Fachhochschule Aachen

Standort Jülich

Fachbereich 9: Medizintechnik und Technomathematik

Studiengang: Technomathematik

Ausarbeitung und Vergleich verschiedener Gitterverfeinerungsstrategien hinsichtlich der parallelen Performance eines Brandsimulationsprogramms mit adaptiver Gitterverfeinerung

Masterarbeit

von

Jana Boltersdorf

Mat.-Nr. 857156

Jülich, November 2017

Eigenständigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst worden.
Es sind keine anderen als die angegebenen Quellen und Hilfsmittel verwendet worden.

Jülich, den 9. November 2017

(Jana Boltersdorf)

Diese Arbeit wurde betreut von:

1. Prüfer: Prof. Dr. Johannes Grotendorst

2. Prüfer: Dr. Lukas Arnold

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Motivation

Numerische Simulationen überbrücken die Distanz zwischen Theorie und Experiment in der Wissenschaft. Wo Theorien abstrakt bleiben und Experimente oft nicht alle Zusammenhänge offenbaren, beschreiten sie einen Mittelweg und eröffnen damit ganz neue Möglichkeiten.
Sie erlauben es, theoretische Modelle mittels Computern praxisnah zu betrachten, die berechneten Ergebnisse mit den experimentell erlangten Beobachten zu vergleichen und zu visualisieren, um Zusammenhänge anschaulicher darzustellen. Dabei hat man im Gegensatz zu einem Experiment volle Kontrolle über alle Parameter und kann andere Einflussfaktoren ausschließen. Vor allem bei Parameterstudien in denen die Auswirkungen der Veränderung eines Wertes untersucht werden sollen, erlaubt dies es, die anderen Werte konstant zu halten, während bei Experimenten gewisse Schwankungen oft kaum vermeidbar sind. Über den Vorteil der erhöhten Aussagekraft hinaus stellen numerische Simulationen oft auch eine erhebliche Reduzierung von Kosten und Aufwand gegenüber herkömmlichen Experimenten dar. So ist es beispielsweise nicht mehr notwendig, zahlreiche Prototypen mit geringfügig unterschiedlichen Eigenschaften herzustellen und experimentell miteinander zu vergleichen. Durch numerische Simulationen können vergleichsweise kostengünstig diejenigen Parametersätze bestimmt werden, die die besten Eigenschaften versprechen und nur noch diese hergestellt und experimentell untersucht werden.
Sie ermöglichen sogar Betrachtungen, die mit herkömmlichen Experimenten kaum

möglich sind, beispielsweise weil sie auf atomarer Ebene abspielen oder in einer Zeitspanne von Jahrmillionen, etwa bei Galaxien. Numerische Simulationen erleichtern es immens, auch solche Vorgänge der wissenschaftlichen Betrachtung zugänglich zu machen. Obwohl es sich bei den numerischen Simulationen um einen recht jungen Ansatz handelt – Theorie und Experiment als wissenschaftliche Methoden wurzeln in der Antike – haben sie es dank ihrer weitreichenden Möglichkeiten rasch geschafft, zu einem zentralen Element wissenschaftlicher Praxis zu werden. Bereits heute sind sie aus vielen wissenschaftlichen Disziplinen nicht mehr weg zu denken, egal ob es darum geht, Proteinfaltungen zu simulieren, die Entstehung von Galaxien besser zu verstehen oder die Auswirkungen des Klimawandels vorher zu sagen.

In ihrer jungen Geschichte haben die numerischen Simulationen dabei eine beachtliche Entwicklung gezeigt, getrieben von den immer weiter wachsenden Ansprüchen der wissenschaftlichen Gemeinschaft, die die Möglichkeiten dieser neuen Methode ausschöpfen wollte. Die Simulationen sollten immer komplexere Vorgänge berechnen, mehr Einflussfaktoren berücksichtigen, sie sollten genauer werden... Kurz: Immer näher an die Realität heran rücken. Und dabei sollte die Dauer der Simulationen dennoch nicht unkontrolliert immer weiter wachsen. Dabei spielt nicht nur eine Rolle, dass eine wachsende Berechnungsdauer in der Regel auch wachsende Kosten bedeutet – es nützt wenig, das Wetter von morgen mit hoher Genauigkeit berechnen zu können, wenn man die Ergebnisse der Berechnung erst in einer Woche hat.
Vor allem anfangs konnten Verbesserungen der Hardware noch einen großen Teil des zusätzlichen Rechenaufwands abfangen indem sie mehr Berechnungen in der gleichen Zeit durchführen konnten. Letztlich schaffte sie es aber nicht, mit der Entwicklung der Simulationen Schritt zu halten. In Zukunft wird die Schere noch weiter auseinander gehen: Die Grenzen numerischer Simulationen sind noch lange nicht erreicht während insbesondere Prozessoren an die physikalischen Grenzen ihrer Leistungsfähigkeit stoßen.
Stattdessen besteht eine immer stärkere Notwendigkeit, die Art wie die Simulation

berechnet wird zu verändern und darauf hin zu optimieren, die .

Hierzu gibt es eine Vielzahl verschiedener Ansätze, die bei ganz unterschiedlichen Aspekten der Simulation ansetzen. Ob es um das mathematische Modell geht, dass das zu simulierende Problem beschreibt, die Diskretisierung mit der ein kontinuierliches Gebiet auf ein Gitter abgebildet wird, die Algorithmen der konkreten Implementierung oder die tatsächliche Berechnung... In allen diesen Teilbereichen der Simulation besteht Potential für Maßnahmen, die helfen können, die Berechnungsdauer der Simulation auf ein akzeptables Maß zu reduzieren.
Insofern ist es naheliegend, dass die wenigsten Simulationen sich auf einen dieser Aspekte beschränken und alle anderen unangetastet lassen. Zwar können nicht bei jeder Simulation ohne Weiteres alle Aspekte auf eine Laufzeitbegrenzung hin modifiziert werden, aber nicht zumindest die vorhandenen Möglichkeiten auszuschöpfen würde bedeuten, Zeit und andere Ressourcen zu verschwenden.

Oft werden die verschiedenen Verfahren zur Begrenzung der Laufzeit allerdings lediglich voneinander isoliert beschrieben und ihre Auswirkungen untersucht. Diese Vorgehensweise ist verständlich, wenn es darum geht, das Potential eines einzelnen Verfahrens zur Verbesserung der Laufzeit gegenüber der ursprünglichen Simulation zu ermitteln.
In der Praxis können verschiedene Verfahren zur Laufzeitreduzierung einander beeinflussen, indem sie die Ausgangsbedingungen für andere Verfahren verändern. Dies ist der Punkt an dem diese Arbeit ansetzt: Am Beispiel eines Brandsimulationsprogrammes werden die Auswirkungen zweier verschiedener Verfahren zur Reduzierung der Laufzeit – adaptive Gitterverfeinerung und MPI-Parallelisierung – untersucht und dabei insbesondere das Zusammenspiel der beiden Methoden betrachtet.

Adaptive Gitterverfeinerung verändert das Gitter auf dem die Simulation berechnet wird. Ihr zugrunde liegt der Gedanke, dass viele Simulationen sowohl eher grobe als auch sehr feine Strukturen beinhalten, die sich darüber hinaus im Ver-

lauf der Simulationen innerhalb des Gitters bewegen können.
Während die feinen Strukturen ein fein aufgelöstes Gitter benötigen, um mit guter Genauigkeit berechnet werden zu können, sind die gröberen Strukturen weniger anfällig für Ungenauigkeiten und tolerieren auch ein gröberes Gitter.
Der herkömmliche Ansatz wäre hier, sich an den feinen Strukturen zu orientieren und das komplette Gitter entsprechend ihrer Ansprüche sehr fein aufzulösen. Dieses Vorgehen würde aber im Gegenzug bedeuten, dass weite Bereiche des Gitters sehr viel feiner aufgelöst werden würden als die dortigen Strukturen es eigentlich verlangen. Während dies der Genauigkeit nicht schadet, treibt es die Berechnungsdauer und damit auch die Kosten der Simulation unnötig in die Höhe.
Ein adaptives Gitter hingegen kann während der Laufzeit der Simulation dynamisch Teile des Gitters feiner auflösen, wo es notwendig ist... Und wieder vergröbern, wenn eine feine Struktur sich aufgelöst oder in einen anderen Teil des Gitters gewandert ist. Das Gitter wird also immer nur so fein aufgelöst wie es die Strukturen in diesem Teil des Gitters tatsächlich verlangen. Dieser Ansatz schafft es, mit minimalen Genauigkeitsverlusten eine erhebliche Laufzeitreduzierung zu erzielen.

Einen ganz andere Ansatz verfolgt die MPI-Parallelisierung. Der stagnierenden Leistungsfähigkeit der Hardware tritt sie entgegen, indem mehr davon genutzt wird. Das zu simulierende Problem wird in kleinere Teilprobleme aufgeteilt, die parallel zueinander von jeweils einem Prozessorkern berechnet werden. Die Kerne tauschen anschließend Nachrichten mit ihren Ergebnissen untereinander aus und ermitteln so die Lösung des eigentlich simulierten Problem.
Im Gegensatz zu den meisten andere Verfahren wird hier keine Reduzierung der Berechnungskosten angestrebt sondern lediglich der Rechenaufwand auf mehrere Prozessorkerne aufgeteilt und so die Laufzeit des Programms reduziert. Die Kosten bleiben also effektiv gleich oder erhöhen sich sogar durch den zusätzlichen Kommunikationsaufwand zwischen den Kernen.
Die Stärke der Methode sind allerdings die guten Beschleunigungsfaktoren, die damit erreicht werden können. Kaum eine andere Methode kann vergleichbare

Werte erzielen. Hinzu kommt, dass Parallelisierungen nur vergleichsweise geringe Änderungen am Quellcode verlangen und bereits von vielen Bibliotheken unterstützt werden. Mittlerweile gibt es kaum noch größere Simulationssoftware, die nicht zumindest eine optionale Parallelisierung anbietet. Es liegt also nahe, sie in der Betrachtung des Zusammenspiels mehrerer Methoden der Laufzeitreduzierung einzubeziehen.

Dabei ist die Kombination von Parallelität und adaptiver Gitterverfeinerung von besonderem Interesse: Wichtig für eine effektive Parallelisierung ist, dass das simulierte Problem gut auf die verschiedenen Prozessorkerne aufgeteilt werden kann. Ideal ist es, wenn die Arbeitslast für alle Kerne annähernd gleich ist und möglichst wenig Kommunikation zwischen den Kernen notwendig ist.
Adaptive Gitterverfeinerung bedeutet aber, dass das Gitter sich theoretisch mit jedem Simulationsschritt verändern kann und damit wieder neu zur Berechnung auf die Prozessorkerne aufgeteilt werden muss. Der daraus entstehende Kommunikationaufwand und die Frage, wie gut eine gleichmäßige Lastverteilung zwischen den Kernen gelingt, sind essentiell für den Erfolg der Parallelisierung.

Zuerst stellt Kapitel 2 den Prozess vor, mit dem aus einer naturwissenschaftlichen Problemstellung eine numerische Simulation entsteht. Dabei werden kurz weitere Methoden angerissen, die zur Begrenzung der Laufzeit von Brandsimulationen genutzt werden. Dies geschieht anhand des Beispiels eines 2D-Vortex-Problems, wie es zur Verifikation von Brandsimulationen genutzt wird. Da dieses Problem auch für die Untersuchungen zu adaptiver Gitterverfeinerung und Parallelität heran gezogen wird, werden hier die theoretischen Grundlagen für diese Betrachtungen gelegt.
Anschließend auf JuFire als Beispiel eines Brandsimulationsprogrammes eingegangen und insbesondere die Implementierung von adaptiver Gitterverfeinerung und Parallelisierung thematisiert.

In Kapitel 3 wird genauer auf die adaptive Gitterverfeinerung eingegangen. Dabei

wird zuerst ihre Funktionsweise erklärt. Im Folgenden werden die im Rahmen JuFires zur Verfügung stehenden Verfeinerungsstrategien beschrieben, die festlegen welche Gitterzellen verfeinert und vergröbert werden sollen. Dabei handelt es sich um globale Verfeinerung, die Fixed-Fraction- und Fixed-Number-Strategie sowie eine Verfeinerungstrategie, die *ursprüngliche Strategie* genannt wird, da es sich bei ihr um die die erste in JuFire implementierte Strategie neben der globalen Verfeinerung handelt. Diese Strategien werden abschließend anhand des 2D-Vortex-Problems im Hinblick auf Genauigkeit und Laufzeit miteinander verglichen.

Kapitel 4 beschreibt grundsätzliche Paradigmen der Parallelprogrammierung und erklärt welche davon im Rahmen von JuFire zum Einsatz kommen. Darüber hinaus wird kurz das JURECA-Cluster vorgestellt, auf dem sämtliche in der Arbeit ausgewertete Simulationen berechnet wurden.
Der Speedup als beliebte Metrik zur Beschreibung der parallelen Leistung eines Programms wird beschrieben und sowohl ohne als auch mit aktivierter adaptiver Gitterverfeinerung auf JuFire angewendet. Der dabei festgestellte Leistungsabfall mit aktivierter adaptiver Gitterverfeinerung wird unter Zuhilfenahme von Amdahls Gesetz erklärt.
Des Weiteren wird mit Scalasca ein Programm zur Leistungsanalyse paralleler Programme vorgestellt und auf die von JuFire berechneten Simulationen mit aktiviertem AMR angewendet.

Zuletzt erfolgt in Kapitel 5 eine kurze Zusammenfassung der im Rahmen der Arbeit gewonnenen Erkenntnisse sowie ein Ausblick über mögliche Ergänzungen des JuFire-Codes im Bezug auf adaptive Gitterverfeinerung, die leider nicht mehr vorgenommen werden konnten, sowie zusätzliche Betrachtungen, die weitere Aufschlüsse über das Zusammenspiel von adaptiver Gitterverfeinerung und parallelen Berechnungen liefern könnten.

Kapitel 2

Theoretische Grundlagen

Computersimulationen folgen einem charakteristischen Prozess, der ein konkretes Problem abstrahiert und schließlich in eine von Computern berechenbare Form bringt.
Die Abstraktion schafft dabei eine gewisse Unabhängigkeit von dem wissenschaftlichen Bereich aus dem das simulierte Problem ursprünglich stammt. Dies ermöglicht es beispielsweise Verfahren, die bei der Simulation von erregbaren Medien wie Herzgewebe entwickelt wurden, auf Simulationen anzuwenden, die sich mit La-Ola-Wellen in Fußballstadien befassen. Die simulierten Probleme sind auf konkreter Ebene unterschiedlich, haben in der Abstraktion aber viele Ähnlichkeiten, so dass viele Erkenntnisse übertragen werden können. [FHV02]

In diesem Kapitel wird der Prozess am Beispiel einer Problemstellung aus dem Bereich der Rauchausbreitung – dem zweidimensionalen Vortex-Problem – demonstriert, welche mit numerischen Methoden behandelt wird. In den folgenden Kapiteln wird dabei auf die am Ende dieses Kapitel entwickelte numerische Simulation des Problems zurück gegriffen um die Auswirkungen von adaptiver Gitterverfeinerung und MPI-Parallelisierung zu untersuchen.
Das von CERFACS entwickelte 2D-Vortex-Problem wurde dabei gewählt, da es sich um ein Benchmark handelt, das etwa vom *Fire Dynamics Simulator*, einer Standard-Software für Brandsimulationen, zur Verifikation des Transportalgorithmus' verwendet wird. [Jou10] [MHM+17]

Die Lösung ist analytisch stabil, wandert aber periodisch durch das Simulationsgebiet. Diese Eigenschaft verlangt von der adaptiven Gitterverfeinerung, das Simulationsgitter kontinuierlich anzupassen um die Bewegung korrekt abzubilden und der Entstehung von Fehlern entgegen zu wirken.

2.1 Mathematisches Modell

Ein mathematisches Modell vereinfacht und abstrahiert einen konkreten Ausschnitt der beobachtbaren Welt, indem es einen ihn mit seinen Zusammenhängen und Bedingungen durch Gleichungssysteme mathematisch beschreibt. Dieses Vorgehen erlaubt es, Vorhersagen über das Verhalten des Problems unter bestimmten Bedingungen zu treffen, indem man die Parameter und Variablen des Modells entsprechend wählt und die Gleichungssysteme nach dem gesuchten Wert löst. Ein typisches Beispiel hierfür wäre es, einen Wert wie Temperatur oder Geschwindigkeit für einen gegebenen Raum- und Zeitpunkt zu ermitteln.
Dabei ist zu berücksichtigen, dass mathematische Modelle stets eine vereinfache Darstellung der Realität sind und somit in der Regel nur Teilaspekte eines Systems betrachten können. Wie realitätsgetreu das Modell diese Aspekte abbildet muss zudem durch Messungen und Vergleiche überprüft werden.
Es existieren zahlreiche Klassen von Gleichungen, die zur Modellierung von Problemen genutzt werden können. Dabei bestimmt das zu betrachtende Problem, welche Gleichungsklassen sich eignen.
Eine Klasse, die sich insbesondere bei der Modellierung physikalischer Probleme als äußerst wichtig erwiesen hat, sind die partiellen Differentialgleichungen. Im Gegensatz zu gewöhnlichen Differentialgleichungen, die lediglich Ableitungen nach einer Variablen beinhalten, beschreiben sie die Veränderung einer Größe durch mehrere voneinander unabhängige Variablen und enthalten somit partielle Ableitungen. Diese Eigenschaften erlauben es ihnen, auch vergleichsweise komplexe Systeme mit mehreren Einflussfaktoren zu modellieren.[Eij14]

Im Fall des zweidimensionalen Vortex-Problems wird ein Strömungsfeld betrach-

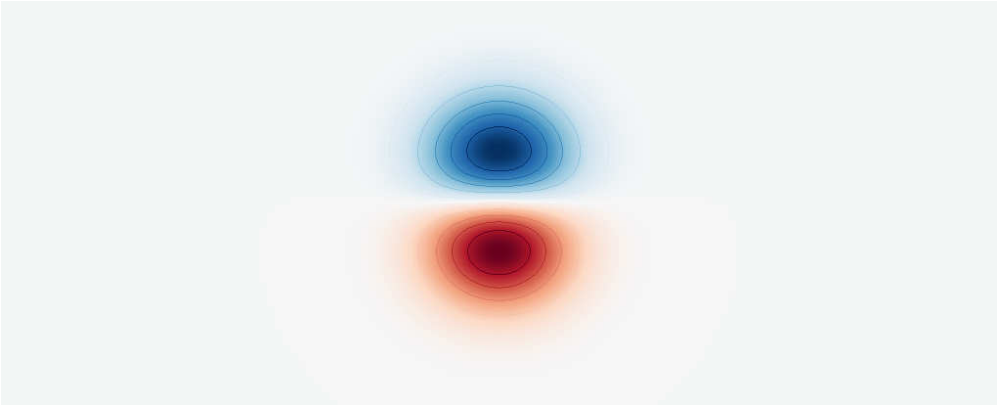

Abbildung 2.1: Das zweidimensionale Vortex-Problem wird zur Verifikation des Transport-Algorithmus' von FDS verwendet. Aufgrund des konstanten Strömungsfeldes und der periodischen Randbedingungen wandert der Vortex wiederholt durch das Rechengebiet. Dabei soll er seine Geometrie über die Zeit erhalten.

tet, das aus einem einzigen Wirbel besteht. Gespeist wird das Feld von einer uniformen Strömung.

Zur Modellierung wird daher auf grundlegende mathematische Modell der Strömungsmechanik zurück gegriffen: die Navier-Stokes-Gleichungen. Dieses nach Claude Louis Marie Henri Navier und George Gabriel Stokes benannte Modell erweitert die eulerschen Gleichungen der Strömungsmechanik um eine Beschreibung der Zähflüssigkeit der betrachteten Fluide. So ergibt sich ein System nichtlinearer partieller Differentialgleichungen 2. Ordnung. [CM00]

Im Fall des 2D-Vortex wird von einem nicht kompressiblen Fluid ausgegangen. Diese Einschränkung erlaubt Vereinfachungen, die in dem folgenden Gleichungssystem resultieren:

$$\begin{aligned} \nabla \cdot \vec{u} &= 0 \\ \vec{f}(T) &= \rho\left(\partial_t \vec{u} + (\vec{u} \cdot \nabla)\,\vec{u}\right) + \nabla p - \nabla\left(2\mu\epsilon_{ij}\left(\vec{u}\right)\right) \\ \gamma &= \rho\left(\partial_t T + (\vec{u} \cdot \nabla)\,T\right) - 2\tfrac{\mu}{c_p}\epsilon_{ij}\left(\vec{u}\right)\,\nabla\vec{u} - \nabla \cdot \left(\tfrac{\mu}{Pr}\nabla T\right) \end{aligned}$$

Dabei ist ϵ_{ij} der Verzerrungstensor

$$\epsilon_{ij}\left(\vec{v}\right) = \frac{1}{2}\left(\nabla\vec{v} + \left(\nabla\vec{v}\right)^T\right).$$

$\vec{v}$ ist die Geschwindigkeit der Strömung, p der physikalische Druck, μ die dynamische Viskosität, $\vec{f}(T)$ eine Volumenkraft. [FBA17]

Der Vortex selbst wird definiert als Gradient des Potentialfeldes

$$\Psi_0(x,y) = \Gamma \exp\left[-\frac{x^2+y^2}{2R_c^2}\right]$$

auf einem quadratischen Rechengebiet mit Kantenlänge $L = 0.3112$ m, periodischen Randbedingung und einer Geschwindigkeit des konstanten Strömungsfelds $U_0 = 35$ m/s. Dabei sind die charakteristische Größe $R_c^2 = L/20 = 0.01556$ und die Intensität des Feldes $\Gamma = 0.04 U_0 R_c \sqrt{e} = 0.0359157$ durch den *FDS Verification Guide* vorgegeben.
Die zu berechnende Größen sind die Geschwindigkeitskomponenten, die bestimmt werden können, indem man das konstante Strömungsfeld der Geschwindigkeit U_0 in positiver x-Richtung mit dem Gradienten des Strömungsfeldes überlagert. Diese werden beschrieben durch

$$\begin{array}{rcccl} u(x,y) & \equiv & U_0 + \partial_y \Psi_0 & = & U_0 - \frac{\Gamma_y}{R_c^2} \exp\left[-\frac{x^2+y^2}{2R_c^2}\right] \\ w(x,y) & \equiv & -\partial_x \Psi_0 & = & \frac{\Gamma_x}{R_c^2} \exp\left[-\frac{x^2+y^2}{2R_c^2}\right] \end{array}$$

wobei u die Geschwindigkeit in x-Richtung und w die Geschwindigkeit in y-Richtung darstellt. Da hier die Betrachtung einer Geschwindigkeitskomponente ausreicht, wird hier ohne Beschränkung der Allgemeinheit lediglich u untersucht.

2.2 Diskretisierung

Exakte analytische Lösungen für die mathematischen Modelle sind nur in wenigen Fällen bekannt. In der Regel muss man sich damit zufrieden geben, die Lösung mit numerischen Methoden zu approximieren.
Hierbei ist es notwendig, das kontinuierliche mathematische Modell durch eine endliche Anzahl Punkte in Raum und Zeit zu diskretisieren. Die Menge dieser Punkte wird Gitter genannt. Der Schritt ist notwendig, um eine Betrachtung des

Problems in endlicher Zeit und mit endlichem Speicherbedarf erst zu ermöglichen. Dabei gibt viele verschiedene Ansätze, unter welchen Gesichtspunkten man diese Diskretisierungspunkte wählen kann. Welche Methode für ein konkretes Problem am Besten geeignet ist hängt dabei stets von dessen spezifischen Eigenschaften und dem Ziel der Simulation ab.
Bekannte Beispiele sind insbesondere die Finite-Differenzen-Methode, die Finite-Volumen-Methode, die Finite-Elemente-Methode und die Randelement-Methode. JuFire verwendet die Finite-Elemente-Methode (FEM), eines der am weitesten verbreiteten Diskretisierungsmethoden. Sie zeichnet sich durch eine extrem hohe Flexibilität bei der Wahl der Rechengebiete aus und bietet beispielsweise die Möglichkeit auf unstrukturierten Gittern zu arbeiten.
Im Gegensatz zu einem strukturierten Gitter müssen die Diskretisierungspunkte hier nicht unbedingt vier Nachbarpunkte haben sondern können weitaus komplexere Nachbarschaftsbeziehungen besitzen. Daraus resultiert desweiteren, dass die Zellen nicht rechteckig sein müssen wie bei einigen simplen strukturierten Gittern sondern die Form eines beliebigen Vierecks annehmen können.
Diese Eigenschaften erlauben es, eine Struktur zu wählen, die das Rechengebiet besonders gut repräsentiert und vermeidet etwa die Entstehung von Treppenstrukturen wie sie bei strukturierten Gittern häufig zu beobachten sind, wenn beispielsweise Rundungen mit rechteckigen Zellen dargestellt werden sollen. Zwar kann auch ein unstrukturiertes Gitter ein rundes Rechengebiet nur approximieren, es kann ihm aber mit der gleichen Anzahl Diskretisierungspunkte wesentlich näher kommen als dies mit einem strukturierten Gitter möglich ist.
Der größte Nachteil der FEM besteht darin, dass die zugrunde liegenden Mathematik deutlich komplexer und die resultierenden Berechnungen aufwändiger sind als bei anderen, älteren Verfahren wie der Finite-Differenzen-Methode. Die FEM wurde daher erst mit dem Aufkommen von Computern nutzbar und entsprechend von Anfang an computergerecht formuliert. [Mat10]

Bei der Betrachtung des 2D-Vortex-Problems werden diese Eigenschaften jedoch nicht ausgenutzt. Der *FDS Verification Guide* schreibt ein quadratisches Rechen-

gebiet mit quadratischen Zellen vor. [MHM+17]

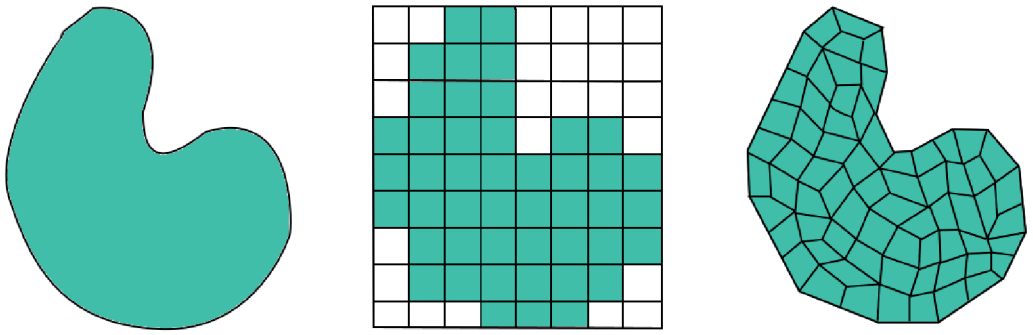

Abbildung 2.2: Ein kontinuierliches Rechengebiet (links) kann sowohl auf strukturierte (Mitte) als auch unstrukturierte Gitter (rechts) abgebildet werden. Diese Entscheidung ist maßgeblich für die Eigenschaften der numerischen Simulation.

Allen Gittern ist gemein, dass sie eine Granularität haben – ein Maß, das beschreibt wie weit die einzelnen Diskretisierungspunkte auseinander liegen. Je geringer die Abstände sind, kleiner werden die zwischen den Diskretisierungspunkten liegenden Zellen und desto genauer kann die Diskretisierung sich an das kontinuierliche Problem beschreiben. Allerdings kostet die Simulation eines Rechengebietes auch umso mehr Ressourcen je mehr Zellen es umfasst.
Dieser Aspekt ist für die Simulation der Rauchausbreitung von besonderer Relevanz: Sie beschreiben eine turbulente Strömung, die sowohl größere als auch sehr feine Wirbelstrukturen enthält, die möglichst genau abgebildet werden sollen. Dies kann auch rein über die numerische Berechnung der inkompressiblen Navier-Stokes-Gleichungen erreicht werden, erzwingt dann aber zur Auflösung der kleineren Wirbel ein sehr feines Gitter. Dieser Ansatz ist durch die damit einher gehenden hohen Berechnungskosten und den gesteigerten Zeitaufwand in der Praxis üblicherweise kaum umsetzbar. Stattdessen wird zumindest teilweise auf ein separates mathematisches Modell zur Abbildung der Wirbelstrukturen zurück gegriffen.

Es gibt verschiedene Wege, mit dieser Problematik umzugehen. Eine denkbare

Vorgehensweise ist, einen Schritt zurück in Abschnitt 2.1 zu den mathematischen Modellen zu machen und die Wirbelstrukturen über ein seperates Modell abzubilden.
Large-Eddy-Simulationen gehen diesen Weg. Die größen Wirbelstrukturen werden direkt numerisch berechnet, die kleineren Strukturen heraus gefiltert und über ein anderes Modell abgebildet. Dies bedeutet einen gewissen Genauigkeitsverlust gegenüber der vollständigen direkten numerischen Simulation, reduziert die Berechnungsdauer aber in einem Maße das die praktische Anwendung in Simulationen erlaubt.
JuFire verfügt zusätzlich zu der in dieser Arbeit schwerpunktmäßig betrachteten Gitterverfeinerung und der Parallelisierung auch über ein Smagorinsky-Lilly-Modell für Large-Eddy-Simulationen. Dabei handelt es sich um ein funktionelles Modell, das sich auf die Zerstreuung der Energie mit physikalisch korrekter Rate konzentriert. Im Kontrast dazu gibt es auch strukturelle Modelle, die diese Vereinfachung nicht vornehmen. [Sma63]

Ein anderer Ansatzpunkt ist die adaptive Gitterverfeinerung (AMR). Hierbei wird nicht das mathematische Modell selbst modifiziert sondern lediglich das Gitter auf dem dieses berechnet wird. AMR erlaubt es, auch während der Laufzeit verschiedene Bereiche des Gitters unterschiedlich fein aufzulösen. Diese Vorgehensweise ermöglicht es, relevante Gitterbereiche mit einer hohen Genauigkeit abzubilden und weniger relevante Bereiche gröber. Hierdurch können Ressourcen eingespart werden. [BO83]
Dies ist im Bereich der Rauchausbreitung von besonderem Interesse, da die relevanten Bereiche des Gitters sich im zeitlichen Verlauf der Simulation oft drastisch verändern etwa in dem sie sich durch das Gitter bewegen oder in ihrer Größe zunehmen. Dies macht es notwendig, insgesamt ein großes Gitter zu betrachten, auch wenn über einen Großteil der simulierten Zeit nur in einem kleinen Teil davon tatsächlich etwas berechnet werden muss.
Wie bereits zuvor erwähnt bildet adaptive Gitterverfeinerung einen der Schwerpunkte dieser Arbeit. Sie wird daher hier nicht weiter ausgeführt sondern in Ka-

pitel 3 detailliert betrachtet.

2.3 Implementierung

Es gibt verschiedene Möglichkeiten, das oben beschriebene Schema in Software zu übertragen, die auch nur ansatzweise zu erläutern den Rahmen dieser Arbeit massiv übersteigen würde. Stattdessen werde ich exemplarisch auf JuFire eingehen.

JuFire ist ein AMR-Framework, das auf der Basis der Finite-Elemente-Methode schwach kompressible Probleme aus dem Bereich von Brandsimulationen löst, bisher insbesondere solche die sich mit Rauchausbreitung befassen. Es wird in der Abteilung *Civil Security and Traffic* des Jülich Supercomputing Centers entwickelt.

Es ist in C++ geschrieben, einer ISO-genormten Multiparadigmen-Programmiersprache, die sowohl effiziente, maschinennahe Programmierung als auch ein hohes Abstraktionslevel erlaubt. Diese Vielseitigkeit kommt JuFire sehr entgegen: Es handelt sich um ein relativ umfangreiches Programm, das ohne einen objektorientierten Ansatz mit Kapselung der Klassen extrem unübersichtlich und schwer wartbar werden würde, als Simulationsprogramm aber auch möglichst schnell und effizient Berechnungen durchführen soll.
Unterstützt wird dies durch eine MPI-Parallelisierung. MPI (Message Passing Interface) ist ein Standard für den Nachrichtenaustausch bei parallelen Rechnungen. In Kapitel 4 werde ich genauer auf die Verwendung von MPI und ihre Implikationen eingehen.

Es gilt generell als gute Praxis in der Software-Entwicklung, vorhandene Bibliotheken zu nutzen statt sämtliche benötigte Funktionalität selbst zu implementieren. Diese Vorgehensweise bedeutet nicht nur oft eine erhebliche Zeitersparnis sondern hat auch qualitative Vorteile. Bibliotheken konzentrieren sich ganz auf eine klar abgegrenzte Aufgabe und bieten hoch optimierte, gut gewartete und doku-

mentierte Funktionen, um diese Aufgabe zu bewältigen. In vielen Bibliotheken stecken Jahre intensiver Entwicklung und Nutzertests, die Fehler nahezu ausgemerzt haben. [Ban13, Lecture 1]

Die wichtigste Bibliothek, auf die JuFire zurück greift ist Deal.II, eine C++-Bibliothek zum Lösen partieller Differentialgleichungen mithilfe von FEM. Sie wurde zuerst im Jahr 2000 veröffentlicht und seither in hunderten wissenschaftlicher Veröffentlichungen verwendet sowie mit einem Preis für numerische Software ausgezeichnet.

Deal.II unterstützt die dimensionsunabhängige Programmierung lokal adaptiver Gitter, bietet viele verschiedene Arten von finiten Elementen jegliche Ordnung, skaliert auch auf massiv parallelen Systemen noch gut und stellt zudem eine umfangreiche Dokumentation sowie Tutorials bereit, um neue Nutzer zu unterstützen. Deal.II greift dabei seinerseits auf existierende Bibliotheken wie beispielsweise PETSc, Trilinos, VTK und p4est zurück, um bestimmte Aufgaben zu übernehmen. [BHK07]

Kapitel 3

Adaptive Gitterverfeinerung

Adaptive Gitterverfeinerung, oft auf Basis der englischen Bezeichnung *adapative mesh refinement* zu AMR abgekürzt, beschreibt die lokale Verfeinerung von Simulationsgittern während der Laufzeit des Programms. Das Gitter hat also keine globale Feinheit, die die Größe sämtlicher Zellen des Gitters bestimmt sondern kann in verschiedenen Bereichen des Gitters unterschiedliche Feinheiten besitzen. Darüber hinaus ist es möglich, diese lokalen Feinheiten im Lauf der Simulation ohne Anwender-Einwirkung immer wieder anzupassen, damit sie auch weiterhin die simulierten Strukturen optimal abbilden können.
Es ist ein numerisches Verfahren, das auf Marsha Berger, Joseph Oliger und Phillip Colella zurück geht. Sie entwickelten in den 1980ern in einer Reihe von Papern, insbesondere mit Anwendungsgebieten aus dem Bereich der Hydrodynamik, einen Algorithmus zur Erzeugung dynamischer Gitter. Diesen bezeichneten sie als *local adaptive mesh refinement*. [BO83] [BC89]
Dieser Ansatz erwies sich als besonders gut geeignet für Probleme mit Unstetigkeiten, steilen Gradienten, Schocks und ähnlichen Eigenschaften – Strukturen, die lokal eine sehr hohe Auflösung des Gitters erzwingen. Im Kontrast dazu gibt es oft auch große Gitterbereiche, die im Bezug auf die Auflösung deutlich robuster sind. AMR ermöglicht es, diese unterschiedlichen Ansprüche zu in einem Gitter zu vereinen.
Heute findet AMR weiter Verbreitung in einer Vielzahl wissenschaftlicher Disziplinen wie beispielsweise Astrophysik, Klimamodellierung, Biophysik und Brand-

simulationen.

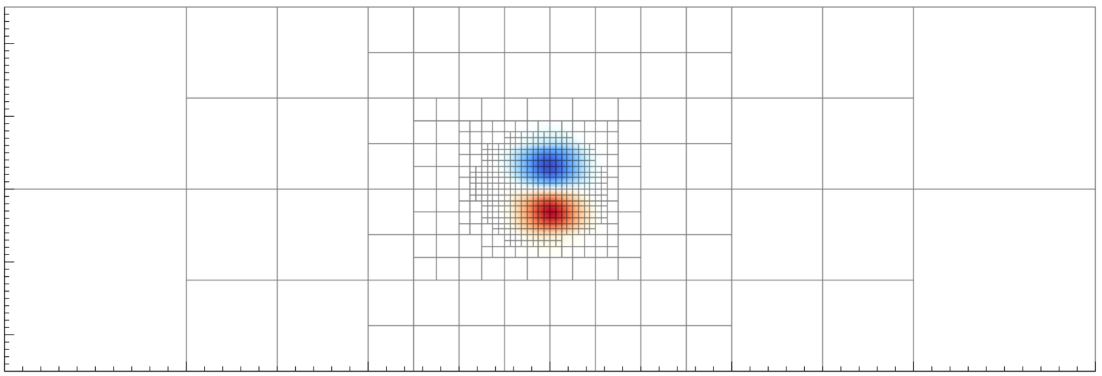

Abbildung 3.1: Adaptive Gitterverfeinerung ermöglicht es, unterschiedlich fein aufgelöste Bereiche innerhalb des Gitters einer Simulation zu haben. Dadurch können fehleranfällige Strukturen detailliert abgebildet und in robusteren Bereichen Ressourcen eingespart werden.

3.1 Funktionsweise

Eine Simulation mit adaptiver Gitterverfeinerung folgt einem klaren Ablauf, bestehend aus einer Startphase vor Beginn der eigentlichen Simulation sowie einem sich regelmäßigen Abständen wiederholenden Schleife währenddessen.

Das Startgitter, das vom Programm erzeugt wird, stellt das gröbste mögliche Gitter dar und wird daher als Verfeinerungslevel 0 bezeichnet. Dabei hat jede aktive Zelle ein eigenes Verfeinerungslevel. Bei jeder Verfeinerung wird die Zelle in jeder Dimension halbiert und das Verfeinerungslevel um 1 erhöht.
Das Startgitter ist in der Regel zu grob um darauf sinnvoll Berechnungen durchführen zu können. Aus diesem Grund muss das Gitter bereits vor Beginn der Simulation verfeinert werden.
Eine Möglichkeit dafür ist die Startphase. Dabei wird der Startwert auf das Gitter interpoliert wird und dieses entsprechend verfeinert. Dieser Schritt wird falls notwendig mehrere Male wiederholt und das resultierende Gitter als Startgitter

der Simulation verwendet. Diese Vorgehensweise bedeutet einen gewissen Zusatzaufwand, vermeidet aber, dass bereits zu Beginn der eigentlichen Simulation Ungenauigkeiten durch ein zu grobes Gitter entstehen.
Alternativ kann auch mit einem hoch verfeinerten Gitter gestartet werden, das während der ersten Simulationsschritte nach und nach vergröbert wird. Dieses Vorgehen hält den Fehler ebenfalls klein, bedeutet aber dass die ersten Simulationsschritte durch die hohe Anzahl von Gitterzellen einen deutlich höheren Berechnungsaufwand – und entsprechend auch höhere Kosten – haben als spätere Simulationsschritte in denen weniger relevante Rechengebiete bereits vergröbert wurden.

Unabhängig von dem gewählten Startgitter ist der Ablauf während der eigentlichen Simulation stets gleich. Er kann als Schleife verstanden werden, die alle n Simulationsschritte durchlaufen wird. Dies basiert auf der Annahme, dass das Gitter sich nicht in jedem Simulationsschritt so sehr ändert, dass es angepasst werden muss. Indem ein Gitter für mehrere aufeinander folgende Simulationsschritte verwendet wird, kann der durch AMR erzeugte Overhead erheblich reduziert werden. Im Folgenden wird stets der Fall $n = 1$ betrachtet, die Schleife also mit jedem Simulationsschritt durchlaufen.

1. **SOLVE-Phase**

   Dieser Schritt umfasst das numerische Lösen der Gleichungssysteme für einen Zeitschritt. Er findet genauso statt wie bei einer numerischen Simulation ohne AMR und wird hier entsprechend nicht weiter thematisiert.

2. **ESTIMATE-Phase**

   Der Fehler des eben berechneten Zeitschrittes wird für jede aktive Zelle des Gitters geschätzt. Da der genaue Fehler – und damit auch die exakte Lösung – in der Regel nicht bekannt sind, helfen Schätzungen, die Güte der Simulation zu beurteilen. Das verwendete Maß wird Verfeinerungsindikator genannt.

3. **MARK-Phase**

Nachdem der Verfeinerungsindikator für jede aktive Zelle ermittelt wurde, gilt es die Entscheidung zu treffen, welche Zellen verfeinert und vergröbert werden sollen. Das Kriterium, für welche geschätzten Fehlerwerte welche Modifikation des Gitters durchgeführt werden soll, nennt man auch Verfeinerungsstrategie. Einige Beispiele dafür werden in 3.2 vorgestellt. Die ausgewählten Zellen werden durch Flags markiert.

4. **REFINE-Phase**

   Dieser Schritt bereitet die eigentliche Verfeinerung vor und führt sie aus. Dabei werden gegebenenfalls noch einmal die gesetzten Flags modifiziert. Dies dient beispielsweise dazu, Bedingungen einzuhalten wie dass die Differenz der Verfeinerungslevel zweier benachbarter Zellen maximal 1 betragen darf. Zuletzt wird das Gitter tatsächlich modifiziert, die durch Flags angekündigten Änderungen umgesetzt und die Lösung auf das neue Gitter übertragen. [Ban13, Lecture 17.5]

3.2 Verfeinerungsstrategien

Im Rahmen der Arbeit wurde entschieden, den Fokus auf die verschiedenen Verfeinerungsstrategien zu legen. Aspekte wie die das Schätzen des Fehlers wurden daher zurück gestellt und nicht weiter betrachtet.
Alle betrachteten numerischen Simulationen beschränken sich aus diesem Grund auf den Kelly-Fehlerschätzer

$$\eta_K = h_K^{1/2} \left(\int_{\partial K} |[\nabla u_h]|^2 \right)^{1/2}$$

als Verfeinerungsindukator. Obwohl er eigentlich lediglich den Fehler des Laplace-Problems mit linearen Elementen schätzen kann, hat er sich in der Praxis dennoch als gute Annäherung für andere Elementordnungen sowie viele andere Probleme erwiesen und wird wegen seiner einfachen Anwendung gern genutzt. [Ban13, Lecture 17.25]

3.2.1 Globale Verfeinerung

Bei der globalen Verfeinerung handelt es sich nicht um eine Verfeinerungsstrategie im engeren Sinne. Es werden schlicht alle Zellen zur Verfeinerung markiert und anschließend verfeinert bis das maximale Verfeinerungslevel der Simulation ausgeschöpft ist. Ab diesem Punkt bleibt das Gitter konstant.
Die Vorteile dieser Strategie sind die garantierte Konvergenz und dass es nicht notwendig ist, den Verfeinerungsindikator überhaupt zu berechnen. Allerdings wird dieser Ansatz in der Regel weitaus mehr Zellen erfordern als eigentlich zur Berechnung des Problems mit der angestrebten Genauigkeit notwendig sind.

3.2.2 Fixed-Fraction-Strategie

Diese Strategie wird auch Bulk-Refinement-Strategie genannt. Ihr Grundgedanke ist es, diejenigen Zellen zu markieren, die den höchsten Verfeinerungsindikator haben und zusammen für einen bestimmten prozentualen Anteil des Gesamtfehlers verantwortlich sind.
Für diese Strategie kann gezeigt werden, dass sie zu optimalen Gittern führt, also nur so viele Zellen wie nötig verfeinert. Außerdem kann zumindest theoretisch Konvergenz garantiert werden. Problematisch werden können allerdings Singularitäten, bei denen nur sehr wenige Zellen für einen Großteil des Fehlers verantwortlich sind. In diesen Fällen können die Berechnungen sehr teuer werden.
JuFire wurde im Rahmen der Arbeit um diese Strategie erweitert.

3.2.3 Fixed-Number-Strategie

Ähnlich wie die Fixed-Fraction-Strategie markiert auch die Fixed-Number-Strategie Zellen mit dem höchsten Verfeinerungsindikator. Im Kontrast dazu markiert sie aber einen festen Anteil der gesamten aktiven Zellen statt derjenigen, die für einen Anteil des Gesamtfehlers verantwortlich sind.
Daraus resultiert der Vorteil, dass man die Anzahl der Gesamtzellen mit dieser Strategie gut kontrollieren kann. Dies erleichtert es wiederum, die Rechenkosten

der Simulation abzuschätzen. Es kann aber auch dazu führen, dass zu viele Zellen verfeinert werden und das Gitter damit abermals nicht optimal ist. [Ban13, Lecture 17.5]
JuFire wurde im Rahmen der Arbeit um diese Strategie erweitert.

3.2.4 Ursprüngliche Strategie

Diese Strategie hat bisher keinen Namen und wurde bereits im Rahmen der ursprünglichen Entwicklung von JuFire implementiert. Prinzipiell funktioniert diese Strategie wie Fixed-Fraction (3.2.2), hat aber die zusätzliche Bedingung, dass eine Verfeinerung nur dann durchgeführt wird, wenn ein Mindestradius der Zelle dadurch nicht unterschritten wird.
Die Abweichungen von der Fixed-Fraction-Strategie können so allerdings nicht vollständig erklärt werden (siehe 5.2).

3.3 Vergleich der Strategien

Um den Vergleich der verschiedenen Verfeinerungsstrategien zu ermöglichen wurde das gleiche Problem - das in Kapitel 2 beschriebene Wirbelbeispiel - jeweils einmal mit jeder der zu betrachtendenden Konfigurationen berechnet. Diese setzen sich zusammen aus der ursprünglichen Strategie, Fixed Fraction und Fixed Number mit jeweils 0.3 und 0.075 als Verfeinerungsschwellwert und 0.3 als Vergröberungsschwellwert. Hinzu kommt zum Vergleich eine Berechnung mit globaler Verfeinerung.
Alle Berechnungen simulieren das Problem für eine Zeiteinheit mit einer Schrittweite von $h_t = 10^{-3}$ und bis zu 6 Verfeinerungsleveln. Die Simulation nutzt keine Startphase sondern startet mit einem sechsfach global verfeinerten Gitter. Dies ist auch das maximale Verfeinerungslevel für die Simulation. Die in 3.1 beschriebene COMPUTE-ESTIMATE-MARK-REFINE-Schleife wird dabei mit jedem Zeitschritt durchlaufen. Es werden also insgesamt über den Lauf der Simulation 1000 Gitterverfeinerungen durchgeführt.

Die zwei wichtigsten Merkmale für die Bewertung einer Strategie sind ihre Auswirkungen auf die Genauigkeit der Simulation und die Kosten, die ihre Verwendung mit sich bringt.
Um die Genauigkeit zu betrachten ist es wichtig, zu bedenken dass das Problem in Raum und Zeit diskretisiert wurde und somit auch jeweils in diesen Dimensionen Diskretisierungsfehler aufweisen. Im Rahmen des AMR wird hier lediglich das räumliche Gitter betrachtet. Aus diesem Grund kann auch ein sehr feines Gitter den Fehler nicht unbegrenzt reduzieren. Als Referenzwert wird hier stattdessen das global verfeinerte Gitter mit 6 Verfeinerungsleveln heran gezogen, da dieses die maximale Genauigkeit unter den gegebenen Umständen (maximal 6 Verfeinerungslevel, fester Zeitschritt 10^{-3}) bietet.
Das verwendete Maß zur Bestimmung der Genauigkeit ist der L^2-Fehler

$$||e|| = \left(\sum_i e_i^2 \right)^{1/2}$$

wobei e_i ein Vektor mit den lokalen L_2-Normen

$$||u - u_h||_{L_2(K)} = \left(\int_K |u - u_h|^2 dx \right)^{1/2}$$

ist. u gibt dabei die kontinuierliche Lösung an, u_h das Feld finiter Elemente und K ein Element des Gitters. [dea]

3.3.1 Fehleranalyse

Wie erwartet zeigt die globale Verfeinerung den geringsten Fehler. Zudem ist es auch der Fehler, der im Laufe der Simulation mit dem niedrigsten Faktor wächst. Die anderen Strategien zeigen grob zusammengefasst drei unterschiedliche Verhaltensweisen, die aber alle einem linearen Fehlerwachstum entsprechen.
Die beiden Konfigurationen mit der ursprünglichen Strategie bleiben vergleichsweise nah am Verhalten der globalen Verfeinerung, insbesondere zu Beginn der

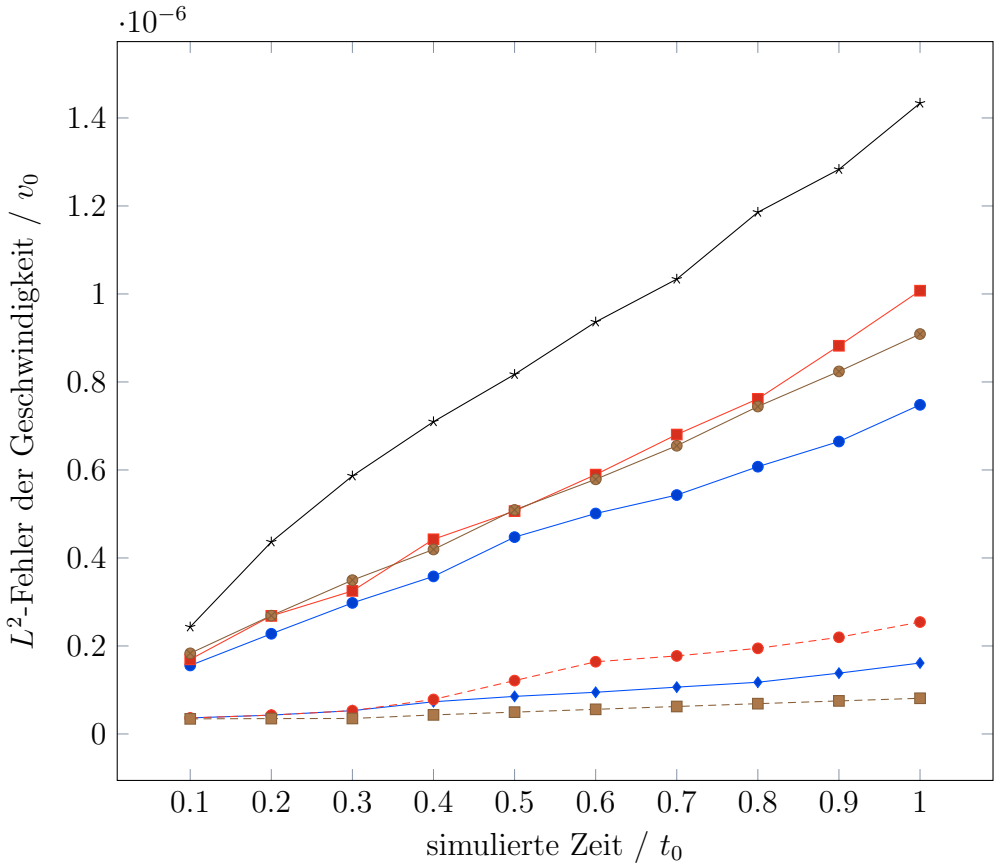

Abbildung 3.2: Bezüglich der Entwicklung des L^2-Fehlers der Geschwindigkeit für die verschiedenen Verfeinerungsstrategien zeigen sich trotz insgesamt ähnlichem Verhalten erhebliche Unterschiede zwischen den Strategien. Trotz der relativ kurzen Simulationsdauer kann keine andere Strategie das Fehlerniveau der globalen Verfeinerung wirklich aufrecht erhalten.

Simulation. Ihr Fehler ist gering und wächst eher langsam. Dabei liegt die Konfiguration mit Verfeinerungsschwellwert 0.3 etwa mittig zwischen der Konfiguration mit Schwellwert 0.075 und der globalen Verfeinerung. Obwohl es sich um die AMR-Strategie mit dem geringsten Fehler handelt ist auch hier der Fehler nach nur 1000

Zeitschritten bereits doppelt so hoch wie beim Vergleichswert der globalen Verfeinerung, $1.6120 \cdot 10^{-7}$ statt $8.1144 \cdot 10^{-8}$, der Fehlerwert der Konfiguration mit Schwellwert 0.075 entsprecht mit $2.5433 \cdot 10^{-7}$ nach der gleichen Anzahl Zeitschritte bereits dem Dreifachen des Vergleichswert.
Noch deutlicher ist die Abweichung bei der nächsten Gruppe, den beiden Fixed-Number-Strategien und der Fixed-Fraction-Straegie mit Verfeinerungsparameter 0.3. Ihr Fehler ist bereits beim ersten eingezeichneten Datenpunkt ($t = 0.1$, nach 100 Zeitschritten) deutlich höher als bei der ersten Gruppe. Der Steigerungsfaktor des Fehlers hingegen ist vergleichbar mit der ersten Gruppe. Während der Fehler der globalen Verfeinerung über die gemessenen Datenpunkte mit einem Faktor von 2.36 und der Fehler der ursprünglichen Strategie mit 4.44 (Verfeinerungsparameter 0.3) beziehungsweise 6.99 (0.075) steigt, liegt dieser Faktor bei der zweiten Gruppe bei 4.96 (FF, Verfeinerungsparameter 0.3), 4.8 (FN, 0.3) und 5.92 (FN, 0.075).
Zuletzt bleibt die Fixed-Fraction-Strategie mit Verfeinerungsparameter 0.075. Hier fällt auf, dass diese Konfiguration mit $2.4316 \cdot 10^{-7}$ sowohl den höchsten Fehler für $t = 0.1$ hat – höher als der Fehler der ursprünglichen Strategie für $t = 1.0$ ($1.6120 \cdot 10^{-7}$) – als auch mit 5.9 einen relativ hohen Wachstumsfaktor. Bei einer längeren Simulationsdauer müsste bei dieser Strategie also mit einem erheblichen Fehlerzuwachs gerechnet werden.

3.3.2 Laufzeitanalyse

Allerdings sind Fehler nicht alles, sonst gäbe es gar keinen Anreiz, AMR zu nutzen statt einfach das komplette Gitter auf das höchte Verfeinerungslevel zu bringen.
Daher soll am dieser Stelle nun die Entwicklung der Laufzeit der verschiedenen Strategien und der verschiedenen Verfeinerungparameter detailliert betrachtet werden. Als Maß dafür wurde die CPU-Runtime – die tatsächlich mit Berechnungen verbrachte Zeit – statt der Walltime – die Differenz aus Systemzeit zu Start und Ende des Problems – heran gezogen.
Besonders sticht dabei die globale Verfeinerung heraus, die bereits für die ersten

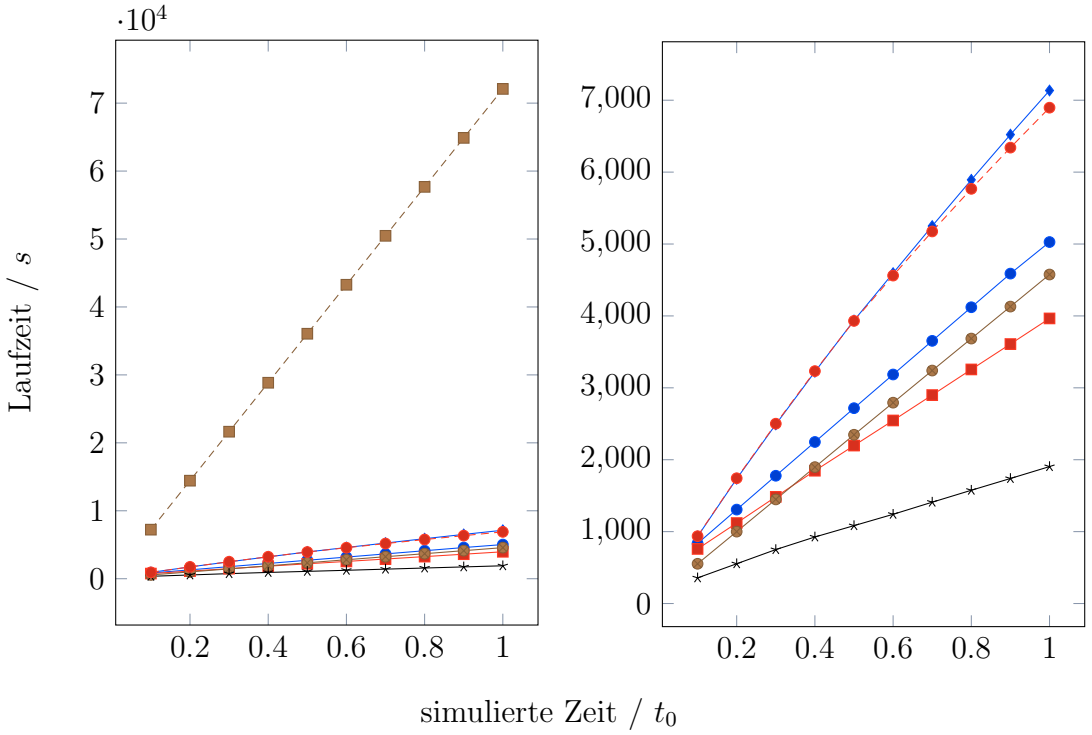

Abbildung 3.3: Vergleicht man die Laufzeiten der verschiedenen Verfeinerungsstrategien, fällt zuerst auf, dass sie alle deutlich unterhalb der Laufzeit der Simulation mit globaler Verfeinerung liegen. Aus diesem Grund wurde im rechten Plot die globale Verfeinerung ausgeblendet, um eine detaillierte Betrachtung der anderen Strategien zu ermöglichen.
Dieser Plot nutzt aus Platzgründen dieselbe Legende wie 3.2.

100 Zeitschritte mehr Zeit braucht (7222.22 s) als selbst die langsamste AMR-Strategie – die ursprüngliche Strategie mit Verfeinerungspararameter 0.3 – für alle 1000 Zeitschritte der Simulation (7134.94 s). Mit 72089.8 s braucht sie letztlich rund zehnmal so lange um die gleiche Simulation abzuschließen, kann dafür aber wie in Abbildung 3.2 gezeigt keineswegs einen um den Faktor 10 geringeren Fehler aufweisen.

Die globale Verfeinerung erhöht die Rechendauer in einem so hohen Maße, dass es nur schwer möglich ist, Details zum Laufzeitverhalten der anderen Strategien zu erkennen. Aus diesem Grund habe ich mich entschieden, den Plot 3.3 zu teilen und sowohl mit als auch ohne Verfeinerung darzustellen. Dieser Kompromiss erlaubt es, sowohl das Verhältnis zwischen der Laufzeit von Simulationen mit glo-

baler Verfeinerung und AMR-Strategien darzustellen als auch die Unterschiede zwischen den einzelnen AMR-Straegien im Detail betrachten zu können.
Dabei ist wieder die gleiche Anordnung in Gruppen wie in den Plots zur Entwicklung des Fehlers (Abbildung 3.2) zu beobachten.

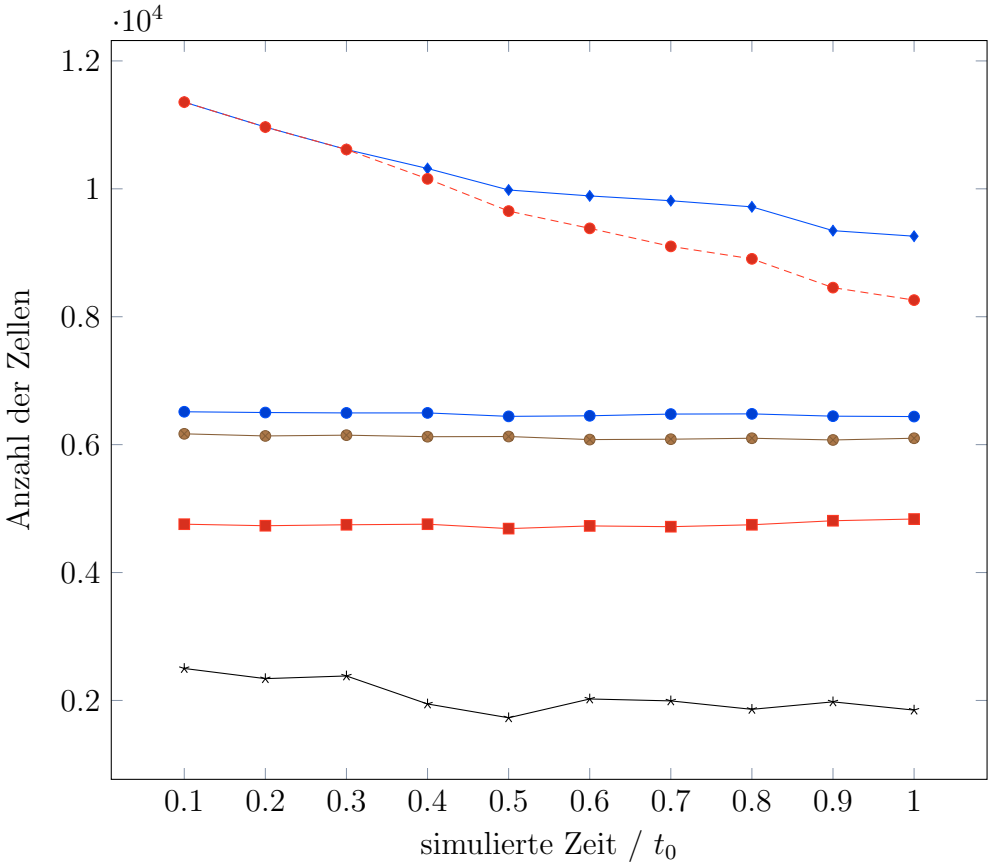

Abbildung 3.4: Auffällig ist hier, dass nahezu alle Strategien zu einer annähernd konstanten Zellenzahl über den Verlauf der Simulation führen.
Dieser Plot nutzt aus Platzgründen dieselbe Legende wie Abbildung 3.2. Auf den Vergleich mit der globalen Verfeinerung wurde hier verzichtet, da sie per Definition eine konstante Zellenzahl hat und zudem den Plot stark verzerren würde.

Diese Entwicklung lässt sich leicht erklären, wenn man die Entwicklung der Anzahl der Zellen betrachtet.
Insbesondere fällt sofort auf, dass die meisten Strategien eine annähernd konstante Zellenzahl über den Verlauf der Simulation aufweisen. Dies ist nicht von

Simulationsbeginn an so: Alle Simulationen beginnen mit dem Gitter der globalen Verfeinerung, haben also 102400 Zellen, die zuerst wegen des auf 6 beschränkten maximalen Verfeinerungslevel nur vergröbert werden können. Erst nach und nach pendeln sich Verfeinerung und Vergröberung ein und führen zu einer annähernd konstanten Anzahl von Zellen.
Eine weitere Auffälligkeit ist, dass sich hier wieder die gleichen Gruppierungen herausbilden, die auch für den L^2-Fehler und die Laufzeit beobachtet werden konnten. Allerdings ist die Reihenfolge gegenüber dem Plot der Fehlerentwicklung 3.2 umgekehrt und orientiert sich eher an den Laufzeiten 3.3.
Die Fixed-Fraction-Strategie mit Verfeinerungsparameter 0.075, die die geringste Laufzeit hatte, hat auch mit Abstand die wenigsten Zellen, während die Fixed-Fraction-Strategie mit Verfeinerungsparameter 0.3 und die beiden Fixed-Number-Strategien bereits ein Vielfaches an Zellen aufweisen und somit Genauigkeitsverluste durch die Diskretisierung besser auffangen können.
Die ursprüngliche Strategie folgt diesem Verhalten dahingehend, dass die Simulationsläufe, die sie zur Gitterverfeinerung nutzen, die höchsten Zellanzahlen, die höchsten Laufzeiten und den geringsten Fehler haben, fallen aber dadurch auf, dass sie zumindest im Zeitraum der Simulation keine konstante Zellzahl aufweisen sondern diese langsam abfällt. Dabei ist bei den vorhandenen Messdaten nicht ersichtlich, ob auch sie später konstant werden oder weiterhin abfallen. Dies müsste mit einer längerfristigen Betrachtung als 1000 Zeitschritten geklärt werden.

Auffällig ist auch, dass die Simulationsläufe mit besonders wenig Schwankungen in der Zellenzahl ein sehr lineares Laufzeitverhalten zeigen während beispielsweise die Laufzeit der Simulationen, die die ursprüngliche Strategie nutzen, im späteren Verlauf der Simulation weniger schnell zunehmen. So benötigt die ursprüngliche Strategie mit Verfeinerungsparameter 0.075 805.937 s um $t = 0.1$ bis $t = 0.2$ zu simulieren, aber lediglich 555.62 s für $t = 0.9$ bis $t = 1.0$. Dies passt zu der Beobachtung in Abbildung 3.4, dass die Anzahl der Zellen über die Laufzeit der Simulation abnimmt. Während bei $t = 0.1$ 11356 Zellen vorliegen, sind es bei $t = 0.9$ nur noch 8455 Zellen.

Nicht in Abbildung 3.4 dargestellt ist die globale Verfeinerung, die entsprechend ihrer Definition über die Laufzeit der Zellen konstant 102400 Zellen behält und somit rund das Neunfache der Zellenzahl der Simulationen mit der ursprünglichen Strategie zum Zeitpunkt $t = 0.1$.

Die Abbildungen 3.2, 3.3 und 3.4 zusammengenommen bestätigen die grundlegende Annahme der adaptiven Gitterverfeinerung: Die Laufzeit der Simulation – und damit die Rechenkosten – hängt in erheblichem Maße mit der Anzahl der Zellen zusammen.
Die Beziehung zwischen Fehler und Zellanzahl hingegen ist deutlich lockerer. Wie stark der Genauigkeitsverlust durch eine geringere Anzahl von Zellen ist, hängt in hohem Maße von der genauen räumlichen Verteilung der Zellen ab. Grundsätzlich begünstigen mehr und feinere Zellen aber einen geringen Fehler.

3.3.3 Bewertung der Strategien

Daraus ergibt sich die Fragestellung, wie man eine AMR-Verfeinerungsstrategie bewertet. Eine einzelne Größe kann es nicht sein, vielmehr erscheint es im Allgemeinen erstrebenswert, einen Kompromiss aus geringem Fehler und geringer Laufzeit zu suchen.
Ich verwende daher

$$L^2 - Fehler \cdot Laufzeit$$

als Maß für die Beurteilung einer Verfeinerungsstrategie. Dabei gilt, dass ein möglichst kleiner Wert auf eine gute Strategie hinweist.

Obwohl sie den geringsten Fehler aufwies, zeigt sich in Abbildung 3.5 die Schwäche der globalen Verfeinerung. Bedingt durch die extrem große Anzahl von Zellen und die daraus resultierende hohe Laufzeit hat sie die das ungünstigste Verhältnis aus Kosten und Genauigkeit nach dem festgelegten Kriterium. Der numerische Mehr-

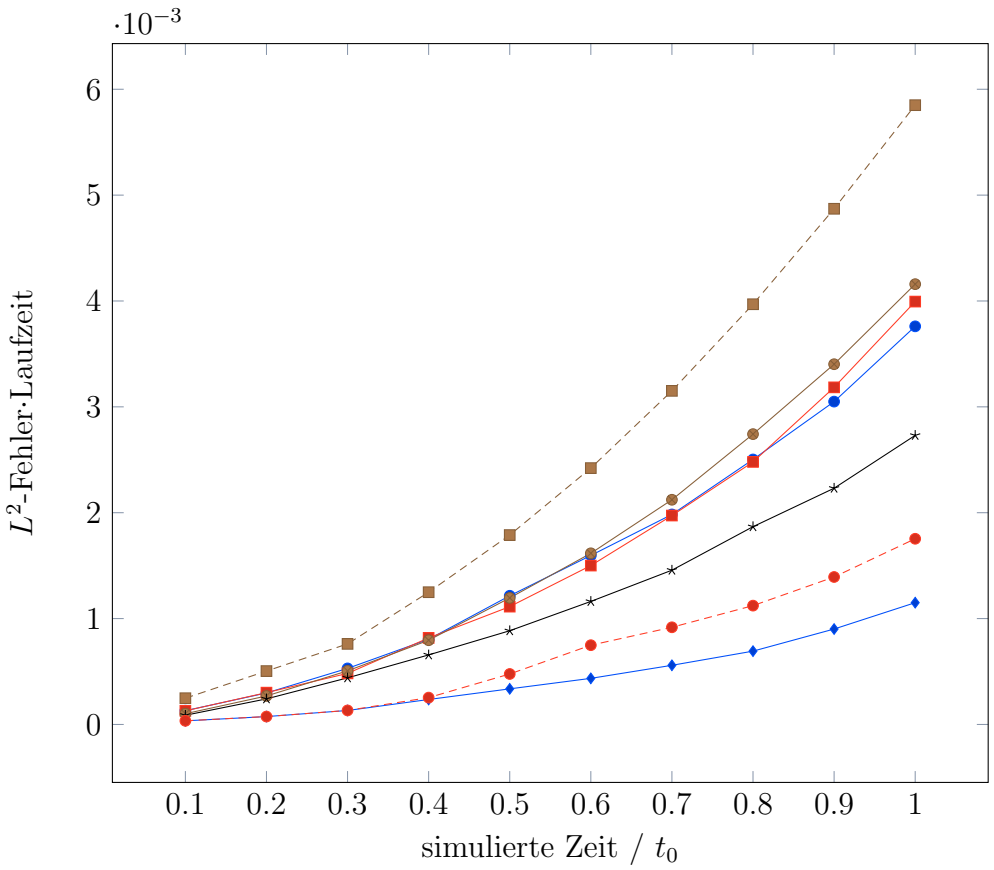

Abbildung 3.5: Um ein Kriterium dafür zu finden, wie gut eine Strategie einen Kompromiss aus reduzierten Kosten und minimalem Fehler findet, werden beide Werte miteinander multipliziert. Dies gibt einen Index für jede der Strategien.
Dieser Plot nutzt aus Platzgründen dieselbe Legende wie Abbildung 3.2.

aufwand übersteigt wie bereits in 3.3.2 geschildert schlicht den Genauigkeitszugewinn einer so feinen Verfeinerung.

Am Besten beurteilt diese Maß hingegen die ursprüngliche Strategie mit ihrem eher konservativen Verfeinerungsverhalten. Ihr Fehler ist im Rahmen der beispielhaft betrachteten Simulation lediglich doppelt bis dreimal so groß wie der durch globale Verfeinerung vorgegebene Minimalfehler und dies in einem Zehntel der Laufzeit.

Die anderen Strategien liegen zwischen diesen beiden Polen. Zwar haben sie einen erheblich höheren Fehler, können dies aber teilweise durch die geringe Zellanzahl

und dir kurzen Laufzeiten ausgleichen.
Sollte ein geringerer Fehler gewünscht sein als dies die vorgestellten Strategien leisten können, so würde es immer noch mehr Sinn machen, eine geeignete AMR-Strategie mit einem höheren maximalen Verfeinerungslevel zu nutzen statt globale Verfeinerung. Die zusätzlichen Zellen würden so eher dort eingesetzt, wo sie möglichst viel dazu beitragen, den Fehler durch die räumliche Diskretisierung so klein wie möglich zu halten.

Welche Strategie letztlich die Geeignetste ist, kann dennoch nicht pauschal beantwortet werden. Vielmehr ist diese Entscheidung abhängig von den persönlichen Zielen und Vorstellungen. Das eben beschriebene Mäß zielt auf einen Kompromiss zwischen kleinem Fehler und Laufzeitreduzierung ab. Diese Herangehensweise ist bei vielen Simulationen ein guter Ausgangspunkt, wenn keine anderen Prioritäten vorliegen. In diesen Fällen ist die ursprüngliche Strategie mit Verfeinerungsparameter 0.3 und Vergröberungsparameter 0.3 eine solide Wahl.
In anderen Situationen hingegen könnte es Priorität haben, die Anzahl der Zellen konstant zu halten, etwa aus Gründen der Laufzeitbegrenuzung. Hier würde sich eine Fixed-Number-Strategie anbieten, da sich mit dieser die Anzahl der aktiven Zellen besonders gut kontrollieren lässt.
Wenn es hingegen in erster Linie wichtig ist, den Fehler zu minimieren und die Laufzeit zweitrangig ist, lohnt es sich durchaus, die globale Verfeinerung in Betracht zu ziehen. Hier werden garantiert alle relevanten Simulationsgebiete fein aufgelöst, aber eben auch viele weniger relevante Gebiete. Alternativ würden auch andere eher konservative Strategien wie die ursprüngliche Strategie mit einem hohen Verfeinerungsparameter hier gute Resultate zeigen.

Im folgenden Kapitel wird daher der Übersichtlichkeit halber nur noch eine Strategie betrachtet. Die Wahl fiel dabei auf die ursprüngliche Strategie mit Verfeinerungsparameter 0.3 und Vergröberungsparameter 0.3, da diese sich als gute General-Purpose-Strategie erwiesen hat. Sie erzielt das günstigste Verhältnis aus Fehler und Laufzeit von allen betrachteten Strategien. Dieser Kompromiss wird

bei vielen Simulationen mit AMR angestrebt.

Kapitel 4

Betrachtung der Parallelität

Wie bereits in 2.3 beschrieben besitzt JuFire eine MPI-Parallelisierung, die über Deal.II und die enthaltene Schnittstelle zur Bibliothek `p4est` umgesetzt wird. [BBHK11]
MPI zeichnet sich dadurch aus, dass die zur Berechnung genutzten Kerne explizit durch Nachrichten miteinander kommunizieren müssen. Dies macht es besonders geeignet für Distributed-Memory-Systeme. Im Kontrast dazu verfügen Shared-Memory-Systeme über einen gemeinsamen Arbeitsspeicher, auf den alle Kerne eines Knotens zugreifen können. Diese implizite Art der Kommunikation zwischen den Prozessoren, beispielsweise zum Austausch von Zwischenergebnissen, ist ressourcensparend und vergleichsweise leicht zu programmieren. Durch ihre Hardware-Anforderung ist sie in der Regel aber nur in kleinem Maßstab umzusetzen.
Distributed-Memory-Systeme hingegen können sehr groß werden und sind daher besonders für sehr umfangreiche Berechnungen geeignet. Die meisten modernen Supercomputer sind Hybrid-Systeme, in denen mehrere Shared-Memory-Systeme, die Knoten, durch Infiniband oder ein vergleichbares Datenübertragungssystem miteinander zu einem Distributed-Memory-System verbunden sind.

Während Programmierparadigmen für Distributed-Memory-Systeme wie MPI auch auf Shared-Memory-Systemen angewandt werden können, indem sie den Vorteil des gemeinsamen Speichers ignorieren und die darauf zugreifenden Prozessoren

wie separate Entitäten behandeln, ist die Umkehrung für Shared-Memory-Programmierparadigmen wie OpenMP nicht möglich. Aus diesem Grund behandeln größere Programme in der Regel entweder das ganze System als Distributed-Memory-System oder nutzen einen hybriden Ansatz, bei denen innerhalb eines Knotens die Shared-Memory-Eigenschaften ausgenutzt und zwischen den Knoten mit einem Distributed-Memory-tauglichen Ansatz kommuniziert wird.

Dabei ist zu bedenken, dass es vergleichsweise leicht ist, ein vorhandenes serielles – also nur mit einem Prozessor berechnetes – Programm beispielsweise mit MPI zu parallelisieren. Weitaus schwerer ist es, dies auf eine Weise zu tun, die die zur Verfügung stehenden Ressourcen optimal ausnutzt. Aus diesem Grund ist es wichtig, sich mit dem parallelen Laufzeitverhalten des Programms zu beschäftigen und zu analysieren ob und inwiefern Raum für Verbesserungen besteht. Die kann etwa durch die Beseitigung von Flaschenhälsen, bei denen eine Programmstelle die Performance des ganzen Programms ausbremst, oder die Reduzierung von Wartezeiten bei der Kommunikation zwischen Kernen geschehen.

Die zur Beurteilung des parallelen Verhaltens von JuFire notwendigen Berechnungen wurden auf JURECA durchgeführt, dem General-Purpose-Supercomputer des Jülich Supercomputing Centers. JURECA wurde 2015 in Betrieb genommen, im Herbst 2017 steht die Erweiterung mit einem skalierbaren Booster-Modul an.
Zum Zeitpunkt der Untersuchung, vor der Erweiterung, umfasst das Cluster 1872 Rechenknoten mit jeweils 2 Intel Xeon E5-2680 v3 Haswell CPUs. Somit stehen 24 Kerne pro Knoten zur Verfügung. 75 Knoten sind zusätzlich mit zwei NVIDIA K80 GPUs zur Beschleunigung ausgestattet. Ergänzt werden die Rechenknoten noch durch zwölf Visualisierungsknoten mit denselben CPUs wie die Rechenknoten sowie jeweils zwei NVIDIA K40 GPUs und bis zu 1024 GiB Speicher. Insgesamt verfügt das Clustermodul somit über 45,216 CPU-Kerne und kann damit eine Spitzenleistung von 1.8 (CPU) + 0.44 (GPU) Petaflop pro Sekunde erreichen. Dies entspricht 10^15 Fließkomma-Rechenoperationen pro Sekunde.
JURECA nutzt CentOS 7, eine minimale Linux-Distribution, als Betriebssystem

und verwendet das Slurm-Batch-System zur Verwaltung der Jobs, die Nutzer übermitteln.

Im Gegensatz zu einem Highly-Scalable-Supercomputer wie JUQUEEN mit seinen 458752 Knoten ist JURECA nicht nur darauf ausgelegt, extrem umfangreiche, stark auf die Parallelisierung hin optimierte Simulationen mit einer großen Anzahl von Knoten zu berechnen. Stattdessen läuft dort eine breite Auswahl Simulationen, die teils nicht einmal einen ganzen Knoten nutzen. Auch in der Lehre wird JURECA genutzt um Studenten die Prinzipien der Parallelprogrammierung und den Umgang mit Supercomputern zu vermitteln.
Ergänzt wird dies durch die Visualisierungsknoten, mit denen direkt auf dem Cluster die Simulationsergebnisse visuell aufbereitet werden können. Auch dieser häufig zeitintensive Arbeitsschritt kann dabei parallel ausgeführt werden. [For17]

4.1 Betrachtung des Speedups

Es gibt diverse Möglichkeiten, wie man die Qualität einer Parallelisierung vergleichbar machen kann. Dies ist beispielsweise notwendig um nach einer durchgeführten Programmänderung deren Nutzen belegen und messen zu können.
Einer der häufigsten und simpelsten Ansätze ist der so genannte Speedup. Diese Metrik beschreibt die Beschleunigung, die mit einer bestimmten Anzahl Kernegegenüber dem besten seriellen Algorithmus erzielt wird. Dabei gilt

$$S_p = \frac{T_S}{T_p},$$

wobei S_p den Speedup mit p Kernen bezeichnet, T_S die Zeit, die die beste serielle Simulation benötigt und T_p die Zeit, die mit p Kernen benötigt wurde. Dabei gibt es zwei Ansätze, die zu unterscheiden sind.
Bei der starken Skalierung wird die Problemgröße konstant gehalten, die zusätzlichen Rechenressourcen sollen die Berechnungsdauer verringern. Als ideal gilt es, wenn der $S_p = p$ gilt. In diesem Fall würde eine Simulation mit zehn Kernen auch

nur noch ein Zehntel der Zeit benötigen, die die serielle Berechnung brauchte. Das würde allerdings voraussetzen, dass das Problem vollständig parallelisierbar ist – dies trifft in der Praxis nur selten zu, da beispielsweise das Schreiben von Ausgabedateien häufig seriell erfolgt. Hinzu kommt, dass die Parallelisierung stets auch einen gewissen Zusatzaufwand bedeutet, insbesondere durch die bei MPI notwendige Kommunikation zwischen den verschiedenen Kernen. Dieser Zusatz wird Overhead genannt. [Rah94]
Schwache Skalierung hingegen hält das Verhältnis von Problemgröße und verwendeten Ressourcen konstant. Für eine Berechnung mit zehn Kernen würde hier also auch ein zehnmal so großes Problem heran gezogen. Dieser Ansatz ist oft nachsichtiger, da nicht parallelisierbare Programmanteile hier weniger stark zum tragen kommen.
Im Folgenden wird die starke Skalierung betrachtet, da die Problemgröße in der Praxis oft vorgegeben ist und die Berechnungsdauer durch Parallelisierung verringert werden soll.

Abbildung 4.1 basiert auf Daten, die nicht mit dem in Kapitel 2 beschriebenen 2D-Vortex-Problem ermittelt wurden sondern mit einem 3D-Problem ohne aktiviertes AMR. Aufgrund der Problemgröße und der Laufzeitbegrenzung von 24 Stunden auf JURECA wurden keine seriellen Berechnungen vorgenommen sondern stattdessen die Berechnungsdauer mit einem Knoten, also 24 Kernen, als Ausgangswert genutzt. Dabei wurde aufgrund des guten Skalierungsverhaltens ab diesem Wert angenommen, dass es auch auf eine niedrigere Anzahl Kerne übertragbar wäre. Es handelt sich also strikt gesehen nicht um einen Speedup, da das Verhältnis der Laufzeiten zweier paralleler Programme verglichen wird. Im Folgenden wird es allerdings aufgrund der ähnlichen Methodik und der Verständlichkeit halber als solcher bezeichnet.

Charakteristisch ist dabei, dass der gemessene Speedup zuerst eng dem Idealverlauf folgt, dann merklich abflacht und schließlich stark abfällt. Der Grund für dieses Verhalten wird offensichtlich, wenn man den Zeitpunkt des Abfalls und die

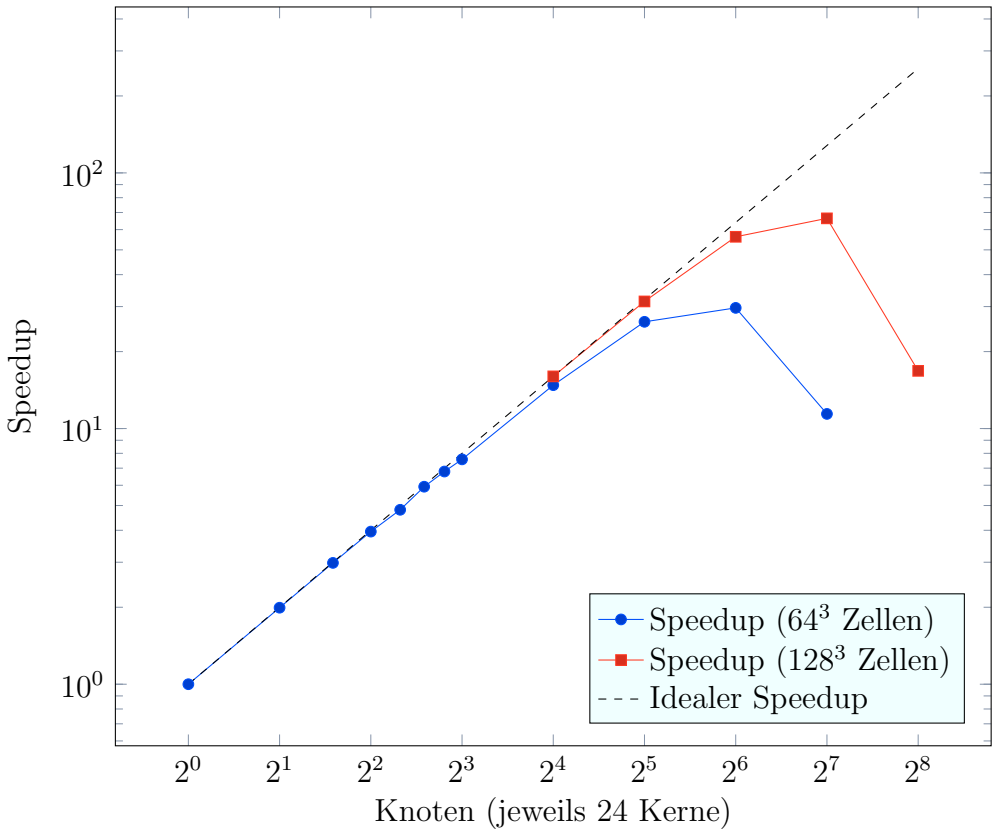

Abbildung 4.1: Bei hinreichender Problemgröße, hier ein 3D-Problem ohne aktiviertes AMR, skaliert JuFire gut. Wird das Verhältnis von Zellen zu Kernen allerdings zu gering, stagniert der Speedup und sinkt schließlich.

Problemgröße miteinander in Zusammenhang setzt. Das größere Problem umfasst 128 Zellen in jeder Dimension, insgesamt also 2097152 Zellen, die auf die verwendeten Kerne aufgeteilt werden. Bei $2^7 = 128$ Knoten mit jeweils 24 Kernen ergibt dies durchschnittlich etwa 683 Zellen pro Kern. Dieser Wert ist vergleichsweise gering und deutet in Verbindung mit dem Kurvenverlauf darauf hin, dass der Mehraufwand des Aufteilens des Rechengebiets und der Kommunikation hier im Verhältnis zum Zeitgewinn durch die Arbeitsteilung zunimmt. Noch übersteigt er ihn allerdings nicht, dies ist erst bei $2^8 = 256$ Knoten und durchschnittlich rund 341 Zellen pro Kern klar anhand des Abfalls der Kurve ersichtlich.

Die zweite Messreihe in Abbildung 4.1 zeigt ein kleineres Problem mit 64 Zellen in jeder Dimension, insgesamt also 262144 Zellen. Der Vergleich der beiden Speedup-

kurven bekräftigt die Vermutung, dass das Problem zuerst gut skaliert, an einem gewissen Punkt aber das Verhältnis von Nutzen und Kosten der Parallelisierung kippt und infolgedessen der Speedup einbricht. Hier ist bereits für $2^6 = 64$ Knoten das Abflachen des Speedups zu erkennen, das die Annäherung von Nutzen und Kosten ankündigt. Dies entspricht hier durchschnittlich etwa 171 Zellen pro Kern. Bei 2^7 Knoten – rund 85 Zellen pro Kern – ist klar zu erkennen, dass die Kosten den Nutzen der Parallelisierung überwiegen.
Auffällig ist hier allerdings, dass es offenbar keine feste Grenze vorliegt, die die durchschnittliche Anzahl von Zellen pro Knoten nicht unterschreiten darf. Beide Kurven basieren auf Simulationen des gleichen Problems, es unterscheidet sich lediglich die Anzahl der Zellen aus denen sich das Gitter zusammen setzt. Dennoch zeigt das feiner aufgelöste Problem bereits mit 341 Zellen pro Kern einen massiven Abfall des Speedups, während das gröbere Problem für $2^5 = 32$ Knoten dasselbe Verhältnis von Zellen und Kernen aufweist, allerdings noch sehr gut skaliert.

Um den Einfluss der adaptiven Gitterverfeinerung auf das parallele Verhalten zu untersuchen wurde bewusst ein kleineres Problem gewählt, das bereits mit einer niedrigeren Anzahl von Kernen den in Abbildung 4.1 beobachteten Knick erreicht. Hierfür bot sich das in Kapitel 2 beschriebene 2D-Vortex-Problem an. Der Endzeitpunkt der Simulation wurde auf $t_E = 0.1$ reduziert, es werden also 100 Simulationsschritte berechnet. Diese Entscheidung beruht darauf, dass die Parallelisierung innerhalb der einzelnen Simulationsschritte stattfindet und nach jedem Schritt die einzelnen Prozesse synchronisiert werden. Dies ist notwendig, da die Berechnungen auf die Ergebnisse des vorherigen Zeitschrittes zurück greifen und eventuell zwischen den Schritten Ausgabedateien geschrieben werden.
Wie in 3.3.3 begründet, beschränkt sich die Betrachtung auf die ursprüngliche Strategie mit Verfeinerungsschwellwert 0.3 und Vergröberungsschwellwert 0.3.

Der dabei beobachtete Speedup zeigt grundsätzlich einen ähnlichen Verhalten wie bei dem 3D-Problem ohne aktiviertes AMR. Er steigt annähernd linear an, flacht an einem gewissen Punkt ab und beginnt schließlich wieder zu fallen. Dass dieser

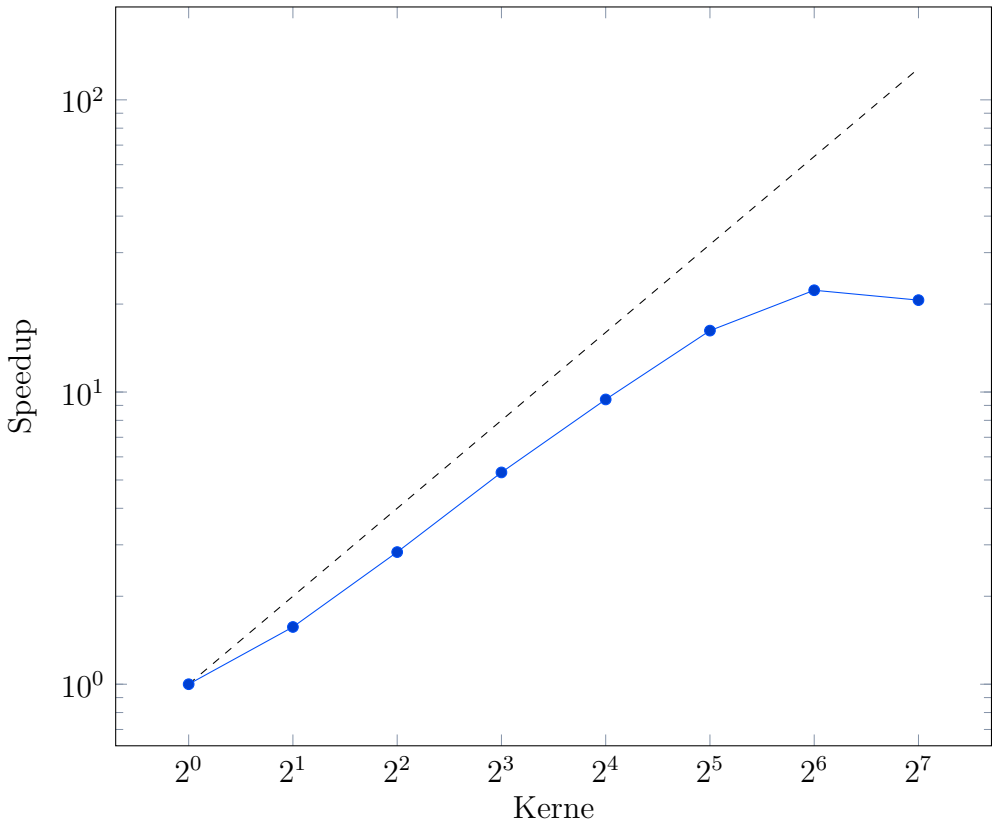

Abbildung 4.2: Auch bei einer Annäherung an das Szenario aus Abbildung 4.1 indem lediglich die Laufzeiten paralleler Simulationen betrachtet werden, ist eine deutliche Abweichung vom idealen Speedup erkennbar.

Verlauf allgemein flacher und der Abfall zuerst nicht ganz so drastisch ist, kann durch die kleine Problemgröße und daraus resultierend auch die geringe Laufzeit und die wenigen verwendeten Kerne erklärt werden. Dadurch werden Änderungen im Verhältnis zwischen Nutzen und Kosten schneller widergespiegelt.
Beispielsweise zeigt sich beim 2D-Vortex-Problem der Peak des Speedups bei 64 Kerne, der Abfall wird bei 128 Kernen beobachtet. Es werden also maximal 64 weitere Kerne hinzu gefügt, die durch die Parallelisierung mehr Kosten verursachen als sie die Berechnung beschleunigen. Im Kontrast dazu ist der Peak des größeren 3D-Problems aus Abbildung 4.1 bei 128 Knoten mit jeweils 24 Kernen, insgesamt

also 3072 Kernen. Der Abfall wird bei 256 Knoten beobachtet, es können also bis zu 3072 Knoten dazu beitragen, die Kosten der Parallelisierung in die Höhe zu treiben.

Besonders fällt hier auf, dass der Speedup zwar bis zum Peak linear verläuft, allerdings nicht an die Ideallinie heran kommt. Auffällig ist hier insbesondere die Berechnung mit 2 Knoten, also die erste parallele Berechnung, ab der der Verlauf etwa der Steigung des idealen Speedups folgt.
Es wäre ein naheliegender Ansatz, daraus zu folgern dass die bei dem 3D-Problem getroffene Annahme, dass wenn ein Problem mit einer parallelen Berechnung als Ausgangswert gut skaliert, dies auch für kleinere parallele sowie serielle Berechnungen gilt, nicht zutreffend ist. In der Tat muss man dabei von einer gewissen Ungenauigkeit ausgehen, insbesondere wenn seriell ein anderer Algorithmus verwendet wird als parallel, was hier allerdings nicht der Fall ist.
Ohnehin kann nicht die komplette Abweichung so erklärt werden. Dies wird offensichtlich, wenn man die serielle Berechnung aus dem Speedup-Graph ausschließt und diesme wie in Abbildung 4.3 auf die Berechnung mit 2 Knoten hin normiert. Hier wird deutlich, dass die Steigung eben nicht dem idealen Speedup entspricht sondern etwas darunter bleibt. /newline
Dieses Verhalten deutet auf einen seriellen Programmanteil hin, der durch die Parallelisierung nicht beschleunigt werden kann. Das Phänomen wird durch Amdahls Gesetz beschrieben:

$$S_{max} = \frac{T_{gesamt}}{t_S + t_{O(n_P)} + \frac{t_P}{n_P}}.$$

Der maximale Speedup S_{max} wird bestimmt durch das Verhältnis der Gesamtlaufzeit T_{gesamt} zu seriellem Programmanteil t_S, dem mit n_p Kernen beschleunigten parallelisierbaren Programmanteil t_P sowie den Zusatzaufwand durch die Parallelisierung $t_{O(n_P)}$. [Amd67]

Da dieses Verhalten bei den Problemen ohne AMR nicht beobachtet wird, erscheint die Folgerung naheliegend, dass das AMR für diesen zusätzlichen seriel-

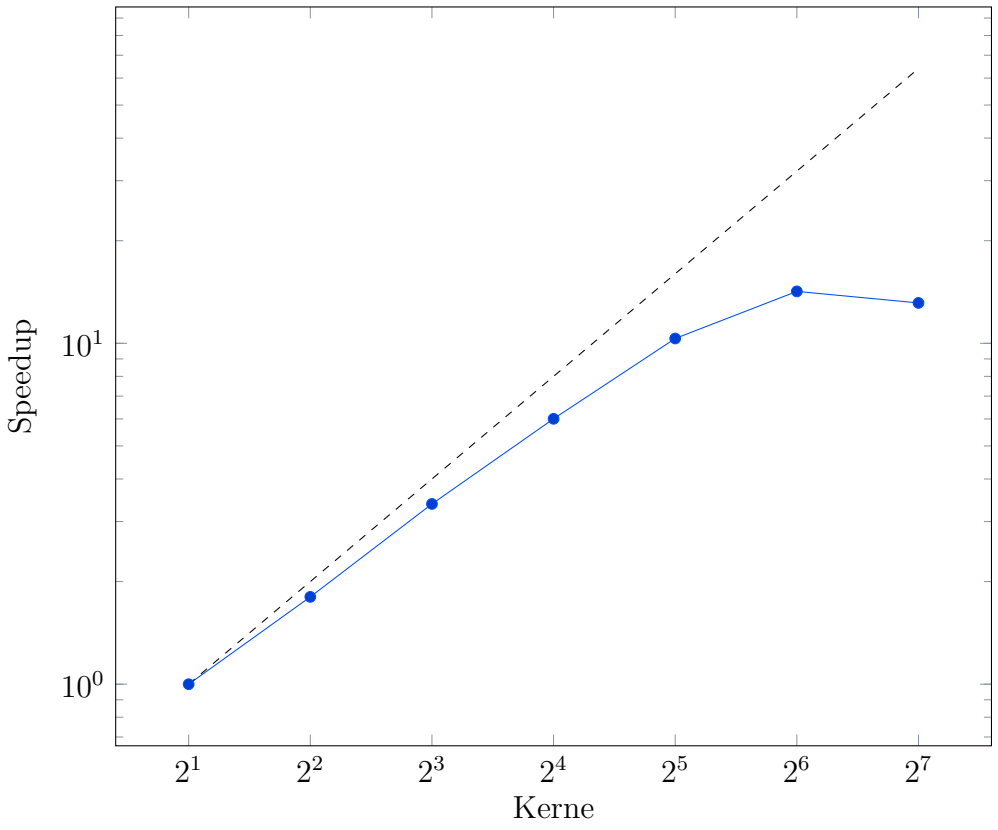

Abbildung 4.3: Bei dem 2D-Vortex mit aktiviertem AMR ist der aus Abbildung 4.1 bekannte Verlauf zu erkennen. Es fällt allerdings auf, dass der Speedup hier stets unterhalb der Ideallinie bleibt, sich aber zwischen 2 und 32 Kernen linear skaliert. Eine mögliche Ursache hierfür wäre das aktivierte AMR.

len Anteil verantwortlich ist. Des Weiteren resultiert der initiale Zusatzaufwand der Parallelisierung gerade bei kleineren Problemen in deutlichen Mehrkosten gegenüber der seriellen Berechnung.

4.2 Scalasca als Werkzeug zur Performance-Analyse

Der Speedup vermittelt einen guten Eindruck vom allgemeinen Potential der Parallelisierung eines Programms und kann helfen, serielle Anteile zu erkennen. Das

Wissen, dass es einen seriellen Anteil gibt ist bereits wichtig. Oft will man aber genauere Auskünfte wie beispielsweise welche Funktionen für den seriellen Anteil verantwortlich sind und ob eventuell Möglichkeiten bestehen, diesen Anteil zu reduzieren. Diese Informationen kann der Speedup nicht vermitteln. Er betrachtet nur die Laufzeit des Programmes sowie die Anzahl der dafür eingesetzten Kerne und setzt diese Werte in ein Verhältnis.

Stattdessen gibt es Software-Tools, die es ermöglichen, die Performance eines parallelen Programms im Detail zu betrachten. Ein solches häufig genutztes Programm ist Scalasca, welches als Gemeinschaftprojekt von Jülich Supercomputing Center, der Technischen Universität Darmstadt und der German Research School for Simulation Sciences entwickelt wird.
Scalasca bietet ein mehrstufiges Vorgehen, mit dem das Laufzeitverhalten eines Programms gemessen und analysiert wird. Die Ergebnisse werden übersichtlich präsentiert und erlauben es so, Flaschenhälse – insbesondere im Bereich der Kommunikation und Synchronisation – zu identifizieren. Dabei werden auch Anhaltspunkte geboten, wie diese behoben werden können. [GWW+10]

4.2.1 Instrumentalisierung

Hierbei handelt es sich um einen Vorbereitungsschritt für die eigentliche Performanceanalyse. Der Quellcode wird neu kompiliert und dabei zusätzlicher Code und Markierungen eingefügt, die während der Laufzeit Kennzahlen zum Programm ermitteln.
Scalasca nutzt zu diesem Zweck Score-P, ein verbreitetes Framework zur Instrumentalisierung paralleler Programme, das auch von anderen Analyse-Tools wie Vampir genutzt wird.Dieses kann aufgerufen werden, indem `scorep` dem üblichem Kommandozeilenbefehl zum Kompilieren voran gestellt wird.
Eine Ausnahme stellen dabei Programme dar, die – wie JuFire – CMake nut-

zen, um das Kompilieren und Verknüpfen mit den genutzten Bibliotheken zu automatisieren. Hier bietet es sich an, stattdessen den von Score-P bereit gestellten Compiler-Wrapper zu nutzen, den es für eine Vielzahl von Compilern gibt. Die integrierte Instrumentalisierung kann dabei über das Compiler-Flag `SCOREP_WRAPPER=off` vor dem CMake-Aufruf deaktiviert werden. In diesem Fall verhalten sich die Wrapper für die Dauer des aktuellen Befehls wie die Standard-Compiler. Dies ist sinnvoll, da beim Aufruf von CMake diverse Tests durchgeführt werden, die aber nicht instrumentalisiert werden sollen, da sie nicht zum eigentlichen Programm gehören.
Wird in Folge dessen `make` aufgerufen, werden die Wrapper aktiv und das Programm mitsamt Instrumentalisierung gebaut.`SCOREP_WRAPPER=off make` hingegen unterdrückt die Instrumentalisierung und das Programm wird so gebaut, wie es mit `cmake .` und `make` der Fall wäre.

4.2.2 Analyse

In diesem Schritt werden Daten zum Laufzeitverhalten und parallelen Eigenschaften des Programms auf Basis der Instrumentalisierung gesammelt und in Ausgabedateien geschrieben. Scalasca nutzt hierfür das Cube-Format, das ähnlich wie Score-P unter Programmen zur Analyse paralleler Performance weit verbreitet ist.
Hierbei wird `scalasca -analyze` (kurz: `scan`) dem Programmaufruf voran gestellt. Dabei besteht auch die Option, mit `scan -f <filter> <programm>` eine Filterdatei einzubinden. Diese legt fest, für welche Programmteile Daten gesammelt werden und welche ausgespart werden sollen. Diese Filterung ist unter Umständen notwendig, um die Analyseergebnisse nicht zu verzerren.
Durch den im Rahmen der Instrumentalisierung eingefügten zusätzlichen Quellcode stellt die Betrachtung mit Scalasca stets einen gewissen Zusatzaufwand dar. Besonders bei häufigen Aufrufen kurzer Funktionen kann dieser Overhead sich akkumulieren und zu Laufzeiten führen, die ein Vielfaches des ursprünglichen Programms betragen. In diesem Fall spiegeln die Ausgabedaten beinahe nur noch den

Overhead durch Scalasca wider und erlauben nur noch wenig Aussagen über das normale Programmverhalten.

4.2.3 Untersuchung

Scalasca unterstützt zwei Modi zur Betrachtung der Ausgabedateien.
Der Standard ist die grafische Aufbereitung mit dem Cube-Filebrowser. Sie bietet eine Aufschlüsselung nach Metriken, Funktionen und Knoten sowie eine farbliche Kodierung entsprechend der ausgewählten Metrik. Durch die geschachtelte Struktur lassen sich leicht die Beziehungen von Funktionen erfassen und die Farbkodierung hilft bei der Identifikation von Lastungleichgewichten zwischen den beteiligten Kernen sowie den Funktionen, die für ein beobachtetes Verhalten ausschlaggebend sind.
Die Cube-Auswertung kann mit `scalasca -examine` (kurz: `square`) aufgerufen werden.

Ebenfalls besteht die Möglichkeit, eine zusammenfassende Auswertung in Form einer Textdatei zu erstellen, die auch in der Konsole betrachtet werden kann. Dies geschieht mit dem Befehl `square -s`. [The16]

4.3 Betrachtung mit Scalasca

Vorbereitung des JuFire-Codes

Wie bereits in 4.2.1 erwähnt wird JuFire mithilfe von CMake und Makefiles kompiliert. Der Einsatz des Score-P-Wrappers ist also notwendig, um den Code zu instrumentalisieren. Dies geschieht mit dem Aufruf `SCOREP_WRAPPER=off cmake . -DCMAKE_C_COMPILER=scorep-gcc -DCMAKE_CXX_COMPILER=scorep-g++` gefolgt von `make`. Darüber hinaus ist die Nutzung eines Filters notwendig. Wie Abbildung 4.5 verdeutlicht, verursacht die Instrumentalisierung ohne Filter einen massiven Overhead, der die Laufzeit der Simulationen in etwa verzehnfacht. Dadurch wird die Aussagekraft der Messungen erheblich reduziert, da fast ausschließlich der Over-

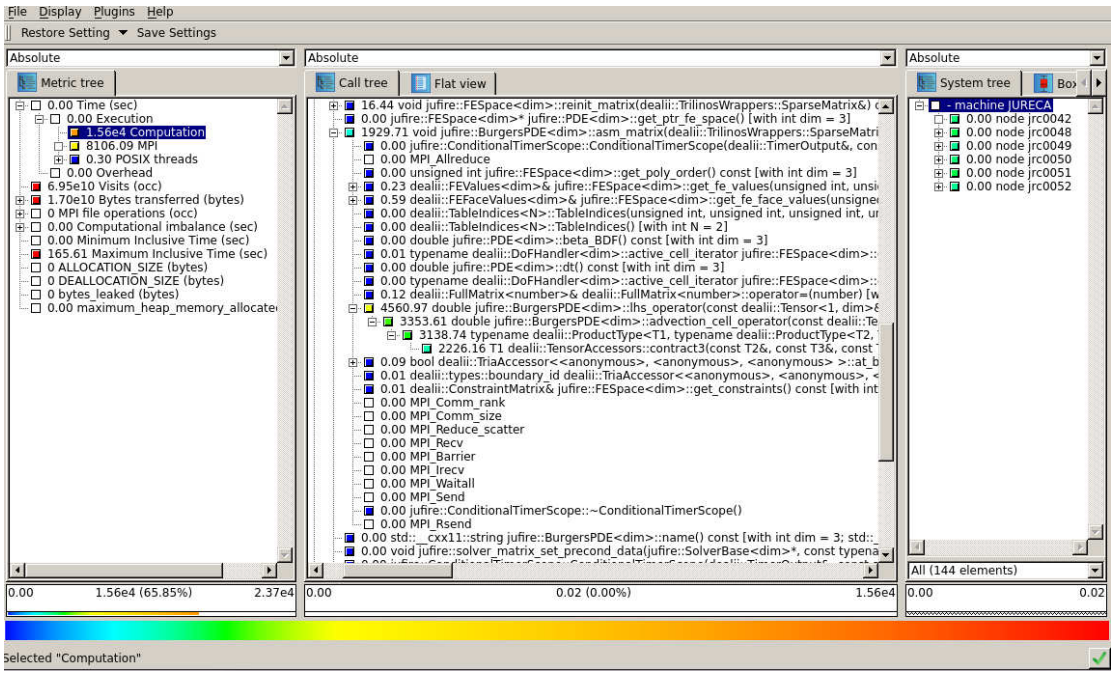

Abbildung 4.4: Der Cube-Filebrowser bereitet die durch Scalasca gesammelten Daten visuell auf. Im linken Fenster befinden sich dabei die zur Verfügung stehenden Metriken, auf deren Basis die Farbkodierung vorgenommen wird, in der Mitte sind die Funktionen des untersuchten Programms aufgelistet. Im rechten Fenster schließlich stehen die zur Berechnung genutzten Knoten und Kerne.

head gemessen wird.

Als Konsequenz wurden vier ineinander geschachtelte Funktionen, identifiziert, die sehr häufig aufgerufen werden, jeweils aber nur eine kurze Laufzeit haben. Dadurch waren sie besonders anfällig dafür, hohe Instrumentalisierungskosten zu akkumulieren. Zudem wurden sie inhaltlich als wenig relevant für das Zusammenspiel von AMR und Parallelisierung bewertet.

Der folgende Filter unterbindet daher ihre Instrumentalisierung:

```
1 SCOREP_REGION_NAMES_BEGIN
2   EXCLUDE
3     typename\ dealii::ProductType<T1,\ typename\ dealii::ProductType<Number,\
      OtherNumber>::type>::type\ dealii::contract3(const\ TensorT1<rank_1,\ dim,\
      T1>&,\ const\ TensorT2<(rank_1\ +\ rank_2),\ dim,\ T2>&,\ const\ TensorT3<
      rank_2,\ dim,\ T3>&)\ \[with\ TensorT1\ =\ dealii::Tensor;\ TensorT2\ =\
      dealii::Tensor;\ TensorT3\ =\ dealii::Tensor;\ int\ rank_1\ =\ 1;\ int\
      rank_2\ =\ 1;\ int\ dim\ =\ 2;\ T1\ =\ double;\ T2\ =\ double;\ T3\ =\ double
      ;\ typename\ dealii::ProductType<T1,\ typename\ dealii::ProductType<Number,\
```

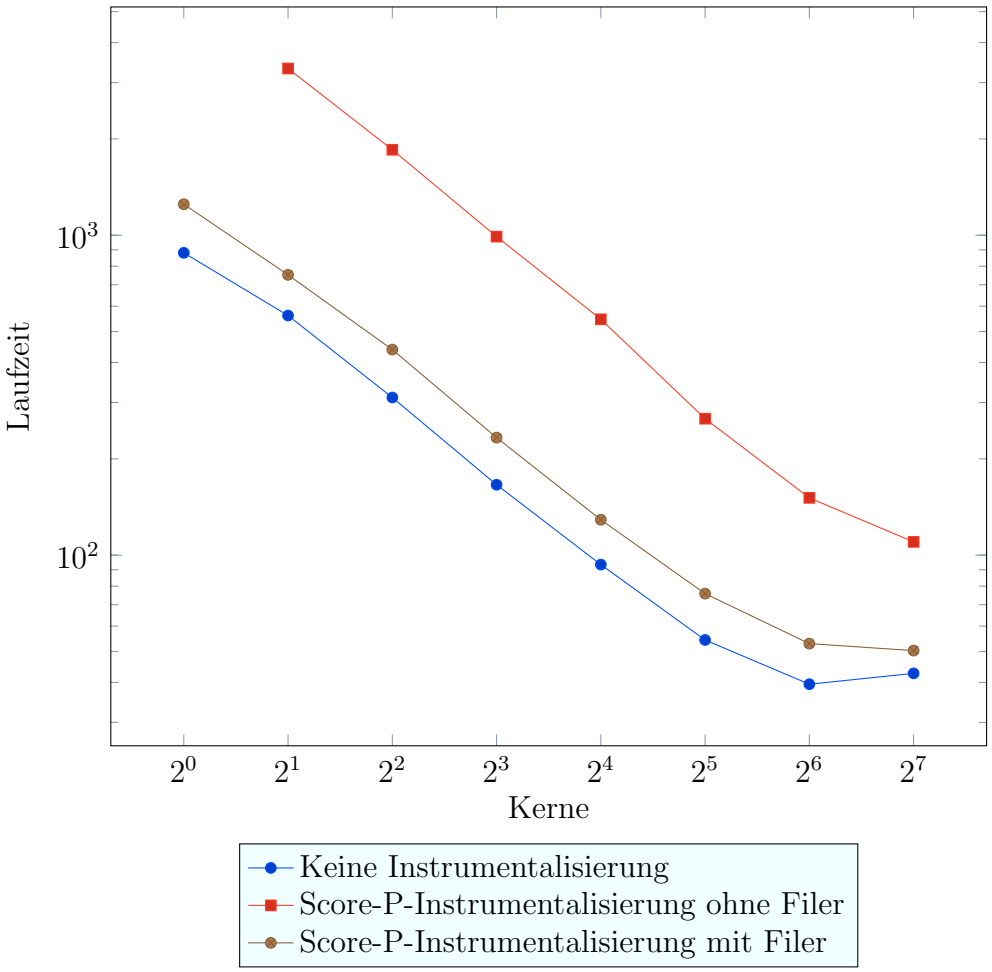

Abbildung 4.5: Ohne Filter ist der Overhead der Score-P-Instrumentalisierung so groß, dass die Messungen unweigerlich stark verfälscht werden. Der Filter kann den Overhead zwar nicht eliminieren, zumindest aber stark reduzieren.

```
    OtherNumber>::type>::type\ =\ double\]
4   double\ jufire::BurgersPDE<dim>::advection_cell_operator(const\ dealii::
    Tensor<1,\ dim>&,\ const\ dealii::Tensor<1,\ dim>&,\ const\ dealii::Tensor
    <2,\ dim>&,\ const\ dealii::Tensor<2,\ dim>&,\ const\ dealii::Tensor<1,\ dim
    >&,\ const\ double&)\ const\ \[with\ int\ dim\ =\ 2\]
5   T1\ dealii::TensorAccessors::contract3(const\ T2&,\ const\ T3&,\ const\ T4&)\
     \[with\ int\ rank_1\ =\ 1;\ int\ rank_2\ =\ 1;\ int\ dim\ =\ 2;\ T1\ =\
    double;\ T2\ =\ dealii::Tensor<1,\ 2,\ double>;\ T3\ =\ dealii::Tensor<2,\
    2,\ double>;\ T4\ =\ dealii::Tensor<1,\ 2,\ double>\]
6   double\ jufire::BurgersPDE<dim>::lhs_operator(const\ dealii::Tensor<1,\ dim
    >&,\ const\ dealii::Tensor<2,\ dim>&,\ const\ double&,\ const\ dealii::Tensor
    <1,\ dim>&,\ const\ dealii::Tensor<2,\ dim>&,\ const\ double&,\ const\ double
    &,\ const\ dealii::Tensor<1,\ dim>&,\ const\ double&,\ const\ double&)\ const
```

```
    \ \[with\ int\ dim\ =\ 2\]
SCOREP_REGION_NAMES_END
```

Die langen Funktionsnamen ergeben sich dabei daraus, dass es sich um C++-Templates handelt, die dimensionsunabhängige Programmierung erlauben und zudem mehrere Parameter von der aufrufenden Funktion erhalten. Zu beachten ist dabei, dass Leerzeichen ebenso wie einige Sonderzeichen escapt werden müssen. Andernfalls würden sie als Beginn einer neuen Funktion interpretier und entsprechend behandelt.
Obwohl nur vier Funktionen, die zum Aufstellen des Gleichungssystems genutzt werden, ausgefiltert wurden, konnte der Overhead so auf ca. 50 Prozent der ursprünglichen Simulation ohne Instrumentalisierung reduziert werden. Dies ist immer noch vergleichsweise hoch – als optimal gelten 10 bis 20% – aber bedeutet bereits eine Reduzierung um den Faktor 20. Mit diesem Overhead ist es möglich, Rückschlüsse auf das Programmverhalten ohne Instrumentalisierung zu ziehen. Sie sollten allerdings weiterhin mit einer gewissen Vorsicht betrachtet werden.

4.3.1 Auswertung

Miteinander verglichen wurden insbesondere die Simulationen des 2D-Vortexproblems mit einem, 16 und 128 Kernen. Diese wurden gewählt, da sie unterschiedliche Phasen im Parallelisierungsverhalten des Programms beschreiben. Die Simulation mit einem Kern steht für die serielle Berechnung, die stets als Referenzwert hinzugezogen wird um Kosten und Gewinn einer Parallelisierung zu betrachten. Wie in Abbildung 4.3 ersichtlich zeigt die Simulation für 16 Kerne noch ein gutes Parallelisierungsverhalten, während sie bei 128 Kernen bereits drastisch abfällt.

Dabei wird ersichtlich, dass die summierte Laufzeit aller an den parallelen Berechnungen beteiligten Kerne deutlich über der Laufzeit der seriellen Berechnung

liegt. Dies war angesichts des Speedups zu erwarten – wie bereits in Kapitel 1 angerissen reduziert eine Parallelisierung den Ressourcenverbrauch nicht, sondern verteilt ihn lediglich auf mehr Kerne. Durch die notwendige Kommunikation entsteht noch ein zusätzlicher Overhead – die aufsummierte Laufzeit steigt.
Das Ausmaß ist allerdings durchaus erheblich: Die Simulation mit 128 Kernen hat laut Scalasca-Messung eine summierte Laufzeit, die etwa dem Fünffachen der seriellen Laufzeit entspricht.
Ebenso gibt es durch die Aufteilung in mehrere Teilprobleme mehr Funktionsaufrufe, von $1.85 \cdot 10^9$ Aufrufen mit einem Kern über $2.15 \cdot 10^9$ Aufrufe mit 16 Kernen bis $22.99 \cdot 10^9$ Aufrufe mit 128 Kernen. Diese zusätzlichen Aufrufe können hauptsächlich verwaltenden Funktionen wie der AMR-Gitterverwaltung zugeordnet werden. Die Anzahl der Aufrufe von Funktionen, die Berechnungen anstellen steigt nur geringfügig.

Wie erwartet wird bei erhöhter Parallelität auch relativ gesehen mehr Zeit auf die mit AMR assozierten Funktionen verwendet. Während die Gesamtlaufzeit der mit 128 Kernen parallelisierten Simulation im Verhältnis zur seriellen Simulation etwa um Faktor 5 stieg, nahm die auf die Funktion `adaptive_refine()` mit Faktor 6 zu, von 110.7 s im seriellen Fall auf 680.59 s mit 128 Kernen. Diese Funktion ist in JuFire für die MARK- und REFINE-Phasen des AMR-Zyklus verantwortlich (siehe 3.1), umfasst also einen Großteil des AMR-Aufwands und insbesondere die Neuaufteilung des Gitters nach jedem Schritt sowie die Verteilung auf die beteiligten Kerne.
Einen erheblichen Anteil dieses Zuwachses machen die im Zuge dessen genutzten MPI-Routinen aus, insbesondere `MPI_Allgather`, welches allein 76.39 Sekunden beansprucht, gefolgt von `MPI_Allreduce` mit 17.96 Sekunden. Bei beiden Routinen handelt es sich um kollektive Kommunikation, bei der Information zwischen einer Gruppe von Kernen ausgetauscht wird. Im Gegensatz dazu wird bei der Punkt-zu-Punkt-Kommunikation jeweils nur zwischen zwei Kernen kommuniziert.
Kollektive Kommunikation ist besonders anfällig für Wartezeiten, da sie eine Synchronisation erzwingt. Das bedeutet, dass alle beteiligten Kerne – in diesem Fall

bis zu 128 – so lange warten, bis auch der langsamste den Befehl erreicht hat, der die Synchronisation ausgelöst hat.
Die beiden verwendeten Routinen zeichnen sich zudem dadurch aus, dass sie nicht nur von allen Kernen Daten empfangen sondern die gesammelten Daten anschließend wieder an alle Prozesse senden. Es sind also im Prinzip zwei kollektive Kommunikationsoperationen in Folge. Das Risiko für Wartezeiten wird hierdurch weiter erhöht.

Abgesehen davon ähneln sich die die Simulationen mit und ohne Parallelisierung in der Betrachtung mit Scalasca weitgehend. Die auf die verschiedenen Funktionen entfallende Laufzeit entspricht in ihrem Anteil an der Gesamtlaufzeit im Wesentlichen dem, was auch bei der seriellen Simulation beobachtet werden kann.
Der Anstieg der Laufzeit und somit auch das Abflachen des Speedups scheint also maßgeblich in Programmteilen verursacht zu werden, die mit der adaptiven Gitterverfeinerung zusammen hängen. Insbesondere MPI-Routinen zur kollektiven Kommunikation tragen hierzu vermutlicherheblich bei.

Eine Bestätigung der Annahme, dass die kollektiven Routinen zu erheblichen Wartezeiten führen, würde allerdings eine Tracing-Analyse erfordern. Diese bietet weit detailliertere Auskünfte über das Programmverhalten und insbesondere die Kommunikationsmuster zwischen den Kernen.
Scalasa unterstützt zwar auch Tracing, eine entsprechende Analyse wurde aber aufgrund des deutlich erhöhten Overheads und dem Umstand, dass man wissen sollte, wonach man mit Scalasca sucht, im Rahmen dieser Arbeit nicht vorgenommen.

Kapitel 5

Fazit

5.1 Zusammenfassung

Im Rahmen der Arbeit wurde der AMR-Teil des JuFire-Codes um zwei weitere Verfeinerungsstrategien ergänzt: die Fixed-Fraction-Strategie sowie die Fixed-Number-Strategie. Diese Erweiterung erlaubt eine flexiblere Anpassung des Programms an die bekannten Eigenschaften des zu simulierenden Problems.
Des Weiteren wurden auch verbesserte Möglichkeiten zur Messung des Effekts von AMR-Strategien implementiert. Dabei handelt es sich insbesondere um die Option, sich ausgeben zu lassen, wie viele Zellen das Gitter im aktuellen Simulationsschritt umfasst, wie viele davon zur Verfeinerung beziehungsweise Vergröberung markiert wurden und eine Schätzung, welche Anzahl Zellen der kommende Schritt anhand dieser Werte umfassen wird sowie die Ausgabe der aktuellen Laufzeit (CPU-Runtime und Walltime, vgl. 3.3.2). Dies ermöglicht es, die Verfeinerung des Gitters auch im Bezug auf die Laufzeit nachzuvollziehen und erleichtert die Auswertung der Zellenzahl erheblich – zuvor bestand lediglich die Möglichkeit, sich die Gitterdaten als Visualisierungsdatei ausgeben zu lassen, welche umständlicher auszuwerten ist.

Bei der Bewertung der Verfeinerungsstrategien zeichneten sich zwei zentrale Eigenschaften ab. Die Laufzeit der Simulation resultiert aus der Anzahl der Zellen

im Gitter und bestimmt maßgeblich die Simulationskosten. Der Fehler der Simulation hingegen ergibt sich eher aus Anordnung und Feinheit der Zellen als aus der Anzahl – mehr Zellen begünstigen es, dass die relevanten Simulationsgebiete fein aufgelöst sind, aber es ergibt sich nicht zwingend.
Die genaue Gewichtung dieser Faktoren hängt von den Zielen der Simulation ab. Ebenso können andere Vorgaben die Wahl der optimalen Verfeinerungsstrategie beeinflussen, etwa der Wunsch nach einer möglichst konstanten Zellenzahl. Im Allgemeinen wird allerdings meist angestrebt, den Fehler möglichst nah an der globalen Verfeinerung halten und unter dieser Vorgabe die Laufzeit soweit wie möglich zu reduzieren. Die Laufzeit sollte dabei um einen größeren Faktor reduziert werden als der Fehler wächst.
Dabei zeichnete sich vor allem die ursprüngliche Strategie mit sowohl Verfeinerungs- als auch Vergröberungsschwellwert 0.3 als guter Kompromiss aus Genauigkeit und Kosten ab. Aus diesem Grund konzentrierte sich die Betrachtung der Parallelität auf diese Strategie.

Dabei zeigte sich, dass JuFire ohne aktivierte adaptive Gitterverfeinerung gut skaliert sofern die Problemgröße angemessen für die Anzahl der verwendeten Kerne gewählt wird. Andernfalls überwiegt der Kommunikationsoverhead schnell den Nutzen der Parallelisierung.
Mit aktiviertem AMR ist hingegen eine Beschränkung des Speedups zu erkennen. Diese kann durch Amdahls Gesetz beschrieben werden und deutet auf einen erhöhten seriellen Programmanteil hin. Vermutlich entsteht dieser serielle Anteil durch den Zusatzaufwand für das Markieren der Zellen und das Anpassen des Gitters im Rahmen der adaptiven Gitterverfeinerung.
Die Betrachtung mit Scalasca, einem Performanceanalyse-Tool für parallele Programme, bestätigt, dass in parallelen Berechnungen verhältnismäßig mehr Zeit in mit AMR assozierten Routinen verbracht wird als im seriellen Programm. Ein großer Anteil davon entfällt auf MPI-Routinen zur kollektiven Kommunikation. Es wird daher vermutet, dass diese Routinen Wartezeiten der Kerne verursachen.

5.2 Ausblick

Die Entwicklung JuFires ist noch nicht abgeschlossen und insbesondere im Bereich der adaptiven Gitterverfeinerung sind noch viele Erweiterungsmöglichkeiten offen. Im Rahmen dieser Arbeit wurde erwogen, weitere Verfeinerungsindikatoren, beispielsweise basierend auf der Richardson-Interpolation wie von Marsha Berger beschrieben [BO83], zu implementieren. Ebenso wäre es interessant, die von Richter beschriebene optimierte Verfeinerungsstrategie um die Möglichkeit Zellen zur Vergröberung zu markieren zu erweitern und somit für eine größere Auswahl an Problemen – darunter auch das 2D-Vortex-Problem – nutzbar zu machen.
Auf beides musste in dieser Arbeit aus zeitlichen Gründen verzichtet werden. Es nachzuholen würde die Möglichkeiten des Programms in großem Umfang erweitern.

Auch ohne Veränderungen des Quellcodes gibt es noch viele Optionen. Ein zentrales Anliegen wäre hier, das Verhalten der so genannten ürsprünglichen Strategie"(siehe 3.2.4) genauer zu untersuchen und zu dokumentieren, da auf Basis der initialen Ziele der Implementierung der Unterschied zum Verhalten der Fixed-Fraction-Strategie (siehe 3.2.2) nicht vollständig erklärt werden kann.
Naheliegend wäre es darüber hinaus auch, die Analysen umfangreicher zu gestalten. Für den Vergleich der AMR-Strategien wurden nun lediglich zwei Verfeinerungsschwellwerte betrachtet, eine umfangreichere Studie zu den Auswirkungen verschiedener Schwellwerte auf das Verhalten der Strategien wäre also naheliegend – insbesondere da bei der ursprünglichen Strategie, die den besten Kompromiss aus Genauigkeit und Laufzeitreduzierung aufwies, das Verhalten bei Parameterwechseln bisher kaum bekannt ist. Ebenso würde es sich anbieten, mehr verschiedene Strategien und entsprechend auch unterschiedliche Schwellwerte auch im Hinblick auf ihre parallele Performance mit der nun betrachteten ursprünglichen Strategie zu vergleichen.
Um die Ergebnisse der Arbeit noch weiter zu bekräftigen wäre es außerdem sinnvoll, Betrachtungen zur parallelen Leistung mit aktiviertem AMR auch mit größe-

ren Problemen durchzuführen sowie eine Tracing-Analyse mit Scalasca oder anderen Tools vorzunehmen.

Zuletzt wurde bei längeren parallelen Simulationen des 2D-Vortex-Problems das Entstehen von Ungenauigkeiten außerhalb der Laufbahn des Vortex beobachtet. Diese haben einen erheblichen Einfluss auf die Verfeinerung des Gitters und ziehen Zellen – und damit Rechenressourcen – von dem tatsächlich zu simulierenden Bereich ab. Bei serieller Berechnung scheinen sie hingegen nicht aufzutreten. Da Parallelität gerade bei umfangreichen Simulationen ein wichtiges Hilfsmittel zur Reduzierung der Laufzeit ist, wäre es ein wichtiges Anliegen, die Ursache dieser Ungenauigkeiten weiter zu erforschen und zu eliminieren.

Literaturverzeichnis

[Amd67] Amdahl, G.: Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In: *AFIPS Conference Proceedings* 30 (1967)

[Ban13] Bangerth, W.: *MATH 676: Finite element methods in scientific computing course*. `http://www.math.colostate.edu/~bangerth/videos.html`. Version: 2013. – aufgerufen am 04.11.2017

[BBHK11] Bangerth, W. ; Burstedde, C. ; Heister, T. ; Kronbichler, M.: Algorithms and Data Structures for Massively Parallel Generic Adaptive Finite Element Codes. In: *ACM Transactions on Mathematical Software* 38 (2011), Nr. 2, S. 14:1–14:28

[BC89] Berger, M.J. ; Colella, P.: Local Adaptive Mesh Refinement for Shock Hydrodynamics. In: *Journal of Computational Physics* 82 (1989)

[BHK07] Bangerth, W. ; Hartmann, R. ; Kanschat, G.: deal.II – a General Purpose Object Oriented Finite Element Library. In: *ACM Transactions on Mathematical Software* 33 (2007), Nr. 4, S. 24/1–24/27

[BO83] Berger, M.J. ; Oliger, J.: Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. In: *Journal of Computational Physics* 53 (1983)

[CM00] Chorin, A. ; Marsden, J.-E.: *A Mathematical Introduction to Fluid Mechanics*. Springer, 2000

[dea] deal.II: *The step-7 tutorial program*. https://www.dealii.org/8.4.0/doxygen/deal.II/step_7.html. – aufgerufen am 08.11.2017

[Eij14] Eijkhout, V.: *Introduction to High Performance Scientific Computing*. 2nd Edition. 2014

[FBA17] Fehling, M. ; Boltersdorf, J. ; Arnold, L.: *Towards smoke and fire simulation with grid adaptive FEM: Verification of the flow solver*. http://juser.fz-juelich.de/record/834633/files/2017_IAFSS_AMR.pdf. Version: 2017. – 12th International Symposium on Fire Safety Science

[FHV02] Farkas, I. ; Helbing, D. ; Vicsekd, T.: Social behaviour: Mexican waves in an excitable medium. In: *Nature* 419 (2002), September, S. 131–132

[For17] Forschungszentrum Jülich GmbH: *JURECA – Configuration*. http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/Configuration/Configuration_node.html. Version: 2017. – aufgerufen am 07.11.2017

[GWW+10] Geimer, Markus ; Wolf, Felix ; Wylie, Brian J. N. ; Ábrahám, Erika ; Becker, Daniel ; Mohr, Bernd: The Scalasca performance toolset architecture. In: *Concurrency and Computation: Practice and Experience* 22 (2010), April, Nr. 6, S. 702–719

[Jou10] Jouhaud, J.-C.: *Benchmark on the vortex preservation*. http://elearning.cerfacs.fr/pdfs/numerical/TestCaseVortex2D.pdf. Version: 2010. – aufgerufen am 07.11.2017

[Mat10] Mathiak, F. U.: *Die Methode der finiten Elemente (FEM): Einführung und Grundlagen*. 2010

[MHM+17] McGrattan, K. ; Hostikka, S. ; McDermott, R. ; Floyd, J. ; Weinschenk, C. ; Overholt, K.: *Fire Dynamics Simulator Technical Reference Guide Volume 2: Verification*. 6th Edition. 2017

[Rah94] Rahm, E.: *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung*. 1. Ausgabe. 1994

[Sma63] Smagorinsky, J.: General Circulation Experiments with the Primitive Equations. In: *Monthly Weather Review* 91 (1963), Nr. 3, S. 99–164

[The16] The Scalasca Development Team: *Scalasca 2.3 User Guide*. `https://apps.fz-juelich.de/scalasca/releases/scalasca/2.3/docs/UserGuide.pdf`. Version: 2016. – aufgerufen am 08.11.2017