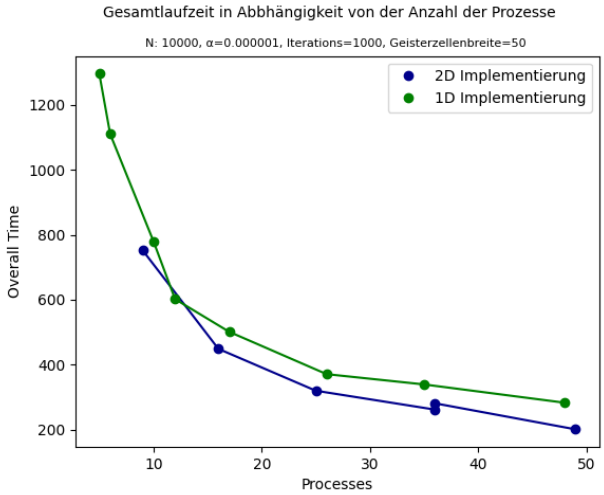

Abbildung 7: Einfluss der Anzahl der Prozesse auf die Laufzeit bei Geisterzellenbreite von 50

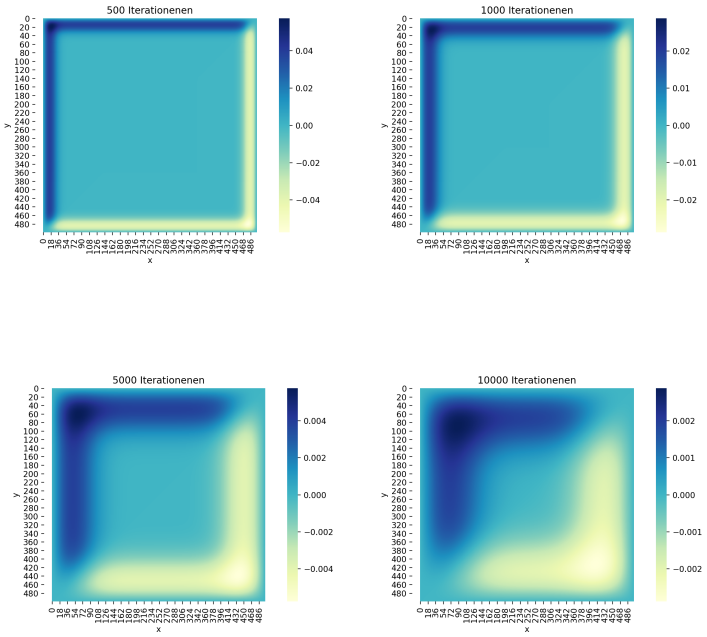

Abbildung 8: Gitterentwicklung eines 500x500 Gitters über 10000 Zeitschritte

Literatur

[1] Fredrik Berg Kjolstad and Marc Snir. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, pages 1–9, 2010.

[2] Peter MA Sloot, CJ Kenneth Tan, Jack J Dongarra, and Alfons G Hoekstra. *Computational Science—ICCS 2002: International Conference Amsterdam, The Netherlands, April 21–24, 2002 Proceedings*, volume 2331. Springer Science & Business Media, 2002.

[3] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerische Mathematik 2*. Springer-Verlag, 2013.

[4] Yang Yang, Huimin Cui, Xiaobing Feng, and Jingling Xue. A hybrid circular queue method for iterative stencil computations on gpus. *J. Comput. Sci. Technol.*, 27:57–74, 01 2012.

[5] David W Walker and Jack J Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.