FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Parallelisierung und Optimierung eines Monte-Carlo-Transportmodells für Kernfusionsanwendungen

Andreas Galonska

FZJ-JSC-IB-2008-08

November 2008

(letzte Änderung: 21.11.2008)

Parallelisierung und Optimierung eines Monte-Carlo-Transportmodells für Kernfusionsanwendungen

Andreas Galonska

21. November 2008

Fachhochschule Aachen, Abteilung Jülich

Fachbereich: Medizintechnik und Technomathematik
Studiengang: Technomathematik

Die vorliegende Diplomarbeit wurde in Zusammenarbeit mit dem Forschungszentrum Jülich GmbH, Jülich Supercomputing Centre, angefertigt.

Diese Diplomarbeit wurde betreut von:

Referent: Prof. Dr. rer. nat. J. Hoffmann (Fachhochschule Aachen)

Koreferent: Dr. habil. P. Gibbon (Forschungszentrum Jülich)

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

(Andreas Galonska)
21. November 2008

Zusammenfassung

In dieser Diplomarbeit wird die bestehende Parallelisierung eines Programms zur Simulation von Transportvorgängen in Kernfusionsexperimenten untersucht und optimiert. Zunächst wird, aufbauend auf dem Standard MPI, eine parametrisch parallelisierte Version des Programms, die den Anwender bei der Parametervariation unterstützt, geschrieben. Dies geschieht getrennt von der späteren Parallelisierung mit OpenMP und hat nur minimalen Einfluß auf die Performance des Programms. Jedoch bietet dies den Vorteil, verschiedene Parametervariationen gleichzeitig zu simulieren und trotzdem durch den Einsatz vieler Prozessoren die Simulation deutlich zu beschleunigen. Hier können beide Parallelisierungen ineinander greifen.
Das Programm behandelt mehrere Simulationsszenarien, welche die verschiedenen Vorgänge wie Teilchentransport, Ionisation, Ablagerung und Erosion darstellen. Eines der am häufigsten verwendeten Szenarien, der Transport von extern eingeführten Teilchen(„Case 4“), zeigt eine besonders schlechte Parallelisierung. Um der Ursache auf den Grund zu gehen werden Werkzeuge wie KOJAK [1] und SCALASCA [2] genutzt, die Performance-Engpässe im Programm identifizieren und analysieren können. Darauf aufbauend werden diese Schwachstellen optimiert, was eine Verbesserung des Speedup um den Faktor 2 zur Folge haben wird.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Computersimulation 1

2 Grundlagen **3**
2.1 Kernfusion 3
2.1.1 Kernfusion als Energiequelle 4
2.1.2 Experimentelle Kernfusion 5
2.1.3 Kernfusionsforschung am Forschungszentrum Jülich 8
2.2 IBM Power6 575 Cluster - JUMP 9
2.3 Monte-Carlo-Simulation 9
2.4 Parallele Programmierung 10
2.4.1 MPI 11
2.4.2 OpenMP 12
2.4.3 Kenngrößen paralleler Programmierung 14

3 Das Monte-Carlo-Programm ERO **16**
3.1 Programmbeschreibung 16
3.1.1 Eingabeparameter 18
3.1.2 Programmablauf 19

4 Parametrische Parallelisierung **20**
4.1 Motivation 20
4.2 Startprogramm zur Parametervariation in ERO : RunERO 20
4.3 Parallelisierung 21
4.4 Vorteile der parametrischen Parallelisierung 22

5 Analyse und Optimierung der Parallelisierung mit OpenMP **24**
5.1 Laufzeitverhalten beim Transport von extern eingeführten Teilchen (Case 4, vgl. Tabelle 1) 26
5.2 Analyse und Optimierung 30
5.2.1 Methode `PartGo` 30
5.2.2 Methode `sort_in` 34
5.2.3 Lastverteilung des OpenMP Schedulers 37
5.3 Resultate der Optimierung 38
5.3.1 `PartGo` 38
5.3.2 `sort_in` 41
5.3.3 `PartGo` und `sort_in` im Kontext 47
5.3.4 Optimierung der Lastverteilung 48
5.3.5 Vergleich der Laufzeiten 52

6 Zusammenfassung und Ausblick **53**
6.1 Zusammenfassung 53
6.2 Ausblick 54

Tabellenverzeichnis

1 Simulationsfälle in ERO 18
2 Beispielwerte der Indextransformation 36
3 Auslastung/ % der Methode `sort_in` bei 16 Threads, 30000 Teilchen 44
4 Auslastung / % bei 32 Threads nach Optimierung der Lastverteilung, 30000 Teilchen 50

Abbildungsverzeichnis

1 Bindungsenergie in Abhängigkeit von der Massenzahl 4
2 Schematik eines Tokamak-Systems 6
3 Beispielhafte Darstellung der physikalischen Prozesse in ERO. 16
4 Serieller Programmablauf eines LoadLeveler-Jobs 19
5 Parametrisch parallelisierter Programmablauf von ERO. Prozess i simuliert Szenario i 23
6 Laufzeitverhalten der untersuchten parallelen Region in Case 4 27
7 CUBE Visualisierung der SCALASCA Analyse von `PartGo` für eine Simulation mit 10000 Teilchen auf 16 Threads 29
8 Ursprünglicher Programmablauf von `PartGo` 31
9 CUBE Visualisierung der SCALASCA Analyse von `sort_in` für eine Simulation mit 10000 Teilchen auf 16 Threads 32
10 Programmablauf der modifizierten Version von `PartGo` mit Trajektoriensammlung in jedem Schritt. 33
11 Schematische Darstellung der Konstruktion der Speicherbereiche 35
11.1 Optimierter Speicherbereich 35
11.2 Optimiertes Gitter mit Randgebiet 35
12 Speedup der parallelen Region in Case 4 nach der Optimierung von `PartGo` 39
13 CUBE Visualisierung der SCALASCA Analyse von `PartGo` nach der Optimierung (10000 Teilchen, 16 Threads) 40
14 Speedup der parallelen Region um `sort_in` nach der Optimierung 42
15 CUBE Visualisierung der SCALASCA Analyse von `sort_in` nach der Optimierung (10000 Teilchen, 16 Threads) 43
16 Speedup der parallelen Region von Case 4 mit vollständiger Optimierung von `PartGo` und `sort_in`, Simulation von 1000 Teilchen 46
17 Vergleich der verschiedenen Optimierungen mit der Ursprungsversion 47
18 Speedup der parallelen Region von Case 4 nach Optimierung der Lastverteilung, Simulation von 30000 Teilchen 48
19 Zeitbedarf von ERO nach erfolgter Optimierung 52

1 Einleitung

1.1 Computersimulation

Die Computersimulation entwickelt sich, neben der Theorie und dem experimentellen Nachweis, zum dritten Standbein der Wissenschaft. Theoretische Grundlagen bilden hierbei die Basis, ein existierendes Problem mit Hilfe numerischer Modelle zu beschreiben und dadurch in der Lage zu sein, experimentelle Ergebnisse vorauszusagen. Dies ist oft eine kostengünstige Alternative zu praktischen Versuchen, die einerseits teuer, aufwendig oder aber praktisch nicht durchführbar sind. Computersimulation bietet heutigen Supercomputern ein weites Feld an Einsatzmöglichkeiten und der Wissenschaft ein wertvolles Werkzeug in der Forschung. Die Modellierung einer Simulation hilft oft dabei, das Zusammenspiel schon bekannter Prozesse besser zu verstehen. Auf der anderen Seite werden solche Simulationen durch immer feinere Diskretisierung teurer und der Rechenaufwand steigt damit in hohem Maße an. Für normale Einzelrechner stellen einige moderne physikalische Simulationen mittlerweile eine zu hohe Komplexitätsklasse dar, die von ihnen nicht in einer annehmbaren Zeit bewerkstelligt werden kann, weshalb man sich an ihrer Stelle der Supercomputer bedient. Diese stellen ihrerseits enorme Rechenkapazität zur Verfügung, um auch Probleme zu lösen, denen normale Rechner nicht gewachsen sind, wollen aber auch effektiv genutzt werden, was eine effiziente Parallelisierung notwendig macht.
Ein entscheidendes Kriterium für diese Effizienz ist dabei die Skalierbarkeit eines parallelen Programms, welche durch seinen seriellen Anteil begrenzt wird. Weiterhin wird die Effizienz vom Datenaustausch zwischen den beteiligten Prozessoren und durch ihre Synchronisation beeinflusst. Ziel einer effizienten Parallelisierung kann es dabei nur sein, den Speedup der Anwendung unter Nutzung mehrerer Prozessoren dahin gehend zu optimieren, eine gute Lastverteilung zu erreichen und den seriellen Overhead möglichst gering zu halten.
Einen wesentlichen Aspekt eines parallelen Programms stellt der Kommunikationsaufwand dar, den die beteiligten Prozessoren beispielsweise auf asymmetrischen Multiprozessorsystemen leisten müssen. Dieser sollte gering gehalten werden, da Kommunikation im Vergleich zur eigentlichen Rechenzeit sehr aufwendig ist oder so gestaltet sein, dass die Prozessoren sich nicht gegenseitig blockieren und weiter rechnen können, während andere Prozessoren kommunizieren. Um dieses Ziel zu erreichen steht Entwicklern paralleler Programme eine Vielzahl an Werkzeugen zur Programmanalyse, Profilierung und Instrumentation zur Verfügung, mit denen sich das Verhalten des Programms genau nachvollziehen lässt. Dies versetzt einen in die Lage, Defizite im Programmablauf, schlechte Ausbalancierung der beteiligten Prozessoren, hohen Kommunikationsoverhead und ungenügende

Auslastung zu erkennen, zu analysieren um damit die Performance des Programms, durch gezielte Elimination dieser leistungshemmenden Faktoren, zu steigern. Das gewünschte Resultat einer solchen Optimierung sollte ein möglichst linearer Speedup des Programms, bei Steigerung der Anzahl beteiligter Prozessoren, und damit eine gute Effizienz, sein. Diese stellt eine Maßzahl zur Verfügung, über die sich sowohl der Speedup, als auch der Kostenfaktor beschreiben lässt.

2 Grundlagen

In diesem Kapitel wird auf die zum Verständnis dieser Arbeit notwendigen Kenngrössen und Begrifflichkeiten eingegangen. Die Fragestellungen der Softwareentwicklung und Programmierung stellen den Schwerpunkt der Arbeit dar. Die physikalischen Grundlagen der Kernfusion werden daher nur kurz beschrieben.

2.1 Kernfusion

Als Kernfusion bezeichnet man die Verschmelzung zweier Atomkerne zu einem neuen Atomkern. Sie kann sowohl endotherm als auch exotherm ablaufen, was maßgeblich von den zur Fusion verwendeten Elementen abhängt (Abbildung 1). Da Atomkerne aus Neutronen (Nukleonen ohne elektrische Ladung) und Protonen (Nukleonen mit positiver elektrischer Ladung) bestehen, besitzen diese eine positive Ladung. Nach den Gesetzen der Elektrostatik stoßen sich gleichnamige Ladungen ab und auf sie wirkt die fern wirkende Coulombkraft. Umgekehrt wirkt bei sehr kleinen Abständen zwischen den Nukleonen, wie dies in einem Atomkern der Fall ist, die auf kurzen Distanzen bestehende Kernkraft, welche die Nukleonen zusammen hält. Möchte man nun zwei oder mehr gleich geladene Teilchen auf einen kürzeren Abstand bringen, muss man Arbeit entgegen der Coulombkraft verrichten, welche mit sinkendem Abstand jedoch immer grösser wird. Es gilt, eine Potenzialbarriere zu überwinden, die auch Coulomb-Barriere genannt wird. Nach den Gesetzen der klassischen Mechanik wäre dieses Unterfangen unmöglich, da die Coulombkraft mit sinkendem Abstand der Teilchen gegen unendlich strebt. Der quantenphysikalische Tunneleffekt postuliert jedoch eine gewisse Wahrscheinlichkeit, mit der Teilchen doch in der Lage sind, diese Barriere zu überwinden. Dadurch ist es den Teilchen möglich, in den Einflussbereich der Kernkraft zu gelangen. Der Abstand, bei dem dies geschieht beträgt etwa 10^{-15} m. Die Kernfusion läuft nur bei der Verschmelzung von leichten Kernen, also bei Kernen mit einer geringen Bindungsenergie, exotherm ab und ist daher zur Energiegewinnung auch nur hier sinnvoll.
In Abbildung 1 ist zu erkennen, dass bei der Verschmelzung von Deuterium ($^2_1$H) und Tritium ($^3_1$H) erzeugtes Helium ($^4_2$He) eine höhere Bindungsenergie als das Ausgangsmaterial besitzt. Diese Energie wird bei der Fusion freigesetzt und äussert sich zudem in einem Masseverlust des entstehenden Kerns. Dieser ist nämlich leichter als die Summe der Massen seiner Nukleonen. Der Massendefekt lässt sich nach Einstein aus der Äquivalenz von Masse und Energie, $E = mc^2$, erklären.

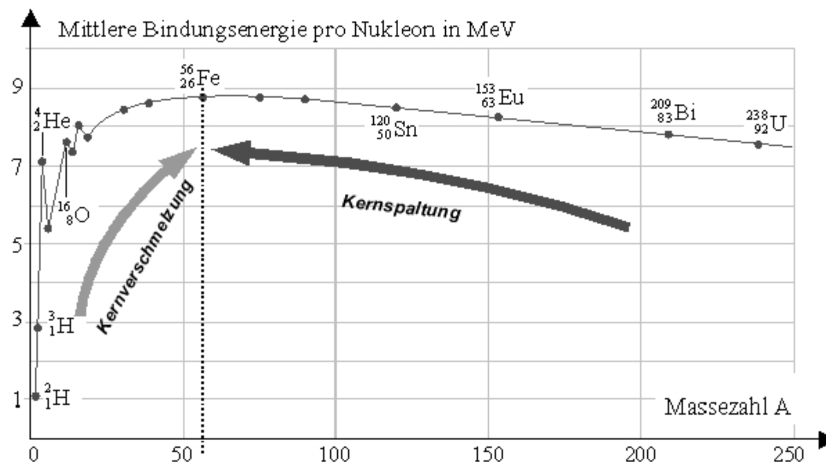
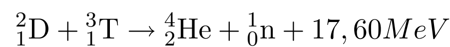


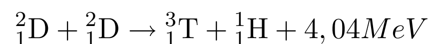
Abbildung 1: Bindungsenergie in Abhängigkeit von der Massenzahl

2.1.1 Kernfusion als Energiequelle

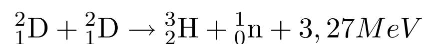
Sterne, wie unsere Sonne, nutzen die Kernfusion als Energiequelle. In ihr werden verschiedene leichte Elemente, aufgrund des hohen Drucks und der hohen Temperatur, zu schwereren Elementen verschmolzen. Unten sind einige dieser Reaktionen aufgeführt.



$$^{2}_{1}\mathrm{D} + ^{3}_{1}\mathrm{T} \rightarrow ^{4}_{2}\mathrm{He} + ^{1}_{0}\mathrm{n} + 17,60 MeV$$



$$^{2}_{1}\mathrm{D} + ^{2}_{1}\mathrm{D} \rightarrow ^{3}_{1}\mathrm{T} + ^{1}_{1}\mathrm{H} + 4,04 MeV$$



$$^{2}_{1}\mathrm{D} + ^{2}_{1}\mathrm{D} \rightarrow ^{3}_{2}\mathrm{H} + ^{1}_{0}\mathrm{n} + 3,27 MeV$$

$^{2}_{1}\mathrm{D}$: Deuterium $\widehat{=}$ $^{2}_{1}\mathrm{H}$
$^{3}_{1}\mathrm{T}$: Tritium $\widehat{=}$ $^{3}_{1}\mathrm{H}$
$^{4}_{2}\mathrm{He}$: Helium (α -Teilchen)
$^{1}_{0}\mathrm{n}$: Neutron
(Quelle: [3])

Die für diese Reaktionen notwendige Zündungsenergie ergibt sich in Sternen aufgrund des Gravitationsdrucks ($> 2 \cdot 10^{16}$ Pa im Zentrum) und der Temperatur. Sie beträgt in unserer Sonne ca. 10 Millionen °C. Ein entscheidendes Kriterium für die Aufrechterhaltung einer

sich selbst tragenden Kernfusion hier auf der Erde ist die Reaktionsrate der zu verschmelzenden Elemente. Sie hängt von deren Wirkungsquerschnitt und der Temperatur ab, wobei der Wirkungsquerschnitt ein Maß für die Wahrscheinlichkeit ist, mit der ein bestimmtes Teilchen mit einem anderen reagiert. Die Bedingungen sind bei einer Deuterium(^2H) - Tritium(^3H) Reaktion am günstigsten, da hier die relative Reaktionsrate von allen in Frage kommenden Teilchenkombinationen am höchsten ist [4]. Das Lawson-Kriterium (Ungleichung 1) liefert eine Definition der Zeit, die der Plasmaeinschluß zusammen gehalten werden muss, damit eine sich selbst erhaltende Fusionsreaktion stattfindet, die Energie liefert.

$$n\tau > \frac{12kT}{\overline{\sigma v} \cdot U_f} \sim 10^{15} \frac{s}{cm^3} \tag{1}$$

n: Teilchendichte
τ: Einschlusszeit
k: Boltzmannkonstante
T: Temperatur
$\overline{\sigma v}$: Über die Teilchengeschwindigkeit gemittelte Reaktionsrate
U_f: Bindungsenergie der Teilchen $\widehat{=}$ frei werdende Reaktionsenergie

Diese Einschlusszeit ist notwendig, damit die bei der Fusion freigesetzten Heliumkerne dem dichten Plasma genug Energie zuführen können, um Energieverluste durch Wärmeleitung, Konvektion und Abstrahlung zu kompensieren.

2.1.2 Experimentelle Kernfusion

Trotz der relativ hohen Hürde, welche das Lawson-Kriterium darstellt, ist es auf der Erde bereits gelungen eine Kernfusion zu erzeugen. Es gibt bereits mehrere Fusionsexperimente, die beispielsweise nach dem Tokamak-Prinzip arbeiten. Allerdings arbeiten diese Systeme nicht wirschaftlich (Ausdruck 2), da die zu ihrem Betrieb notwendige Energie die Energie, welche bei der Fusion freigesetzt wird, übersteigt.

$$Q = \frac{P_{Fusion}}{P_{Heiz}} \tag{2}$$

Q: Leistungsverstärkung
P_{Heiz}: extern zugeführte Heizleistung
P_{Fusion}: Fusionsleistung

Dabei bedeutet $Q < 1$, dass weniger Energie freigesetzt als dem System zugeführt wurde, was bei allen, heute zu experimentellen Zwecken eingesetzten Systemen, der Fall ist. Das nach dem Tokamak-Prinzip arbeitende Kernfusionsexperiment ITER soll das erste mal demonstrieren, dass Energieausbeuten mit $Q \geq 10$ möglich sind, was die Kernfusion wirtschaftlich sinnvoll macht.
Eine Zündung der selbst-tragenden Fusionsreaktion hier auf der Erde bedingt durch den viel geringeren Druck eine Erhöhung der Temperatur, um die Dichte zu erzeugen, die hierfür notwendig ist. Sie liegt bei ca. 50 Millionen °C. Das hitzebeständigste Element, Wolfram, hat einen Schmelzpunkt von ca. 3400 °C. Deshalb ist es unmöglich, das Fusionsplasma mechanisch einzuschließen. Da Plasma aber aus Ionen besteht, gibt es die

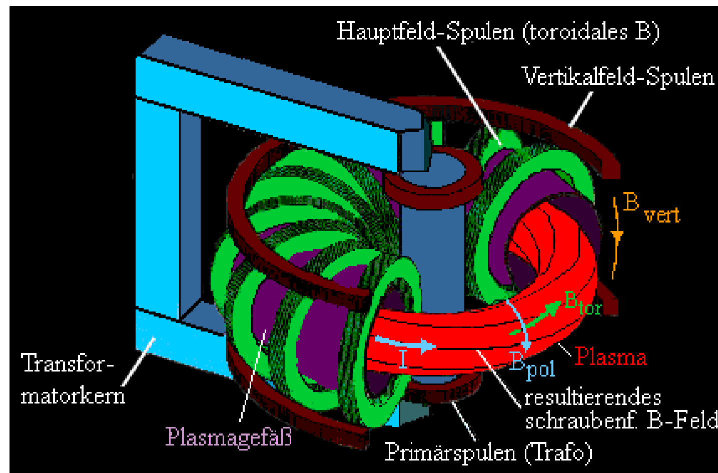


Abbildung 2: Schematik eines Tokamak-Systems

Möglichkeit eines helikalen, magnetischen Plasmaeinschlusses, wie dies zum Beispiel in einem Tokamak oder Stellarator System geschieht. Dabei wird das Plasma von torusförmig angeordneten Spulen in einem Ring „gehalten“ . In Abbildung 2 ist die Schematik eines Tokamak-Systems dargestellt.

Man sieht, dass die torusförmig angeordneten Spulen das toroidale Magnetfeld B_{tor} erzeugen.
Durch die verschiedenen Bahnradien der Teilchen, die durch die Breite des Torus bedingt sind, ergibt sich eine Ladungstrennung, die zu einer Störung des Plasmaeinschlusses führt. Daher ist es notwendig, dass die Magnetfeldlinien eine poloidale Komponente, B_{pol}, erhalten, so dass sich die resultierenden magnetischen Feldlinien verdrillen. In einem Tokamak wird die Verdrillung der Magentfeldlinien durch Induktion eines Stromes in das Plasma erzeugt, wogegen bei einem Stellarator die geometrische Anordung der Spulen für die Verdrillung sorgt. Solch ein Stellarator ist etwa der Wendelstein 7-X, der derzeit im Max-Planck Institut für Plasmaphysik in Greifswald gebaut wird und bis 2014 fertig gestellt sein soll [5].
Zur Zündung muss das Fusionsplasma aufgeheizt werden. Dies geschieht durch verschiedene Mechanismen:

- Neutralteilcheninjektion
  In das Plasma wird ein Strahl schneller Neutralen geschossen, die durch ihre Bewegungsenergie und Stösse zwischen den Teilchen das Plasma aufheizen. Dies ist auch die einzige Möglichkeit neuen Brennstoff in die Fusionskammer einzubringen.

- Mikrowellenheizung
  Durch Einkopplung von Mikrowellen in das Plasma, deren Energie auf der Zyklotronfrequenz (Umlauf um die Torusachse) der Teilchen sehr effektiv absorbiert wird, erhöht sich die Temperatur des Plasmas.

- Ohmsche Heizung
  Der durch die Transformationsspule in das Plasma induzierte Strom bewirkt gleichfalls eine Erhöhung der Plasmatemperatur, die jedoch zur Zündung einer selbsttragenden Fusionsreaktion nicht ausreicht.

2.1.3 Kernfusionsforschung am Forschungszentrum Jülich

Die Erforschung neuer Energieträger, die eine Unabhängigkeit von fossilen Brennstoffen voran treiben soll, befasst sich unter anderem mit der Kernfusion zur Energiegewinnung, welche gegenüber der schon seit einem halben Jahrhundert betriebenen Kernspaltung einige Vorteile aber auch Hindernisse bietet, die es zu überwinden gilt. In den letzten Jahren wurden auf diesem Gebiet wesentliche Fortschritte erzielt, die eine kommerzielle Nutzung dieser Energie in Aussicht stellen.
So steht am Forschungszentrum Jülich das Tokamak-Kernfusionsexperiment TEXTOR (Tokamak EXperiment for Technology Oriented Research)[6], welches zu großen Teilen der Erforschung der Plasma-Wand-Wechselwirkung und des Verunreinigungstransports bei der Kernfusion dient. Es soll unter anderem heraus gefunden werden, welche Materialien sinnvoll in der Gefäßwand eines Tokamak eingesetzt werden können, da diese den extremen Temperaturen des Plasmas, welche um die 50 Millionen Grad betragen, ausgesetzt sind und daher hohe Ansprüche erfüllen müssen. Dabei sollen diese Materialien bei Beschuss von Teilchen möglichst wenige Stoffe selber freisetzen, was zu einer Verunreinigung des Plasmas führen würde. Die Prozesse der Plasma-Wand-Wechselwirkung können in TEXTOR an speziellen Wandkomponenten, sogenannten Testlimitern, untersucht werden.
Innerhalb von TEXTOR sind zwei „Limiter Locks“ installiert, womit verschiedene Testlimiter mit eingebautem Injektionskanal an den Plasmarand gefahren werden können. Durch den Kanal werden Teilchen geblasen, und deren Einfluß auf den Limiter beobachtet. Die Erkenntnisse, die sich aus den Simulationen und den praktischen Experimenten in TEXTOR ergeben, sollen in die Entwicklung und Konstruktion des ersten internationalen Kernfusionsexperimentes ITER (International Thermonuclear Experimental Reactor) mit einfließen, das ab 2009 gebaut und dann in etwa 10 Jahren fertig gestellt werden soll [7].
Damit leistet TEXTOR und die Modellierung mit ERO einen wesentlichen Beitrag zur Konstruktion und zum Betrieb von ITER, da hier das Design von kritischen Komponenten erforscht wird, ohne die ein solches Forschungsprojekt undenkbar wäre. Ein einziger Plasmaeinschluß in ITER wird sehr kostenintensiv sein und will daher gut geplant sein, vor allem aufgrund der Aussicht, durch Kernfusion die Energieprobleme der Gegenwart und Zukunft lösen zu können.

2.2 IBM Power6 575 Cluster - JUMP

Die Diplomarbeit wurde auf dem Jülicher Multiprozessor (JUelicher Multi Prozessor) nach seiner Migration auf das Power6 System von IBM durchgeführt.
JUMP besteht aus 14 SMP [1] Knoten mit jeweils 32 SMT [2] Prozessoren, also insgesamt 448 Prozessoren. Innerhalb eines jeden Knoten teilen sich die Prozessoren einen gemeinsamen Adressraum von 128 GByte (1,8 TByte total). Jeder Prozessor taktet mit 4,7 GHz und besitzt folgende Caches:

- **L1:** 64kByte
- **L2:** 8 MByte pro Dual-Core-CPU
- **L3:** 32 MByte (extern, nicht auf der Die[3] integriert)

Die einzelnen Knoten sind über ein schnelles Infiniband Netzwerk gekoppelt und erreichen eine Peak Performance von 8,4 TFLOPS [4] respektive 5,4 TFLOPS im Linpack Benchmark [8].

2.3 Monte-Carlo-Simulation

Die MC-Simulation ist ein in der Teilchenphysik häufig eingesetztes Rechenmodell, um Vorgänge zu simulieren, die „zufällig“ ablaufen sollen [9]. Dabei werden beispielsweise Teilchen generiert, deren Eigenschaften einer vorher festgelegten statistischen Wahrscheinlichkeitsverteilung gehorchen. Auf der anderen Seite könnte auch bei festen Teilcheneigenschaften der Transport dieser Teilchen einer Wahrscheinlichkeitsverteilung folgen. Die simulierten Teilchen, oder genauer gesagt deren Eigenschaften, bilden immer nur eine repräsentative Untermenge der Teilchen, die in der Realität auftreten würden. Um eine Aussage über die zu simulierenden Vorgänge zu treffen reicht dies aber aufgrund des „Starken Gesetzes der größen Zahlen“ vollkommen aus (Ausdruck 3).

[1] Symmetrisches Multiprozessing, vgl. Kap. 2.4.2
[2] Simultaneous Multithreading: Mehrere Berechnungen können hardwareseitig gleichzeitig durchgeführt werden
[3] Die: Prozessorkern
[4] 1 TFLOPS $\widehat{=}$ 10^{12} Gleitkommaoperationen pro Sekunde

Sei $X_1, X_2, X_3, ...$ eine unendliche Folge von Zufallszahlen mit dem gleichen Erwartungswert μ, dann gilt:

$$P(\lim_{n \to \infty} \overline{X}_n = \mu) = 1 \tag{3}$$

mit

$$\overline{X}_n = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{4}$$

P: Wahrscheinlichkeit ($0 \leq P \leq 1$)
$\overline{X}_n$: Arithmetisches Mittel der Zahlenfolge
μ: Erwartungswert der Folge

Für eine hinreichend große Folge an Zufallsgrössen bedeutet dies, dass sie bei steigender Anzahl der Folgenglieder gegen ihren Erwartungswert konvergiert, die Wahrscheinlichkeit für das Eintreten dieses Ereignisses also 1 ist. Das arithmetische Mittel kann jedoch nur über endlichen Folgen gebildet werden, daher kann n nicht gegen ∞ laufen. In der MC-Simulation repräsentieren N Monte-Carlo-Teilchen eine Menge von N_0 realen Teilchen, wobei gilt: $N < N_0$. Durch (3) wird aber für eine genügend grosse Anzahl an Monte-Carlo-Teilchen garantiert, dass diese die realen Teilchen mit hinreichender Genauigkeit repräsentieren können. Genauer gesagt bedeutet dies, dass für eine hinreichend große Menge an Zufallsgrössen, in unserem Fall die Teilchen, garantiert werden kann, dass der statistische Fehler, der sich durch die Nutzung von weniger Teilchen ergibt, als in der Realität auftreten würden, gegen 0 konvergiert und damit vernachlässigbar klein wird. Ziel der MC-Simulation ist es, ein möglichst genaues Abbild der Realität zu simulieren.

2.4 Parallele Programmierung

Unter paralleler Programmierung versteht man den Vorgang, ein Programm so zu gestalten, dass viele Berechnungen gleichzeitig verarbeitet werden können. In seiner einfachsten Form geschieht dies in einem einzelnen Prozessor durch Pipelining. Dazu werden in einem Programm Befehle auf Daten gleichzeitig ausgeführt, die keine gemeinsamen Abhängigkeiten besitzen. Dieser Vorgang kann in der Regel nicht vom Programmierer beeinflusst

werden, da er einerseits eine hohe Komplexität besitzt und auf der anderen Seite von der verwendeten Hardware abhängt. Daher wird diese Aufgabe vom Compiler übernommen, welcher den gegebenen Quelltext in ein Programm übersetzt, in dem diese Art der parallelen Verarbeitung realisiert wird.
Um ein Programm nun nicht nur von einem Prozessor, sondern von beliebig vielen verarbeiten zu lassen, muss man sich verschiedener Programmierparadigmen bedienen. Hier kann man nicht einen speziellen Compiler hinzu ziehen, der den Programmablauf automatisch verteilt. Der Programmierer selbst muss dafür Sorge tragen. Zum Zeitpunkt der Erstellung dieser Arbeit existieren hierfür zwei Standards, deren Einsatz auf zwei verschiedene Typen von Multiprozessorsystemen abzielt.

2.4.1 MPI

Heutige massiv parallele Supercomputer (MMP-Systeme [5]) bestehen aus vielen tausend Prozessoren, die über ein schnelles Verbindungsnetzwerk miteinander kommunizieren können. Die Arbeit dieser Prozessoren muss jedoch gut koordiniert sein, um Situationen zu vermeiden, in denen Rechenzeit nicht effizient genutzt wird. Die Hauptaufgabe besteht darin, den Kommunikationsaufwand zwischen den beteiligten Prozessoren möglichst gering zu halten. Dieser ist nämlich, im Vergleich zur eigentlichen Rechenarbeit, sehr langsam und daher aufwendig.
Um diese Kommunikation einheitlich gestalten zu können wurde der Programmierstandard MPI, das *Message Passing Interface* [10], eingeführt. Dabei handelt es sich lediglich um eine Schnittstelle. Die Implementierung ist auf den meisten Systemen recht unterschiedlich, aber dennoch ein Standard. MPI stellt dabei eine Vielzahl von Methoden und Funktionen zur Verfügung, die zur Kommunikation genutzt werden können. Diese reichen vom einfachen Versand von Daten zwischen zwei Prozessoren über Broadcast-Mechanismen bis hin zur Erstellung von komplexen Kommunikationsnetzwerken in denen Prozessoren zusammen gefasst werden, um beispielsweise ein gegebenes reales geometrisches Verbindungsnetzwerk, wie einen Hypercube, abbilden zu können.
Die effiziente Nutzung dieser Methoden obliegt dabei dem Programmierer. Die verschiedenen Instanzen eines MPI-Programms, welche beim Ablauf auf den eingesetzten Prozessoren laufen, werden auch Prozesse genannt.

[5] Massively Parallel Processor

2.4.2 OpenMP

Neben den massiv parallelen Superrechnern existiert heute noch die SMP-Architektur (*Symmetrisches Multiprozessing*). Symmetrisch bedeutet dabei, dass alle Prozessoren auf einem gemeinsamen Hauptspeicher arbeiten und daher auf die gleichen Daten zugreifen können. Dies bietet den großen Vorteil, die Kommunikation über den gemeinsamen Speicher ablaufen lassen zu können und nicht auf ein, im Vergleich dazu, langsames Verbindungsnetzwerk zurückgreifen zu müssen.
Zur Programmierung auf solchen Systemen steht die Schnittstelle OpenMP (*Open Multi-Processing*)[10] zur Verfügung. Sie stellt verschiedene Direktiven zur Verfügung, die die Arbeitslast des Programms auf mehrere Prozessoren verteilen können, die in diesem Zusammenhang Threads genannt werden. Dabei läuft das Programm zunächst seriell auf einem Thread, bis eine der OpenMP-Direktiven erreicht wird. Daraufhin spaltet sich das Programm in mehrere Threads auf, die dann parallel auf den zur Verfügung stehenden Prozessoren laufen.
Das Haupteinsatzgebiet dieser Schnittstelle stellt die parallele Abarbeitung von `for`/`do`-Schleifen dar. Aber auch andere Konstruktionen sind denkbar, für deren Kontrolle der Programmierer zuständig ist.
Um die Lastverteilung in den parallelen Regionen steuern zu können, stellt OpenMP verschiedene Scheduling-Verfahren zur Verfügung, die im folgenden vorgestellt werden.

- Statisches Scheduling
  ```
  #pragma omp for schedule(static, chunksize)
  ```
  - Statische Aufteilung der Arbeitslast
  - Jeder Thread bekommt Stücke der Grösse `chunksize`
  - Zuweisung im Round-Robin-Verfahren [6]
  - Fehlt chunksize → Aufteilung in $\frac{N}{P}$ Stücke
    (N: Problemgrösse, P: Anzahl der Threads)

[6] Gleichmässige Verteilung der Stücke

- Dynamisches Scheduling
  `#pragma omp for schedule(dynamic,chunksize)`
  - Dynamische Aufteilung der Arbeitslast
  - Jeder Thread bekommt höchstens Stücke der Grösse `chunksize`
  - Zuweisung eines neuen Stücks, wenn der Thread wieder arbeitsbereit ist
  - Fehlt `chunksize` → Aufteilung in Stücke der Grösse 1

- Gesteuertes Scheduling
  `#pragma omp for schedule(guided,chunksize)`
  - Gesteuerte Aufteilung der Arbeitslast
  - Jeder Thread bekommt Stücke, deren Grösse exponentiell abnimmt, bis `chunksize` erreicht ist
  - `chunksize` ist die minimale Grösse der Stücke
  - Zuweisung eines neuen (kleineren) Stücks, wenn der Thread wieder arbeitsbereit ist
  - Fehlt `chunksize` → Minimale Grösse der Stücke ist 1

- Laufzeit-Scheduling
  `#pragma omp for schedule(runtime,chunksize)`
  - Aufteilung der Arbeitslast zur Laufzeit
  - Steuerung über Umgebungsvariable OMP_SCHEDULE
  - export OMP_SCHEDULE "[scheduling],[chunksize](optional)"
    (scheduling: static, dynamic, guided; `chunksize`: Grösse der Stücke)

Zudem können die Gültigkeitsbereiche der zu verarbeitenden Daten festgelegt werden. Da alle Threads auf dem gleichen Hauptspeicher arbeiten, sind die Daten auch für alle sichtbar, was aber durch OpenMP unterbunden werden kann. So können auch in einem parallelen Abschnitt private Variablen existieren, die jeweils nur ein Thread selbst sieht. Innerhalb der Abschnitte sind wiederum Anweisungen möglich, die bewirken, dass nur ein Thread eine bestimmte Anweisung ausführt, oder Bereiche trotzdem seriell abgearbeitet werden. Dies macht aber nur selten Sinn, widerspricht dem Konzept der parallelen Programmierung und hat einen negativen Einfluss auf die Performance des Programms.

OpenMP Programme können meist ohne grosse Veränderungen auf einem, wie auch auf mehreren Prozessoren ablaufen.

2.4.3 Kenngrößen paralleler Programmierung

Um parallele Programme effektiv bewerten zu können ist es notwendig, sie in Relation zu seriellen Programmen zu stellen, die den gleichen Zweck erfüllen. Bei Verwendung von OpenMP kann ein serieller und ein paralleler Programmlauf meist direkt verglichen werden. Unter Verwendung von MPI ergibt sich durch die zusätzliche Kommunikation eine andere Programmstruktur. Daher vergleicht man hier den schnellsten seriellen mit dem zu untersuchenden parallelen Algorithmus.
Von Interesse ist bei beiden Verfahren der, durch die Parallelisierung erzielte, Geschwindigkeitsgewinn. Diesen bezeichnet man als *Speedup* S_p:

$$S_p = \frac{T_s}{T_p} \tag{5}$$

T_s: Ausführungszeit des seriellen Programms
T_p: Ausführungszeit des parallelen Programms mit p Prozessoren
Eine ideale Speedup-Kurve wäre eine Gerade mit $(1, 1)$ als Ursprung und Steigung 1. Da ein paralleles Programm jedoch grundsätzlich auch einen seriellen Anteil besitzt, wird dieses Ideal nie erreicht.
Das „Amdahl'sche Gesetz“ postuliert die Abhängigkeit des Geschwindigkeitszuwachses eines Programms von seinem seriellen Anteil.

$$T_p = \frac{T_s}{p}(1-\alpha) + T_s\alpha$$
$$\Rightarrow S_p = \frac{T_s}{\frac{T_s}{p}(1-\alpha) + T_s\alpha} = \frac{p}{(1-\alpha)+p\alpha} \leq \frac{p}{p\alpha} = \frac{1}{\alpha} \tag{6}$$

T_s: Ausführungszeit des seriellen Programms
T_p: Ausführungszeit des parallelen Programms mit p Prozessoren
p: Anzahl der genutzten Prozessoren
α: Sequentieller Anteil des Programms

Wie gezeigt, ist der Speedup umgekehrt proportional zum sequentiellen Anteil des Programms. Deshalb ist es notwendig, den sequentiellen Anteil so gering wie möglich zu halten, um den Speedup zu erhöhen. Andererseits stellt der Reziprok des sequentiellen Anteils damit eine obere Grenze für den Speedup dar. Deshalb ist auch klar, warum ideale

und gemessene Speedup-Kurve nicht übereinander liegen können.
Eine weitere Kenngrösse zur Bewertung paralleler Programme stellt die Effizienz dar. Sie ergibt sich direkt aus dem Speedup und setzt diesen in Relation zur Anzahl der eingesetzten Prozessoren. Sie bietet einen Überblick darüber, wie gut ein paralleles Programm die ihm zur Verfügung stehenden Ressourcen nutzt.

$$E_p = \frac{S_p}{p} \tag{7}$$

S_p: Speedup unter Nutzung von p Prozessoren

Es gilt $E_p \leq 1$. Mit zunehmender Anzahl eingesetzter Prozessoren nimmt die Effizienz immer weiter ab und nähert sich 0.
Daher ist es nicht sinnvoll, übermässig viele Prozessoren auf ein paralleles Programm anzusetzen. Auf der anderen Seite bedeutet ein näherungsweiser linearer Speedup eine gute Organisation des Datenaustausches und der Synchronisation der beteiligten Prozessoren. Im Allgemeinen bedeutet dies, dass die Rechenlast gleichmässig auf alle teilnehmenden Prozessoren aufgeteilt ist. Dagegen sind extreme oder periodische Veränderungen im Speedup ein Indikator für schlechte Synchronisation der Prozessoren, einen hohen seriellen Anteil des Programms, oder Speicherprobleme, die ebenfalls die parallele Verarbeitung von Daten ausbremsen können.
Es ist daher vor Beginn der Optimierung zuñachst eine Analyse des Speedup notwendig, um die Qualität der Parallelisierung zu charakterisieren.

3 Das Monte-Carlo-Programm ERO

3.1 Programmbeschreibung

ERO wird am Institut für Energieforschung (IEF-4) des Forschungszentrum Jülich eingesetzt und entwickelt [11]. Es ist eine, größtenteils in C/C++ geschriebene Monte-Carlo-Simulation, die ursprünglich am Institut für Plasmaphysik in Garching entwickelt wurde [12].
ERO ist ein 3D-Monte-Carlo-Code [13], der die Plasma-Wand-Wechselwirkung, Transport von Verunreinigungen im Plasma und ihre Lichtemission in einem kleinen Bereich eines Fusionsexperimentes simuliert, der sinnvoller Weise einen Testlimiter umgibt, der untersucht werden soll. Dieser Bereich wird im Verlauf dieser Arbeit aufgrund seiner Struktur als Simulations-Gitter bezeichnen. ERO ist dabei nicht auf die Geometrie eines Tokamak beschränkt, sondern in der Lage beispielsweise auch lineare Plasmaeinschlüsse zu simulieren. Die Simulation arbeitet mit einer Monte-Carlo Annäherung von Testteilchen und verfolgt nur die auftretenden Verunreinigungen. Das Plasma selbst bildet dabei den Simulationshintergrund und wird als Input an das Programm übergeben. Es wird also selbst nicht berechnet.

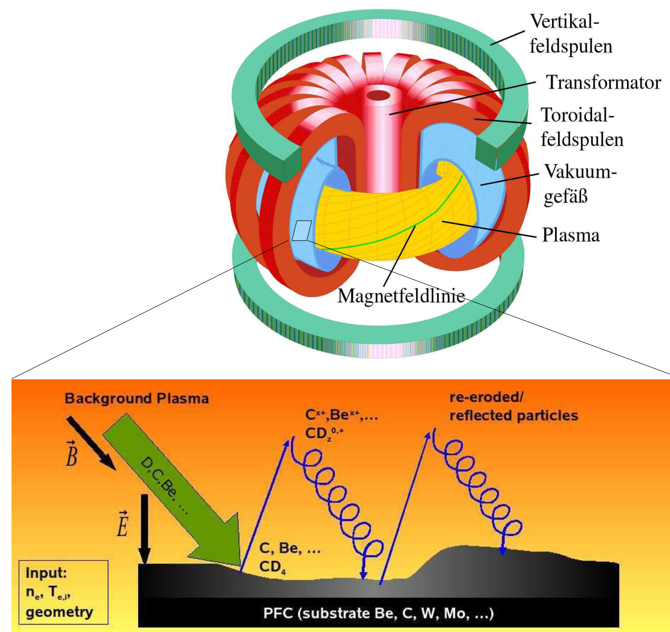


Abbildung 3: Beispielhafte Darstellung der physikalischen Prozesse in ERO.

Die produzierten Daten können mit einem eigens für diesen Zweck entwickelten Matlab-Programm visualisiert und dargestellt werden. ERO wird unter anderem verwendet, um Experimente an TEXTOR nachzuvollziehen, zu validieren und vorauszusagen. Da die Durchführung eines Experiments in TEXTOR einen großen Aufwand darstellt, versucht man mit ERO schon im Vorfeld herauszufinden, welche Daten das Experiment liefern wird und wie es abläuft. Dadurch lassen sich Experimente vermeiden, die zu wenig brauchbaren Ergebnissen führen und es lässt sich gezielt feststellen, welche Ereignisse unter welchen Voraussetzungen zu erwarten sind. Die Simulationsdaten decken sich dabei jedoch nicht immer mit den in Experimenten gewonnenen Daten. Durch diese Rückkopplung lässt sich aber feststellen, welche Verbesserungen an ERO gemacht werden können, wie beispielsweise die Implementierung von weiteren physikalischen Prozessen, um einen möglichst realistischen Simulationsablauf zu erhalten. Schliesslich soll ERO zur vorhersagenden Modellierung von zukünftigen Experimenten wie ITER verwendet werden.

3.1.1 Eingabeparameter

Zur genaueren Trennung und zielgerichteten Simulation ist ERO in sieben Fälle unterteilt worden, die den unterschiedlichen Szenarien bei der Simulation der Wandwechselwirkung entsprechen. Jedes dieser Szenarien kann dabei je nach Bedürfnis wahlweise an- oder ab-

Fall	**Beschreibung**
0	Physikalische Erosion
1	Chemische Erosion
2	Erosion durch Sauerstoff
3	Ablagerung im Plasma
4	Transport von extern eingeführten Teilchen
5	Transport von chemisch erodierten Teilchen
6	Transport von physikalisch erodierten Teilchen

Tabelle 1: Simulationsfälle in ERO

geschaltet werden.
Als Startparameter dient ERO eine Datei, die alle für die Simulation relevanten Daten enthält, nach der ERO simulieren soll. Um nicht auf jeden Parameter einzeln einzugehen, bietet sich hier ein grober Überblick:

- Geometrie
- Plasma
- Anzahl der Zeitschritte und Diskretisierungsinformation
- Zu simulierende Prozesse (Case 0-6, s. Tabelle 1)
- Beobachtete Elemente (Be, C, D, T, ...)
- Ausgabeoptionen

Nach dem Einlesen eines Parametersatzes ist ERO bereit, mit der eigentlichen Simulation zu beginnen. Treten jedoch beim Einlesen unerwartet Fehler auf, oder ist ERO nicht in der Lage, die Eingabeparameter zu verarbeiten, bricht das Programm noch vor der Simulation mit einer genauen Fehlerbeschreibung ab.

3.1.2 Programmablauf

Die Diskretisierung in Einzelzeitschritten wird in ERO über mehrere einzelne Programmaufrufe erreicht. Dabei simuliert jeder Programmaufruf von ERO einen Einzelzeitschritt. Die Ausgabedaten werden im nächsten Zeitschritt wieder eingelesen und als Basis für die Simulation des nächsten Zeitschrittes genutzt. Dieser Vorgang wird über die Batch-Ausführung des IBM LoadLevelers [14] auf JUMP umgesetzt. Hinter dem Ausführungsbefehl von ERO steht dabei in der Batchdatei der Aufruf für die Batchdatei des nächsten Zeitschrittes. Zunächst wird diese Batchdatei als Basis genommen und eine temporäre Batchdatei für den nächsten Zeitschritt erstellt. Der Zustand des Programms, also der momentane Zeitschritt, wird dabei in einer separaten Datei abgespeichert, um später zu wissen, wo der nächste Programmaufruf mit der Simulation fortfahren soll. Ist der Zeitschritt erfolgreich abgeschlossen worden und sind noch zu simulierende Zeitschritte vorhanden, wird die vorher erstellte Batchdatei umbenannt, so dass sie dem Aufruf des LoadLevelers entspricht. Dieser ruft dann als nächsten Befehl eben jene Batchdatei auf. Dadurch entsteht eine Verkettung von Aufrufen, die die gesamte Simulation repräsentiert (Abbildung 4).

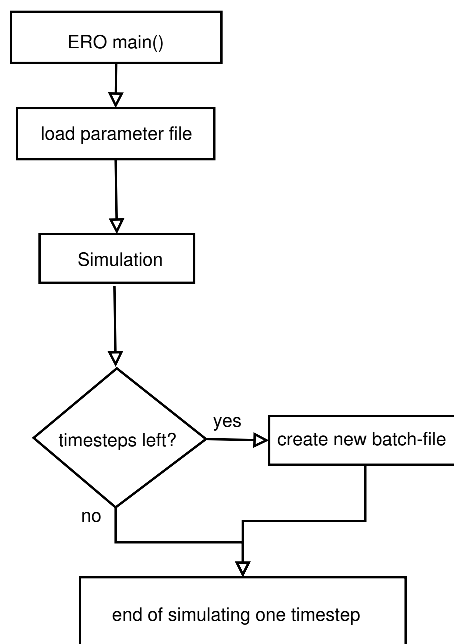


Abbildung 4: Serieller Programmablauf eines LoadLeveler-Jobs

4 Parametrische Parallelisierung

Um einen Einstieg in das Programm zu bekommen und einen ersten Beitrag zur Verbesserung der Simulation mit ERO zu leisten wurde das Programm zunächst parametrisch parallelisiert. Dabei sollen mehrere Instanzen des Programms parallel auf mehreren Prozessoren laufen, die alle unterschiedliche Eingabeparameter haben.

4.1 Motivation

Die Steuerung des Programmablaufs durch eine Eingabeparameterdatei resultiert in einer statischen Simulation für fest eingestellte Eingabeparameter. Möchte man nun ein Szenario unter Variation einiger weniger Eingabeparameter betrachten, ergibt sich die Notwendigkeit mehrere Läufe, mit nur geringfügig geänderten Parametersätzen, durchzuführen. Da der Einsatz des IBM LoadLevelers fester Bestandteil des Programms ist, benötigt man für jedes zu variierende Szenario einen einzelnen Job. Die zeitliche Termination jeder Variation von Parametern ist dabei durch das queue-basierte Vergabesystem von Rechenzeit relativ offen, obwohl sich die zu simulierenden Szenarien nur geringfügig unterscheiden und daher in etwa den gleichen Zeitverbrauch haben. Man muss daher im Extremfall sehr lange auf den Zeitpunkt warten, an dem alle Simulationen terminieren und man diese vergleichen kann. Praktischer wäre hier ein zeitgleiches Ende aller zu untersuchenden Szenarien um diese möglichst zeitnah in Relation stellen zu können. Dies ist vor allem im Hinblick auf die Untersuchung von Fehlern im Programm von Bedeutung. ERO sollte alle Parameterdatensätze korrekt abarbeiten. Bei einem Fehler, der nur bei einer Variation auftritt, sollte das Programm trotzdem alle anderen Simulationen abbrechen, da deren Betrachtung ansonsten keinen Sinn mehr macht.

4.2 Startprogramm zur Parametervariation in ERO : RunERO

Um die verschiedenen Simulationsszenarien nicht per Hand über manuelle Eingabe der zu variierenden Parameter in der Parameterdatei zu erzeugen, gibt es für diesen Verwendungszweck ein Startprogramm, mit dem sich dies automatisch bewerkstelligen lässt. Dazu werden die Parameter, die später eingesetzt werden sollen, als Array in einer speziellen Eingabedatei im Matlab-Format abgespeichert, welche vom Benutzer erstellt werden muss. Weiterhin benötigt man eine Parameterdatei, die später als Basis für alle Szenarien dienen soll, sowie eine LoadLeveler-Batchdatei, die ebenfalls als Basis für die Läufe dient.
'RunERO' liest dabei die Parameterkombinationen aus genannter Datei aus, erstellt auf dieser Grundlage Unterverzeichnisse für jeden Lauf und speichert darin die durch Ersetzung der ausgelesenen Parameter erzeugte ERO-Parameterdatei, sowie die Batch-Datei

ab. Bei letzter wird zusätzlich das Arbeitsverzeichnis modifiziert, welches nun auf den Ordner des jeweiligen Laufs zeigt. Nun übergibt 'RunERO' jede der Batch-Dateien an den LoadLeveler, der diese in die Queue einreiht. Wie bereits eingangs erwähnt ist dabei die zeitliche Synchronisierung der einzelnen Szenarien nicht gegeben.

4.3 Parallelisierung

Um die zeitgleiche Terminierung aller Szenarien sicher zu stellen, bot sich die Aufspaltung von ERO in mehrere unabhängige Prozesse an, die dennoch parallel ablaufen, aber auf verschiedenen Parametersätzen simulieren. Hierbei ist die Parallelisierung mit MPI sinnnvoll, da dieses implizit für die Trennung der Adressbereiche der Prozessoren sorgt und ERO bereits Regionen enthält, die mit OpenMP parallelisiert worden sind (s. Kap. 2.4.2). Um dieses Ziel zu erreichen, wurde die Verarbeitung der Parameterliste des Programms verändert. Zuvor akzeptierte ERO lediglich einen Dateinamen zu einer einzelnen Parameterdatei.

Nun lässt sich der Einsatz der parametrischen Parallelisierung durch ein Flag steuern, dem eine Liste von Parameterdateien anhängt, welche die verschiedenen Prozesse von ERO verarbeiten sollen. Ist dieses Flag gesetzt, spaltet sich ERO kurz nach dem Start in die Anzahl von Prozessen auf, von der auch Parameterdateien im Aufruf vorhanden sind. Jeder Prozess arbeitet jetzt autark an seiner Simulation, lediglich die zeitliche Diskretisierung über die Batch-Dateien des LoadLevelers wird zentral von einem Master-Prozess gesteuert.

Zur Vereinfachung der Benutzung dieser parametrischen Parallelisierung wurde RunERO ebenfalls modifiziert. Die Vorgehensweise von RunERO bleibt dabei bis auf die Verarbeitung der LoadLeveler-Batch-Datei gleich. Ordner müssen angelegt und die notwendigen Parametersätze modifiziert und dort hinein kopiert werden. Der Unterschied ergibt sich jedoch bei der Behandlung der Batch-Datei.

Anstelle von n Batch-Dateien bei n Parametersätzen wird beim Aufruf der parametrisch parallelisierten Version von ERO nun lediglich eine benötigt (Abbildung 5).

Die Batch-Datei liegt dabei im Wurzelverzeichnis, welches die Unterverzeichnisse zu den verschiedenen Szenarien enthält. In dieser wird der Aufruf von ERO derart modifiziert, so dass die parametrisch parallelisierte Version gestartet und die Parametersätze mit ihrem relativen Pfad als Aufrufparameter angehängt werden.

Zudem wird anhand der Anzahl der Parametervariationen die benötigte Prozessoranzahl ermittelt und ebenfalls in die Batchdatei geschrieben. Zu guter Letzt liefert RunERO diese Batchdatei an den LoadLeveler aus, der diese bei Verfügbarkeit von genügend Ressourcen startet.

4.4 Vorteile der parametrischen Parallelisierung

Die Nutzung von MPI bei der parametrischen Parallelisierung bietet einige Vorteile bei der Nutzung des Programms. Es können nun, lässt man die ebenfalls vorhandene Parallelisierung mit OpenMP außer Acht, ebenso viele Szenarienvariationen untersucht werden, wie Prozessoren auf dem genutzten MMP-System zur Verfügung stehen. Die Adressbereiche der einzelnen Prozesse sind dabei implizit getrennt, so dass keine Abhängigkeiten zwischen den Prozessen bestehen und es nicht zu Problemen beim Speichermanagement kommt. Jedem Prozessor steht sein eigener Speicherbereich zur Verfügung, der von den anderen getrennt ist. Für dieses Verfahren ist zudem keine Kommunikation zwischen den Prozessen notwendig, wodurch die Parallelisierung sehr einfach gehalten werden kann. Da der Programmablauf erst endet, wenn alle Prozesse mit ihrer Simulation fertig sind, ist auch für die notwendige Synchronisation gesorgt. Es ist sicher gestellt, dass bei Simulation eines neuen Zeitschrittes, der vorige Zeitschritt von allen Prozessen zu Ende simuliert worden ist. Dadurch stehen nach Ende der Simulation Daten von allen Szenarienvariationen zur Verfügung, die nun zeitnah verglichen und ausgewertet werden können. Durch die Kopplung der Prozessoren über MPI ist ausserdem sichergestellt, dass bei einem Programmfehler auch die Simulationen abbrechen, die nicht davon betroffen sind und damit der Verbrauch von unnötiger Rechenzeit unterbunden wird. Natürlich lässt sich bei dieser Vorgehensweise keine Aussage über den Speedup machen, da die Parallelisierung auch nicht zum Ziel hatte, eine höhere Ausführungsgeschwindigkeit des Programms zu erreichen. Unter Beachtung der vorhandenen Parallelisierung mit OpenMP ist man jedoch nun in der Lage, beide Verfahren zu kombinieren. Dabei können einerseits mehrere Szenarien gleichzeitig, und diese andererseits mit einer Performancesteigerung unter Benutzung mehrerer OpenMP-Threads, simuliert werden. Eine solche Struktur bietet auch im Hinblick auf zukünftige MMP-Architekturen einige Vorteile, solange einigen wenigen Prozessoren noch ein gemeinsamer Speicher zur Verfügung steht.

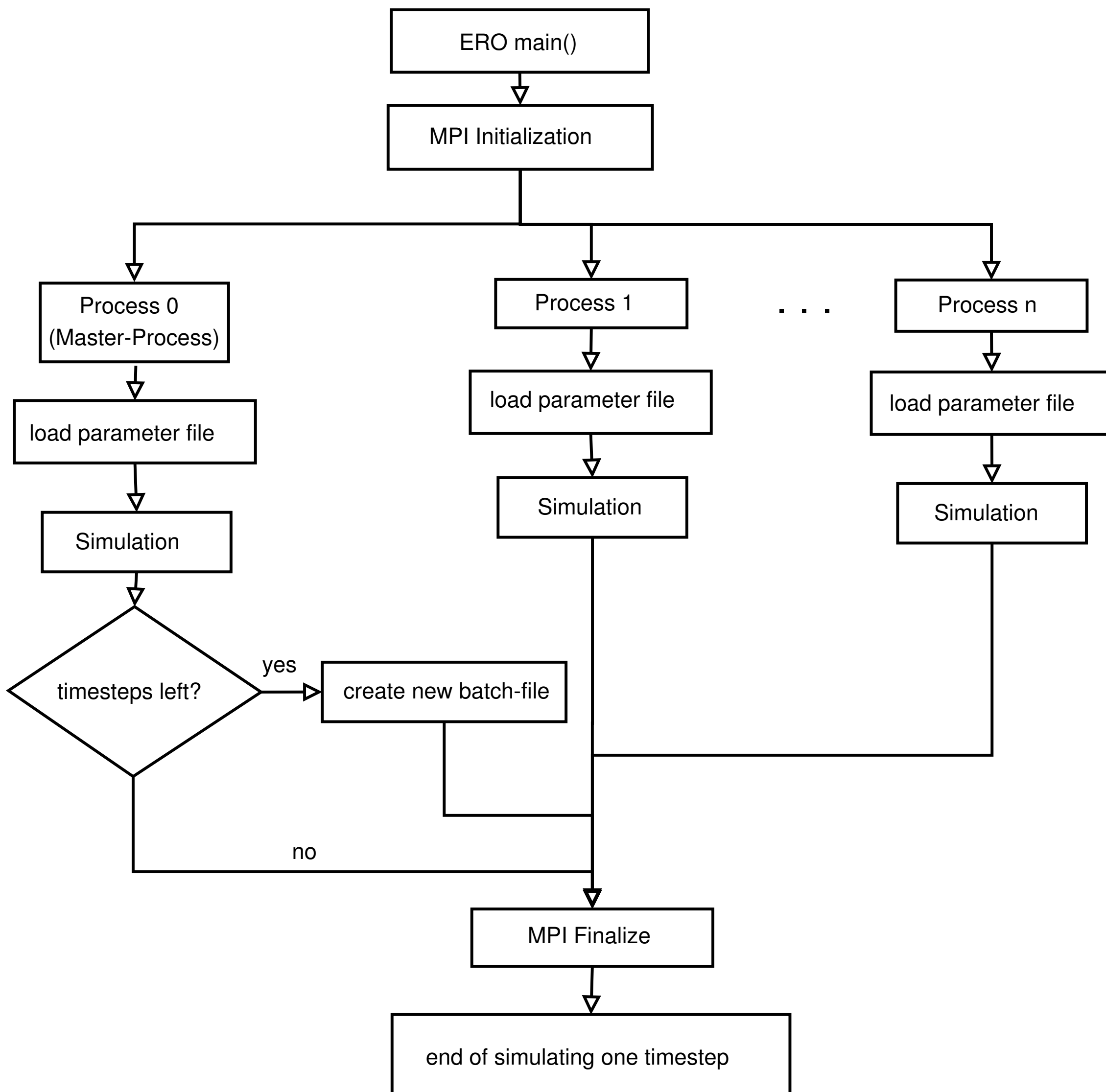


Abbildung 5: Parametrisch parallelisierter Programmablauf von ERO. Prozess i simuliert Szenario i

5 Analyse und Optimierung der Parallelisierung mit OpenMP

Der Fokus dieser Arbeit, nämlich die Verbesserung des Speedup der Parallelisierung mit OpenMP, wird in diesem Kapitel betrachtet. Dazu wurde zunächst der zu bearbeitende Teil des Programms eingehend mit Profilern untersucht, um Schwachstellen im Code zu finden, die negativen Einfluß auf die Performance haben. Nach der Lokalisierung dieser Schwachstellen wurden Strategien entwickelt, diese zu beheben, um die Parallelisierung zu verbessern.
Um die verschiedenen Messungen sinnvoll vergleichen zu können, war es notwendig, für alle Läufe die gleichen Bedingungen zu schaffen, was bei einer Monte-Carlo Simulation in der Regel, verursacht durch den Pseudo-Zufallsgenerator, nicht der Fall ist. Die Simulationen wurden daher mit einer konstant bleibenden Anzahl von Testteilchen und einem konstanten Random-Seed, der die Basis der MC-Simulation bildet, durchgeführt. Weist der Random-Seed nämlich einen konstanten Wert ungleich Null auf, wird der Aufruf des Pseudo-Zufallsgenerators übersprungen und dieser Wert als Basis für den Lauf genommen. Dadurch ergibt sich ein statischer Simulationsablauf, der eine gute Vergleichbarkeit des Zeitaufwands ermöglicht. Das Verhalten wurde über die in ERO eingebaute Zeitmessung untersucht, welche für einen groben Überblick völlig ausreichend ist. Die Threadanzahl wurde in einem Spektrum von eins bis zweiundreißig variiert, weil diese Höchstgrenze durch die Architektur von JUMP vordefiniert ist, die für jeden SMP Knoten 32 Prozessoren zur Verfügung stellt (vgl. Kap. 2.2).

Im Folgenden werden die wichtigsten Simulationsparameter aufgeführt.

- Testteilchen: 1000
- Random Seed: 1207644223
- Zeitschritt: 1
- Spektroskopienetz / [mm]
  - poloidal boundary min/max: -195/195
  - toroidal boundary min/max: -195/195
  - radial boundary min/max: 0/250
  - poloidal cell size: 12
  - toroidal cell size: 12
  - radial cell size: 12
- Finemeshgeometry / [mm]
  - poloidal boundary min/max: -60/60
  - toroidal boundary min/max: -60/60
  - radial boundary min/max: 0/200
  - poloidal cell size: 4
  - toroidal cell size: 2
  - radial cell size: 2
- Traced elements
  - Beryllium
  - Carbon
  - re-deposited Carbon [7]

[7] Kohlenstoff und re-deponierter, auf der Oberfläche wieder abgelagerter, Kohlenstoff werden in ERO separat betrachtet, um den Effekt erhöhter Re-Erosion, von re-deponierten Kohlenstoff, berücksichtigen zu können.

5.1 Laufzeitverhalten beim Transport von extern eingeführten Teilchen (Case 4, vgl. Tabelle 1)

Die Unterteilung von ERO in mehrere zu untersuchende Fälle ermöglicht eine gezielte Beobachtung des Laufzeitverhaltens. Da die den vier Fällen (Case 0-3, vgl. Tabelle 1) der Simulation keine Parallelisierung benötigten, lag der Schwerpunkt der Untersuchung nun bei den Fällen, bei denen eine Parallelisierung aufgrund des Zeitaufwandes der simulierten Prozesse sinnvoll ist. Der Transport von extern eingeführten Teilchen (Case 4) stellt ein solches Szenario dar.
Im besagten Szenario befinden sich Monte-Carlo Testteilchen in der Situation, von einer bestimmten Zelle der Oberfläche der Tokamak-Wand, mit einer Maxwell-verteilten Geschwindigkeit und einem azimutal/cosinus-verteilten Anfangswinkel aus zu starten und sich darauf hin durch das Plasma zu bewegen, bis einer der folgenden 3 Fälle auftritt:

- Das Teilchen wird auf der Wand (re-)deponiert
- Das Teilchen verlässt das Simulations-Gitter
- Das Teilchen hat einen „timeout“ [8]

Auf ihrem Weg wird die Geschwindigkeit der Teilchen und deren Bewegungsrichtung durch die B- und E-Felder, Stösse mit anderen Teilchen sowie durch Ionisation, Dissoziation und Rekombination beeinflusst.
Um einen groben Überblick der Performance von ERO im Hinblick auf den zu untersuchenden Fall zu erhalten, wurde zunächst eine Bestimmung des parallelen Verhaltens von ERO durchgeführt, in der das *„Strong Scaling“* des Programms nur mit diesem eingeschalteten Szenario betrachtet wird. Insbesondere der Speedup (Ausdruck 5) und die Effizienz (Ausdruck 7) sind hier von Interesse.

[8] Es kann vorkommen, dass es sehr lange dauert, bis eins der vorher genannten Ereignisse eintritt, oder dass diese Fälle aufgrund bestimmter Situationen gar nicht eintreten. Um eine Endlossschleife zu vermeiden ist deshalb jedes Teilchen mit einer „timeout“ -Zeit ausgestattet, nach der die Verfolgung automatisch abgebrochen wird.

Aus den gemessenen Laufzeiten wurde eine Speedup-Kurve erstellt, um das Verhalten möglichst gut abbilden und visualisieren zu können (Abbildung 6). Es hat den Anschein,

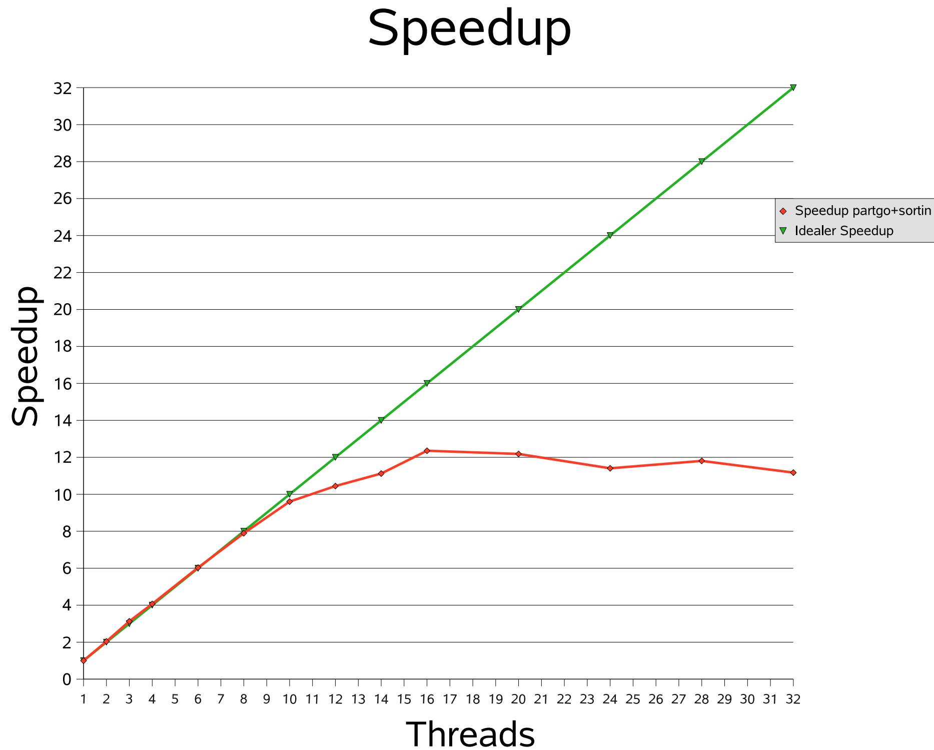


Abbildung 6: Laufzeitverhalten der untersuchten parallelen Region in Case 4

das auf jeden Thread noch ein überhöhter Anteil an serieller Arbeit zukommt, der einen negativen Einflußauf die Balance des parallelisierten Bereichs hat.
Zur genaueren Analyse kamen nun der Profiler KOJAK (Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks), sowie der darauf aufbauende SCALASCA Instrumentierer zum Einsatz. Beide Programme sind am Jülicher Supercomputing Centre entstanden und werden dort sowohl erprobt als auch eingesetzt, weshalb auf einen reichhaltigen Erfahrungsschatz zurückgegriffen werden kann.

Listing 1: Beispiel zur SCALASCA Instrumentierung

```
int main (int argc , char *argv[])
{
#pragma pomp inst init // Initialisierung von SCALASCA
Anweisung 1
#pragma pomp inst begin(region1) // Intrumentierte Region 1
Anweisung 2
Anweisung 3
#pragma pomp inst end(region1)
Anweisung 4
Anweisung 5
#pragma pomp inst begin(region2) //Instrumentierte Region 2
Anweisung 6
#pragma pomp inst end(region2)
}
```

Um den Output von SCALASCA gering zu halten und damit die Analyse zu vereinfachen wurden nur die Regionen instrumentiert, die für die Performance des untersuchten Falls Relevanz haben (Listing 1). Dies geschieht in SCALASCA durch die Definition von Abschnitten innerhalb des Codes, die untersucht werden sollen. So wird einerseits dafür Sorge getragen, den Analyseaufwand möglichst gering zu halten und anderseits resultiert dies in einer guten Übersichtlichkeit der Ausgabedaten, die sich dann nur auf die gewünschten Regionen beziehen. Die Meßdaten können mit CUBE (CUBE Uniform Behavioral Encoding [15]) graphisch dargestellt werden. Mit dieser regionsbezogenen Instrumentierung wurde nun der Bereich immer weiter eingeschränkt, der für einen negativen Einfluß auf die Performance verantwortlich sein könnte und schließlich eine Methode identifiziert, die den Großteil an Rechenzeit verbraucht (Abbildung 7).

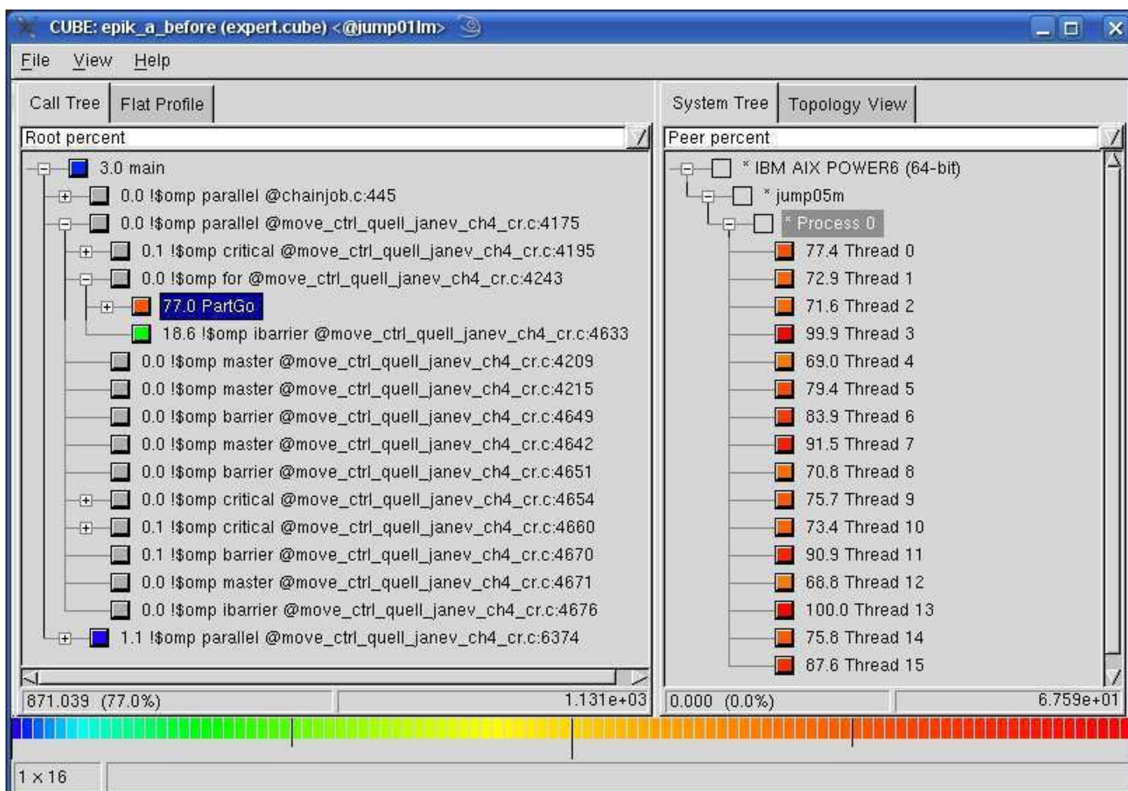


Abbildung 7: CUBE Visualisierung der SCALASCA Analyse von `PartGo` für eine Simulation mit 10000 Teilchen auf 16 Threads

5.2 Analyse und Optimierung

5.2.1 Methode `PartGo`

Die Methode `PartGo` befindet sich in einer parallelisierten `for`-Schleife, die die Arbeit an den Testteilchen auf die verfügbaren Threads verteilt. Das bedeutet für jeden Thread, dass er diese Methode für jedes ihm zugewiesene Teilchen einmal aufrufen muss und zugleich, dass jene nicht selbst parallelisiert ist.
In Abbildung 7 ist die SCALASCA-Analyse mit CUBE visualisiert dargestellt. In der linken Seite des Fensters sind die einzelnen parallelen Regionen, sowie die manuell instrumentierte Regionen des Programms (vgl. Listing 1), dargestellt. Diese Darstellung repräsentiert die Aufrufhirarchie in ERO. Der Bezeichner der Region setzt sich aus folgenden Daten zusammen: Links steht der prozentuale Anteil des Zeitverbrauches dieser Region bezogen auf die nächst höhere Region in der Hirarchie. Danach folgt entweder der Name, welcher bei der manuellen Instrumentation gewählt wurde, oder die Quelldatei, einschließlich der Zeile, in der diese Region beginnt. Auf der rechten Seite des Fensters ist die prozentuale Auslastung der beteiligten Threads dargestellt, bezogen auf die in der linken Seite des Fensters ausgewählte Region.
Wie man erkennen kann, ist es sinnvoll sich mit dieser Methode zu beschäftigen, da sie den Großteil an Zeit in der Simulation konsumiert. Von SCALASCA ist am Ende der parallelen `for`-Schleife auch ein wesentlicher Zeitverbrauch an einer impliziten Barriere festgestellt worden, was auf ein Synchronisationsproblem der beteiligten Threads hindeutet. Dies ist deutlich im rechten Fensterabschnitt von Abbildung 7 zu sehen, wo die Auslastung der Threads angezeigt wird, welche sehr ungleichmässig ist. Daher lohnt es sich, einen näheren Blick auf die Arbeitsweise der Methode zu werfen, um die Probleme zu identifizieren.
`PartGo` hat die Aufgabe, den Transport der Testteilchen zu berechnen. Die Testteilchen lassen sich dabei in zwei Gruppen zusammenfassen: Ionen und Neutrale. Neutrale sind dabei Teilchen ohne elektrische Ladung, die einen relativ langen Weg zurücklegen können, da sie nicht durch das elektromagnetische Feld beeinflußt werden. Ihre Verfolgung ist dabei besonders aufwändig und sie stellen den Großteil an beobachteten Teilchen in diesem Szenario dar. Ionen sind Teilchen mit einer elektrischen Ladung und unterliegen daher dem Einfluß des elektromagnetischen Feldes und von Coulomb-Stößen mit den Plasmaionen.
Das große Problem bei der Verfolgung dieser Teilchen ist der Umstand, dass aufgrund der hohen Komplexität der Simulation keine Aussage darüber getroffen werden kann, wann ein Ereignis bei diesen Teilchen eintritt. Die Trajektorien der Teilchen sind höchst unterschiedlich, weshalb auch keine quantitative Aussage über den jeweiligen Zeitverbrauch pro Teilchen getroffen werden kann. Die Teilchen bewegen sich schließlich so lange durch das

Plasma, bis sie eine Wand treffen oder sie ionisiert und als Ionen weiter verfolgt werden (Abbildung 8).

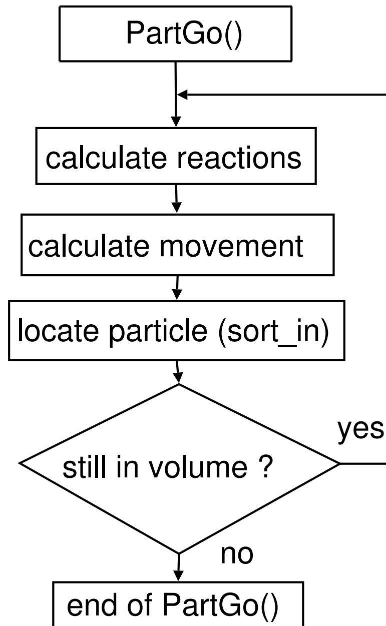


Abbildung 8: Ursprünglicher Programmablauf von `PartGo`

Damit kann nicht gewährleistet werden, bei der Parallelisierung eine gleichmäßige Lastverteilung über alle beteiligten Prozessoren zu erhalten.
Ein zusätzlicher Faktor mit negativem Einfluß auf die Performance dieses Vorgangs ist der Vorgang des Einsortierens der Teilchen in das Simulations-Gitter. Das Simulations-Gitter wird in ERO durch ein oder mehrere dreidimensionale Felder beschrieben. Dabei kann in Bereichen von besonderem Interesse eine feinmaschigere Auflösung gewählt werden.
Bewegt sich nun ein Teilchen durch dieses Gitter, wird esin jede Zelle einsortiert, die es durchläuft. Dies geschieht bei jedem Durchlauf der Schleife (vgl. Abbildung 8), was sehr zeitaufwändig ist, da die genau Anzahl der Durchläufe von vorn herein nicht festgelegt ist. Diese Aufgabe wird von der Methode `sort_in` bewältigt, die jeweils ein Teilchen pro Durchlauf lokalisiert und in die Zelle, in der sich das Teilchen gerade befindet, einsortiert. Wie in Abbildung 8 ersichtlich ist, wird sort_in in jedem Schleifendurchlauf aufgerufen. Dies ist sehr aufwendig, da `sort_in` direkt auf dem Simulations-Gitter arbeitet, welches erheblichen Platz im Speicher verbraucht.

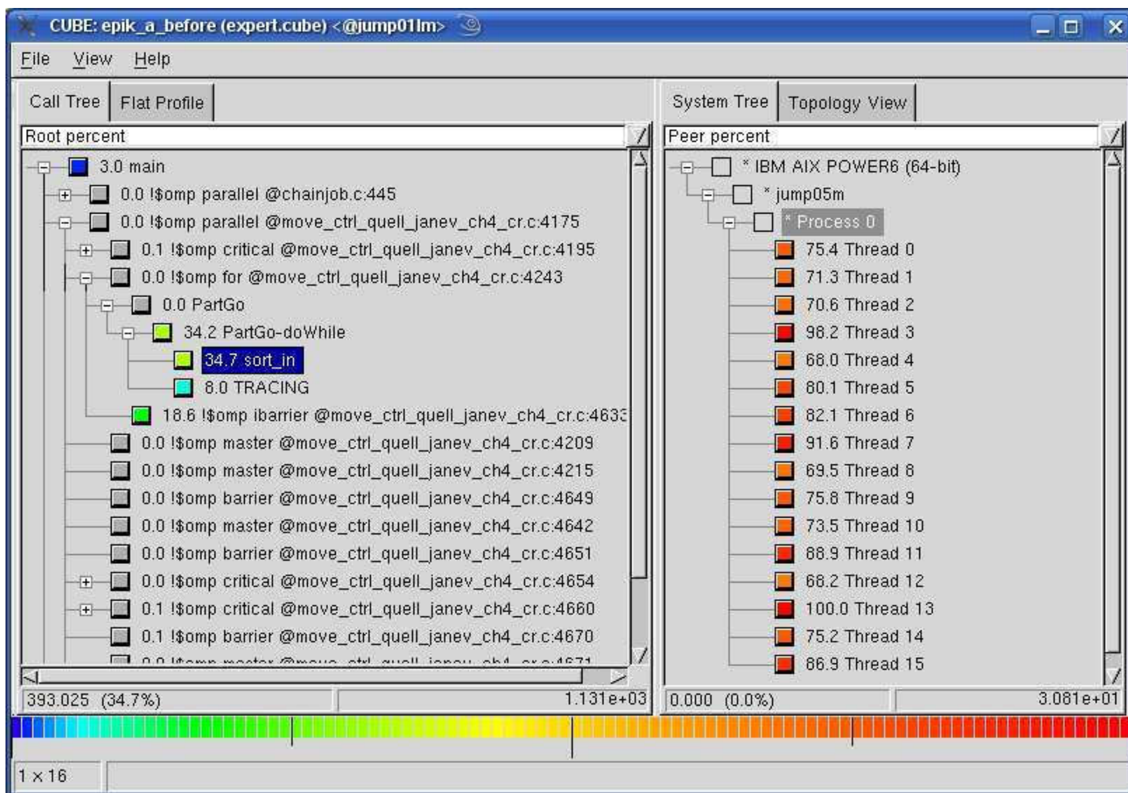


Abbildung 9: CUBE Visualisierung der SCALASCA Analyse von `sort_in` für eine Simulation mit 10000 Teilchen auf 16 Threads

Um diesen Nachteil zu kompensieren wurde ein Feld eingeführt, in der Teilchentrajektorien gespeichert werden können. Anstatt nun das zu simulierende Teilchen in jedem Schleifendurchlauf in das Gitter einzusortieren, fügt man es zuächst dem Feld hinzu. Hier aggregieren sich die Daten jedes Teilchens zu einer entsprechenden Trajektorie. Diese Trajektorie endet in folgenden Fällen:

- Ladungsänderung des Teilchens
- Feld(Collection) ist voll
- Teilchen hat das Simulations-Gitter verlassen

Tritt eines dieser Ereignisse ein, muss das komplette Feld einsortiert werden (Abbildung 10).

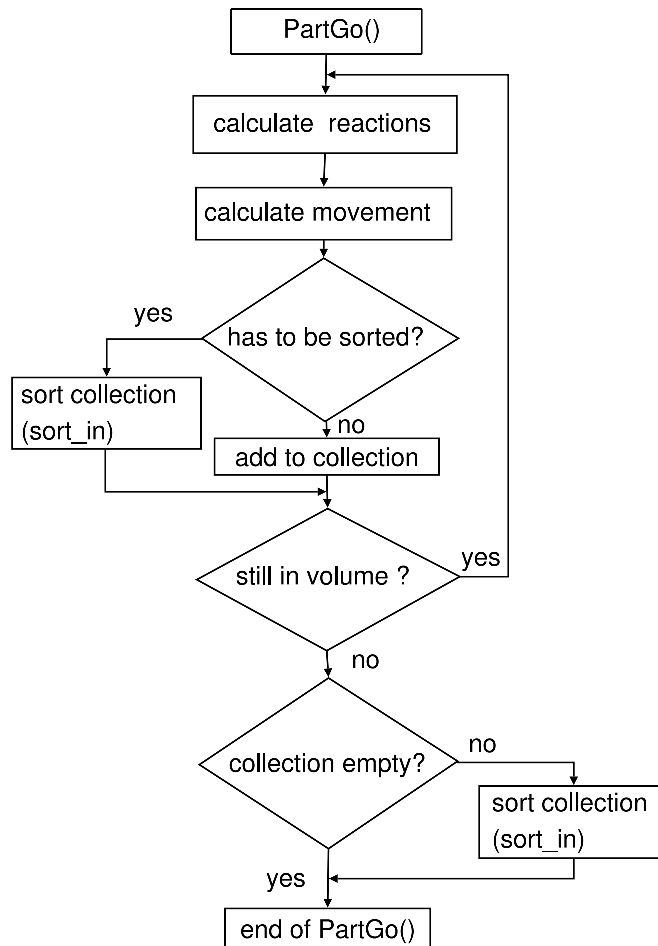


Abbildung 10: Programmablauf der modifizierten Version von `PartGo` mit Trajektoriensammlung in jedem Schritt.

5.2.2 Methode `sort_in`

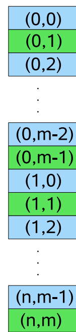
Die Aufgabe des Einsortierens der Teilchen in das Gitter wird von der Methode `sort_in` bewältigt. `sort_in` bekommt dabei lediglich den Zeiger auf das aktuelle Teilchen und den Zeiger auf das Simulations-Gitter geliefert. Es muss festgestellt werden, ob sich das Teilchen überhaupt im Gitter befindet, um dann seinen Ort bestimmen zu können. Da die Methode sehr kurz ist, hat man sich hierbei für die einfachste Möglichkeit entschieden, den Ort des Teilchens gegen den gültigen Bereich durch `if`-Abfragen zu prüfen, dann die Position im Gitter zu bestimmen und es anschließend einzusortieren (Listing 2). Wie bereits erwähnt wird `sort_in` jedoch sehr häufig aufgerufen, was zu einem negativen Einfluß auf die Gesamtperformance führt. Der Aufruf von Kontrollstrukturen bringt immer einen gewissen Aufwand mit sich, der durch resultierenden Cache-Misses noch verstärkt wird, denn falls der Prozessor im Cache ein benötigtes Datum, in diesem Fall eine benötigte Zelle, nicht findet, muss er es zeitaufwändig aus dem Hauptspeicher laden.

Listing 2: Pseudocode von `sort_in`

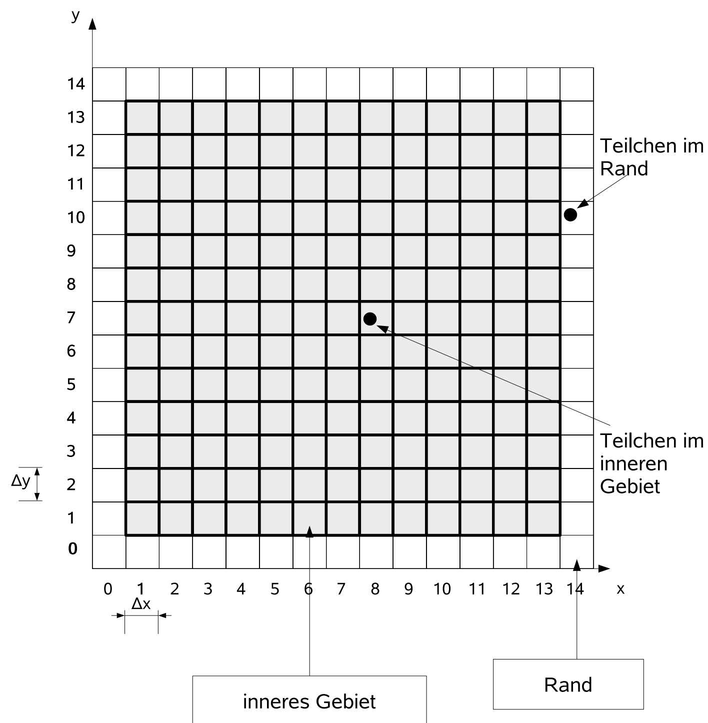
```
Prozedur: sort_in
Zweck: Lokalisierung eines Teilchens im Gitter
Parameter: Teilchen T, Gitter G

Begin der Prozedur

Falls 0 <= T.X < G.MaxX
  Berechne x

  Falls 0 <= T.Y < G.MaxY
    Berechne y

    Falls 0 <= T.Z < G.MaxZ
      Berechne z

      Sortiere T in G(x,y,z) ein

Ende der Prozedur
```

Um diese negativen Effekte zu eliminieren, musste die Methode umstrukturiert werden. Ziel war es, die Verzweigungen aus dem Code zu entfernen und eine bessere Cache-Auslastung zu erreichen. Zu diesem Zweck wurde das gesamte Gitter mit einem Rand ausgestattet, der den Bereich, außerhalb des Simulations-Gitters repräsentieren soll. Auf diesen Rand sollen alle Teilchen abgebildet werden, die sich nicht mehr innerhalb des Gitters befinden. In der programmtechnischen Realisierung ist das Gitter zunächst ein 2D-Pointer-Feld, welches nachteilig alloziert wurde, da die Speicherbereiche, die jenes Feld aufspannen, nicht unmittelbar hintereinander liegen. Die Elemente dieses Feldes enthalten dann jeweils noch ein 1D-Feld, welches die z-Koordinate des Zelle repräsentiert.
Zur Optimierung wurde nun ein Hilfsfeld eingeführt, welches in jeder Dimension um 2 Elemente größer ist als das Ursprungsfeld. Das Hilfsfeld, wie auch das Ursprungsfeld, wurde so angelegt, dass ein zusammenhängender Speicherbereich existiert (Abbildung 11.1). Dabei sind die Elemente im inneren des Hilfsfeldes diejenigen, die zum Ursprungsfeld gehören (Abbildung 11.2).

```
v_net =  calloc(m,sizeof(...));
v_net[0] =  calloc(m*n,sizeof(...));
for(i=1; i<m; ++i)
    v_net[i] = v_net[i-1] + n;
```



11.1: Optimierter Speicherbereich



11.2: Optimiertes Gitter mit Randgebiet

Abbildung 11: Schematische Darstellung der Konstruktion der Speicherbereiche

Um die Teilchen nun im Hilfsfeld einordnen zu können ist eine Funktion notwendig, welche die Indizes des Ursprungsfeldes in Indizes des Hilfsfeldes übersetzt. Im folgenden bezeichne ich mit U das Ursprungsfeld und mit H das Hilfsfeld. Sei m die Anzahl der Zeilen und n die Anzahl der Spalten von U. Das Feld U hat somit die Dimension $m \times n$. Das Hilfsfeld hat die Größe des Ursprungsfeldes und zusätzlich einen Rand von einem Element Breite an jeder seiner Seiten. Damit ergibt sich für das Hilfsfeld die Dimension $(m+2) \times (n+2)$. Es müssen alle Teilchen, die sich außerhalb des Ursprungsfeldes befinden auf den äusseren Rand des Hilfsfeldes abgebildet werden. Alle Teilchen innerhalb des Ursprungsfeldes werden auf ihre entsprechende Stelle im Hilfsfeld abgebildet. Zwischen den Koordinaten in U und H, die nicht zum Rand von H gehören, besteht folgende Beziehung:

Für $j \in \mathbb{N} \cap [0, m-1], i \in \mathbb{N} \cap [0, n-1]$

$$U(j, i) = H(j+1, i+1) \tag{8}$$

Die Umrechnung der Indizes für eine Koordinate wird in (9) ersichtlich.

$$i = max\{min\{(i^{'} + 1), max_i\}, 0\} \tag{9}$$

$i^{'}$: Index, relativ zum Ursprungsfeld U
i: Berechneter Index im Hilfsfeld H
max_i: Maximaler Index des Hilfsfeldes

Zur Veranschaulichung werden unten repräsentative Beispiele gewählt(Tabelle 2), die zeigen, dass die angegebene Funktion ihre Gültigkeit hat. Vereinfachend sei $max_i = 14$.

$i^{'}$	$min\{(i^{'} + 1), max_i\}$	$max\{min\{(i^{'} + 1), max_i\}, 0\}$
-15	-14	0
-5	-4	0
0	1	1
7	8	8
12	13	13
13	14	14
17	14	14

Tabelle 2: Beispielwerte der Indextransformation

Durch diese Vorgehensweise werden die Teilchen in genau der richtigen Zelle des Ursprungsfeldes eingeordnet, ohne zu überprüfen ob das Teilchen sich im Gitter befindet. Jene, die sich außerhalb des Ursprungsfeldes befinden, werden alle auf den Rand des Hilfsfeldes abgebildet. Das Hilfsfeld wird nur für diese Einordnung genutzt und im weiteren Verlauf des Programms nicht mehr benötigt. ERO wertet später ausschließlich das Ursprungsfeld aus, in dem sich die korrekt lokalisierten Teilchen befinden.
In Zukunft könnte das Hilfsfeld aber von physikalischer Seite aus interessant werden, da es dazu genutzt werden kann, den Ort zu bestimmen, an dem das Testteilchen das Simulations-Gitter verlassen hat.

5.2.3 Lastverteilung des OpenMP Schedulers

Da nun die wesentlichen Methoden optimiert worden sind, die im parallelen Bereich des untersuchten Falls liegen, macht es Sinn, sich den parallelen Bereich und dessen Konfiguration selbst anzuschauen. Dabei fällt auf, dass das Schleifenscheduling nicht optimal implementiert worden ist. Während für das Verfahren selbst eine dynamische Lastaufteilung gewählt wurde, ist die `chunksize`, also die Grösse der Stücke, welche verteilt werden, auf $\frac{N}{P}$ gesetzt worden, wobei N die Problemgrösse, also in diesem Fall die Anzahl der zu simulierenden Teilchen, und P die Anzahl der eingesetzten Prozessoren ist.
Diese Konfiguration führt weg von einer dynamischen und hin zu einer statischen Lastaufteilung, da jeder Thread zu Anfang schon ein Stück der maximalen Grösse erhält. Sinniger ist es, die Stücke kleiner aufzuteilen, um eine dynamische Aufteilung während der Bearbeitung der Schleife überhaupt zu ermöglichen. In der Implementierung wird

$$\texttt{chunksize}_{dynamic} = \frac{N}{10 \cdot P} \tag{10}$$

N: Problemgrösse
P Anzahl eingesetzter Threads
$\texttt{chunksize}_{dynamic}$: Grösse der Stücke bei dynamischer Lastverteilung

gesetzt und die Teilchenzahl, zur Messung des Einflusses der `chunksize`, auf 30000 erhöht um Simulationsbedingungen zu schaffen, die bei den meisten Szenarien angewendet werden.

5.3 Resultate der Optimierung

Zunächst werden die Ergebnisse der Optimierung von `PartGo` betrachtet, da diese aus Sicht des Programmablaufs eine Stufe über dem Aufruf von `sort_in` rangiert. Danach wird auf die Verbesserungen der Methode `sort_in` eingegangen, und deren Einfluß auf die endgültige Performancesteigerung des Case 4.
Um die beiden Optimierungen voneinander abgrenzen zu können, wurde der Zeitverbrauch des Programms erst nur mit jeweils einer aktiven Optimierung gemessen und anschließend in der Kombination. Dadurch wird besser ersichtlich welches Optimierung welchen Einfluß auf die gemessene Speedup-Kurve hat. Es wird weiterhin ein Überblick über beide Optimierungen gegeben und in Relation zu der Situation vor der Optimierung gestellt. Schließlich wird der Einfluss der `chunksize` auf die Auslastung untersucht, sowie der Zeitverbrauch vor und nach der Optimierung verglichen.

5.3.1 `PartGo`

Die Aggregation der Teilchentrajektorien und deren anschließende Einsortierung sorgen für ein späteres Abflachen der Speedup-Kurve (Abbildung 12). Es lässt sich eine Steigerung des Speedup bis zu einer Anzahl von 24 Threads beobachten, auch wenn die Effizienz in diesem Bereich suboptimal ist. Lässt man den Lauf mit 32 Prozessoren außer Acht, ergibt sich bei der Programmversion vor der Optimierung ein maximaler Speedup von etwas über 10 (Abbildung 6). Mit den vorgenommenen Verbesserungen in der Methode `PartGo` lässt sich nun jedoch ein maximaler Speedup von ca. 16 erreichen (Abbildung 12). Dies bedeutet für sich allein schon einen Gewinn von über 50 %. Es sind nun zwar noch die gleiche Anzahl an Zugriffen auf das Gitternetz zu verzeichnen, diese konzentrieren sich aber aufgrund des Weges, den ein Teilchen zurücklegt, nun in bestimmten Bereichen und es erhöht sich dadurch die Wahrscheinlichkeit eine bestimmte Zelle bereits im Cache vorzufinden, was einen schnelleren Zugriff bedeutet. Dies schlägt sich offensichtlich in der Speedup-Kurve nieder und ebenfalls in der Auslastung der beteiligten Threads (Abbildung 13).

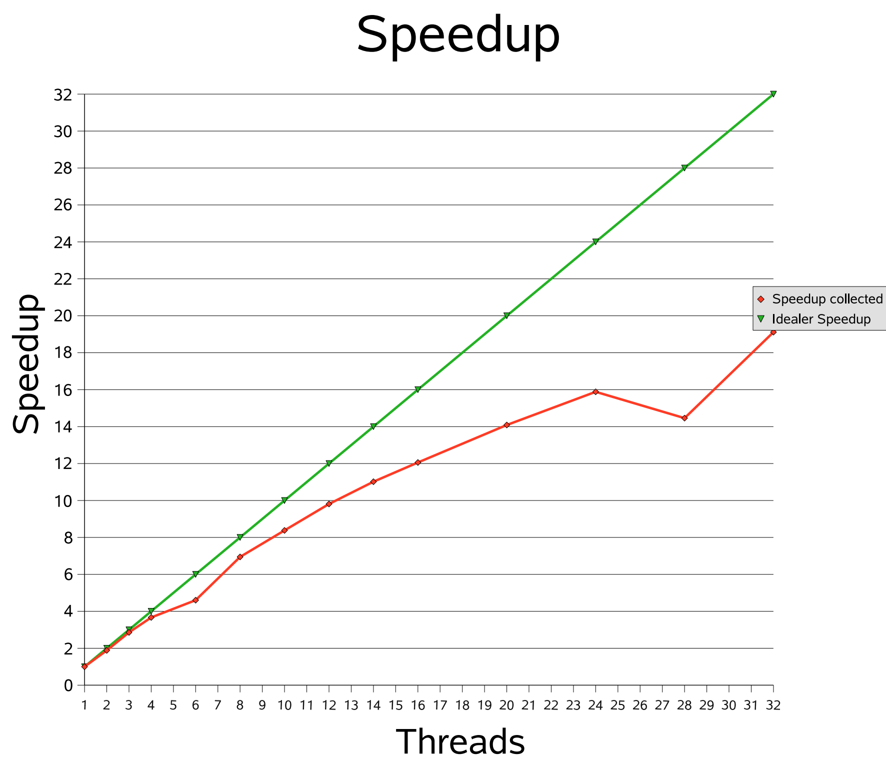


Abbildung 12: Speedup der parallelen Region in Case 4 nach der Optimierung von `PartGo`

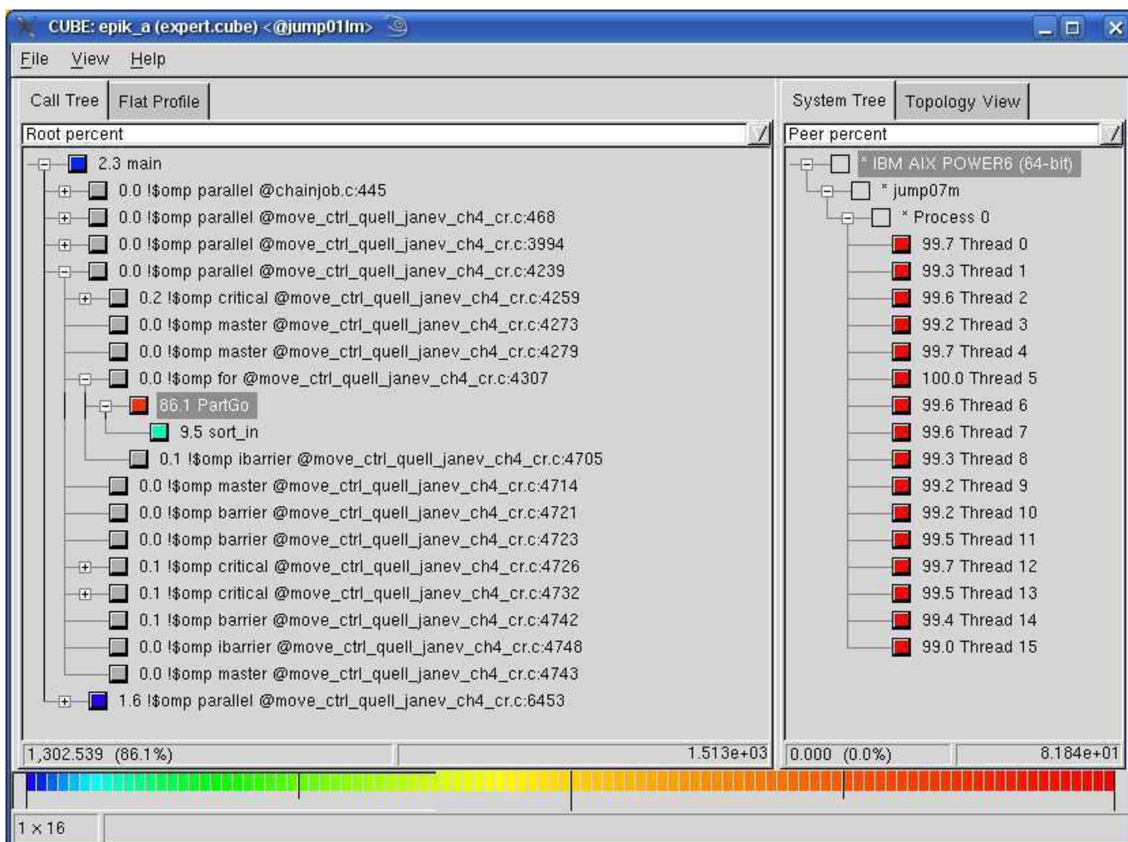


Abbildung 13: CUBE Visualisierung der SCALASCA Analyse von `PartGo` nach der Optimierung (10000 Teilchen, 16 Threads)

5.3.2 sort_in

In der ursprünglichen Version von `sort_in` musste jedes Teilchen eine Kaskade von Bereichsabfragen durchlaufen, um eingeordnet werden zu können. Die Optimierung bedingt jetzt einen gleichen Arbeitsaufwand für jedes Teilchen (Listing 3).

Listing 3: Psuedocode von optimierter `sort_in`-Methode

```
Prozedur: sort_in
Zweck: Lokalisierung eines Teilchens im Gitter
Parameter: Teilchen T, Gitter G

Begin der Prozedur

x = MAX(MIN(T.X + 1, G.MaxX), 0)
y = MAX(MIN(T.Y + 1, G.MaxY), 0)
z = MAX(MIN(T.Z + 1, G.MaxZ), 0)

Sortiere T in G(x,y,z) ein

Ende der Prozedur
```

Dies führt zu einem flüssigerem Programmablauf und zu einer guten Ausnutzung des Cache-Speichers, da das Gitter unfragmentiert im Speicher alloziert ist und deshalb besser im Cache abgebildet werden kann. Die Speedup-Kurve zeigt nun über eine weitere Strecke einen näherungsweise linearen Verlauf an (Abbildung 14), als dies vorher der Fall war.

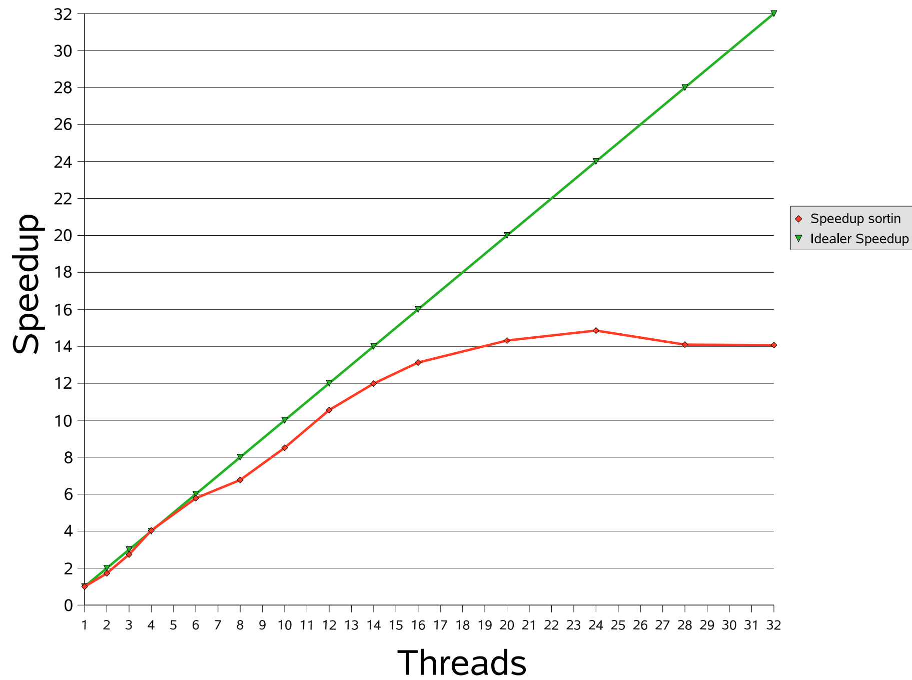


Abbildung 14: Speedup der parallelen Region um `sort_in` nach der Optimierung

Da immer noch ein kleiner Anteil serieller Arbeit im Programm stattfindet, deckt sich die Speedup-Kurve (rot dargestellt) natürlich nicht mit der idealen Speedup-Kurve (grün dargestellt). Sie entfernt sich außerdem bei einem Anstieg der eingesetzten Prozessoren immer weiter von ihrem idealen Verlauf.

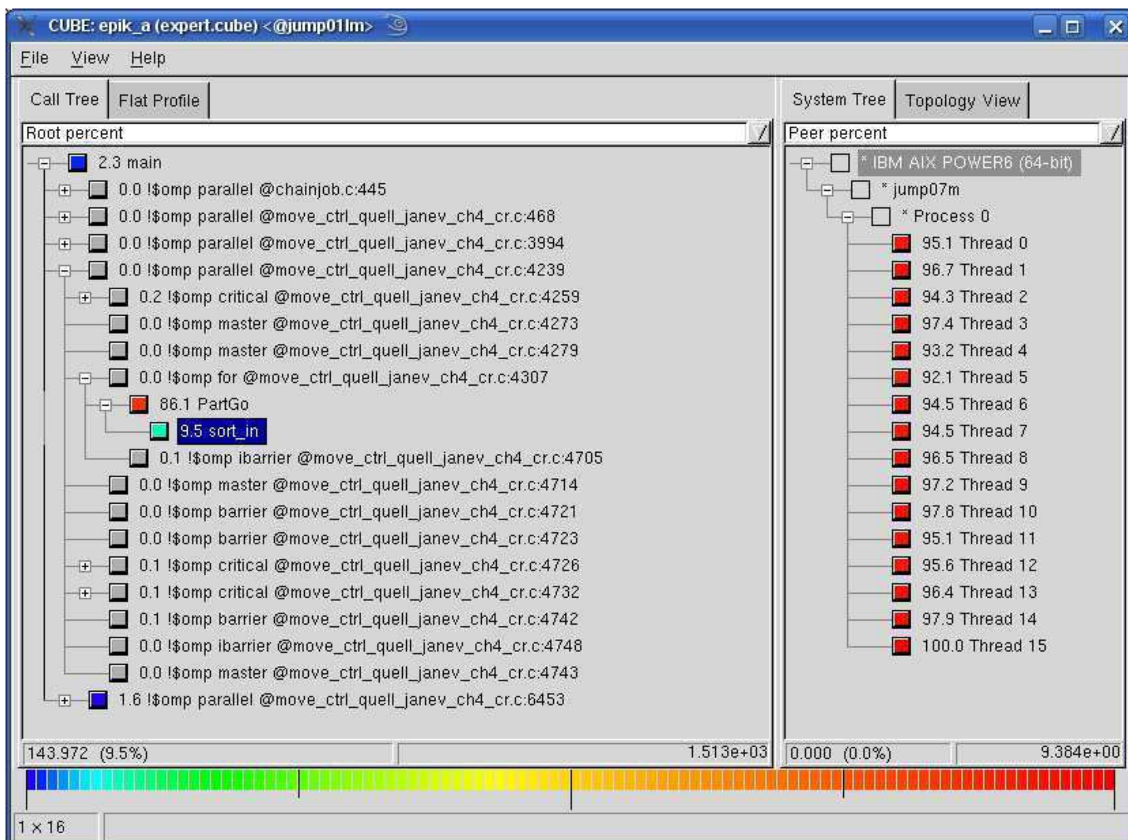


Abbildung 15: CUBE Visualisierung der SCALASCA Analyse von `sort_in` nach der Optimierung (10000 Teilchen, 16 Threads)

Wie man an der KOJAK-Analyse von `sort_in` (Abbildung 15) sieht, ist die Auslastung der Threads jetzt gleichmässiger. Dies lässt sich auch in Tabelle 3 erkennen. Hier wurde unter typischen Simulationsbedingungen mit 30000 Teilchen simuliert. Der Anteil von `sort_in` an der Methode `PartGo` ist von knapp 35 % auf unter 10 % gesunken, womit der Einfluss dieser Methode auf den Speedup gemindert worden ist. Dies ist insofern sinnvoll, da `sort_in` selbst keine physikalischen Berechnungen durchführt und der Anteil an der Gesamtsimulation daher möglichst gering gehalten werden sollte. Dem wurde durch die Optimierung Rechnung getragen. Zudem ist die Wartezeit an der impliziten Barriere stark gesunken. Durch die verbesserte Synchronisierung wird insgesamt weniger Zeit in dieser Methode verbraucht.

Tabelle 3: Auslastung/ % der Methode `sort_in` bei 16 Threads, 30000 Teilchen

Thread	**Auslastung / %**	
	Vor Optimierung	**Nach Optimierung**
0	75.4	96.4
1	71.3	87.1
2	70.6	89.1
3	98.2	94.3
4	68.0	91.7
5	80.1	88.4
6	82.1	100.0
7	91.6	88.3
8	69.5	84.4
9	75.8	87.9
10	73.5	87.2
11	88.9	96.7
12	68.2	97.9
13	100.0	98.7
14	75.2	96.0
15	86.9	90.7
Mittelwert	**79.71**	**92.18**
Standardabweichung	**10.51**	**4.93**
Varianz	**110.49**	**24.29**

Der Mittelwert der Auslastung ist gestiegen und die Standardabweichung respektive die Varianz ist gesunken. Dies sind Charakteristika, die auf eine verbesserte Auslastung aller beteiligten Threads hinweisen. Jeder Thread hat nun in etwa gleich viel zu tun und ist fast komplett ausgelastet. Die gleichmäßige Auslastung aller Prozessoren hat auch deshalb einen so großen Einfluß auf den Performance Gewinn, da am Ende eines parallelen Bereichs alle beteiligten Threads den Masterprozess „joinen“ müssen und dort auf den langsamsten gewartet wird.

$$T_{Bereich} = max\{T_0, T_1, \ldots, T_n\} \tag{11}$$

$T_{Bereich}$: Zeit, welche das Programm für den gesamten parallelen Bereich benötigt
T_i: Zeit, welche Thread i für den parallelen Bereich benötigt

Die Zeit, welche das Programm für die Abarbeitung eines parallelen Bereichs benötigt, ist gerade das Maximum der Zeiten aller beteiligten Threads (Ausdruck 11). Daher bestimmt der langsamste Thread die Laufzeit des gesamten parallelen Bereiches.
Aus den Resultaten der jeweiligen Optimierung von `PartGo` respektive `sort_in` für sich allein genommen lässt sich schon ein positiver Einfluß auf die Gesamtperformance ablesen. Interessant wird aber erst die gleichzeitige Betrachtung beider Optimierungen. Es besteht nämlich die Möglichkeit, dass eine Optimierung die Vorteile der anderen Optimierung teilweise wieder aufhebt, was nicht geschehen sollte. Im Gegenteil ist die Art der Optimierung hier so gewählt worden, dass diese sich gegenseitig ergänzen, oder zumindest nicht neutralisieren. Wie man in Abbildung 16 sieht, ergänzen sich beide Optimierungen sogar und heben den Speedup weiter an. Die gemessene Kurve liegt nun noch enger an der idealen an und es lässt sich ein maximaler Speedup von beinahe 19 erzielen.

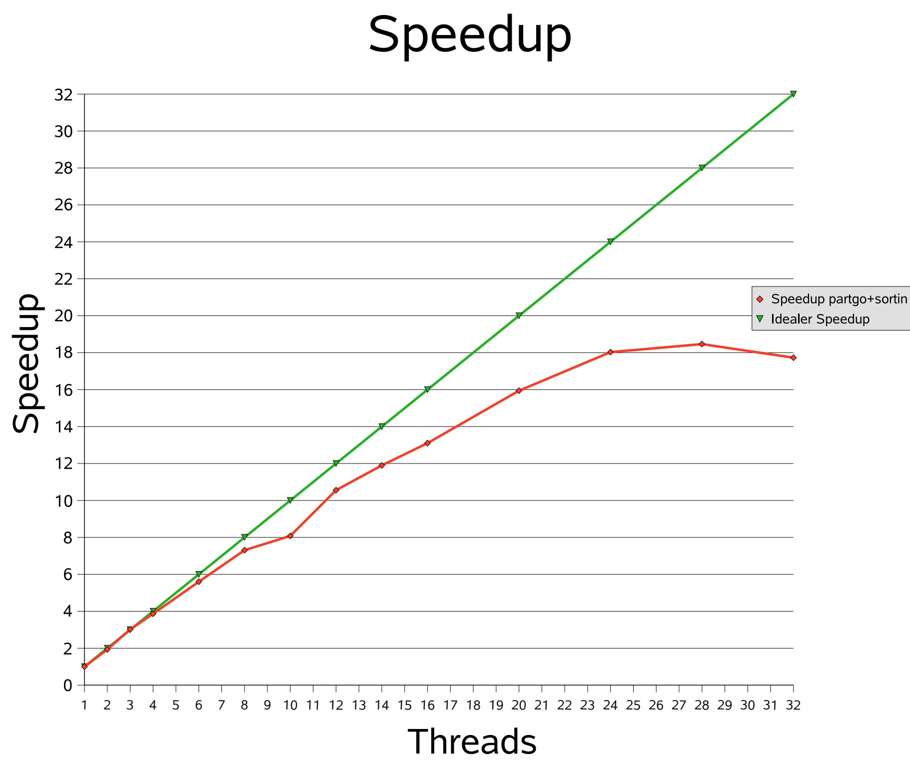


Abbildung 16: Speedup der parallelen Region von Case 4 mit vollständiger Optimierung von `PartGo` und `sort_in`, Simulation von 1000 Teilchen

5.3.3 `PartGo` und `sort_in` im Kontext

Um einen besseren Überblick zu erhalten, werden im Folgenden die Optimierung der Methoden `sort_in` und `PartGo` und der damit einhergehende Speedup detaillierter anhand einer Zusammenstellung der verschiedenen Meßkurven beschrieben (Abbildung 17). Die Speedup-Kurve der Lastverteilung wird hier nicht mehr aufgeführt, da sie mit einer anderen Teilchenanzahl gemessen und den Vergleich damit in den falschen Kontext setzen würde. Gut zu erkennen ist, dass die Kurve der ursprünglichen Version im gesamten De-

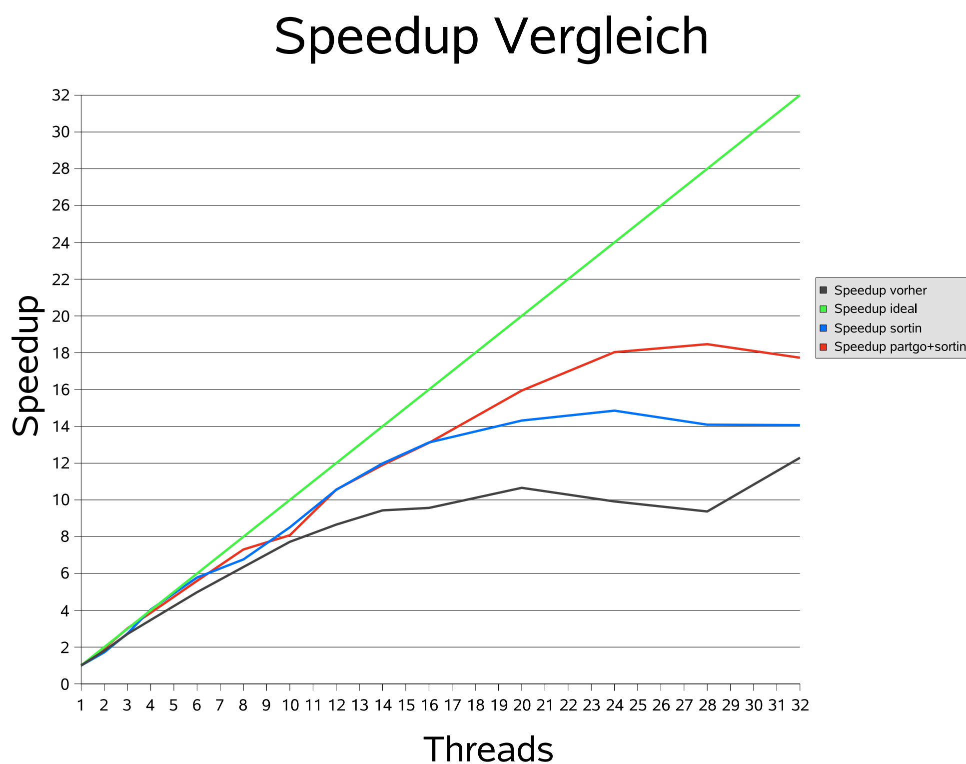


Abbildung 17: Vergleich der verschiedenen Optimierungen mit der Ursprungsversion

finitionsbereich unter den Kurven der optimierten Varianten verläuft. Es wurde also bei geringen Thread-Anzahlen nichts verloren, wogegen auf jeden Fall bei Läufen mit mehr als 10 Threads etwas gewonnen wurde. Ebenso ist zu sehen, dass die rote Kurve im Bereich bis zu 8 Threads im Vergleich zu den anderen am nächsten an der idealen Speedup-Kurve liegt. Nachdem sie dann leicht unter der blauen Kurve verläuft, welche ausschließlich die Optimierung von `sort_in` repräsentiert, liefert sie bei 24 Threads den besten Speedup von allen. An diesem Punkt liegt immer noch eine Effizienz von 71 % vor.

5.3.4 Optimierung der Lastverteilung

Da die Lastverteilung vor der Optimierung zwar dynamisch, die Anzahl der zu verteilenden Stücke aber genau der Anzahl der Prozessoren entsprach, ergab sich im Prinzip eine statische Aufteilung, die den Sinn der dynamischen Verteilung ad absurdum führte. Der Scheduler hatte hierdurch nicht die Möglichkeit, einem Thread nach Abarbeitung eines Stücks, ein neues zur Verfügung zu stellen, weil bereits im ersten Schritt alle Stücke verteilt worden sind.
Zur Verbesserung wurde nun die Stückgrösse um den Faktor $\frac{1}{10}$ reduziert (vgl. Gleichung 12), respektive die Anzahl der Stücke somit um den Faktor 10 erhöht. In der so erhaltenen Speedup-Kurve (Abbildung 18) kann ein näherungsweise linearer Anstieg festgestellt werden, der auch bei höheren Threadzahlen nur unwesentlich abknickt bis er bei 24 Threads stärker abflacht, aber immer noch ein Maximum von 20 aufweist. Das ist der höchste Wert, aller bisher gemessenen Kurven.

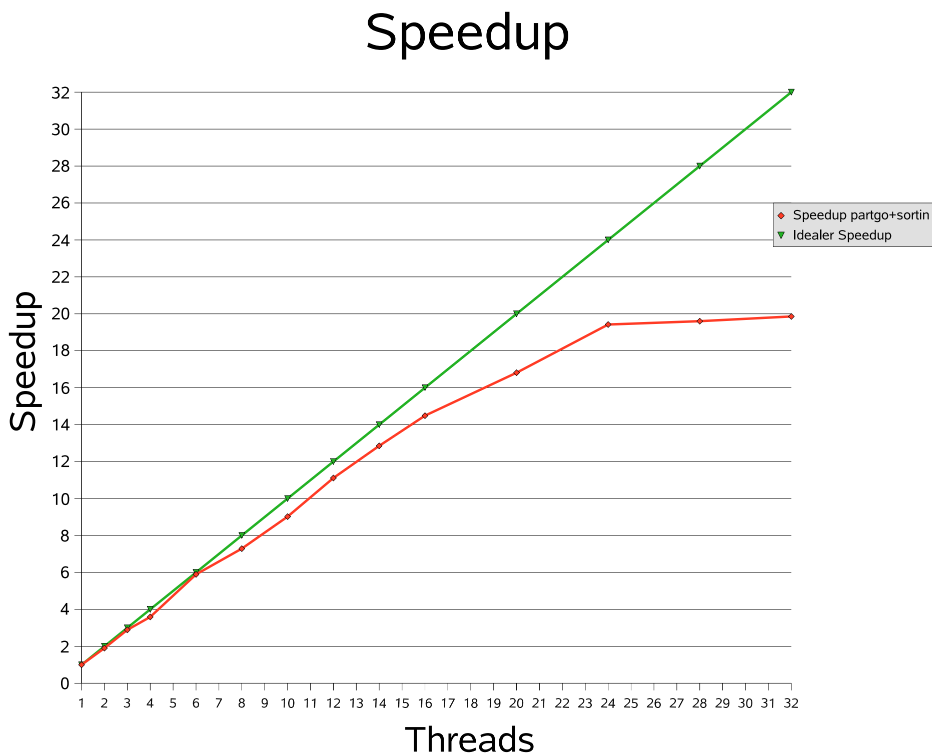


Abbildung 18: Speedup der parallelen Region von Case 4 nach Optimierung der Lastverteilung, Simulation von 30000 Teilchen

Zur näheren Betrachtung des Einflusses der `chunksize` auf die Auslastung der Prozessoren in der untersuchten Region wurde nun in einem statischen Simulationsszenario deren Grösse variiert und die Performancemessung mit SCALASCA aufgezeichnet. Es ergibt sich folgendes Bild:

Tabelle 4: Auslastung / % bei 32 Threads nach Optimierung der Lastverteilung, 30000 Teilchen

Thread \ `chunksize`	**937.5**	**500**	**100**	**50**
0	89.0	71.1	97.0	87.3
1	91.7	90.9	88.8	87.9
2	87.3	91.6	96.1	87.6
3	97.1	95.7	91.2	99.7
4	100.0	91.1	85.7	92.6
5	98.8	67.6	88.3	89.4
6	83.1	93.2	88.9	95.9
7	88.0	91.9	93.8	90.5
8	85.9	94.5	85.6	88.0
9	86.3	68.1	87.4	87.8
10	84.3	100.0	97.1	100.0
11	95.5	96.2	89.9	93.3
12	83.0	95.5	87.6	88.6
13	89.0	89.0	98.8	88.1
14	85.5	90.5	88.7	99.6
15	95.4	96.0	96.8	90.9
16	93.6	92.4	85.1	88.1
17	85.4	91.0	87.9	87.4
18	87.6	94.5	100.0	98.0
19	81.9	94.6	92.6	90.0
20	87.1	89.5	88.0	93.2
21	88.2	91.9	88.6	91.2
22	83.5	99.3	86.2	90.6
23	96.0	91.6	87.6	92.3
24	87.3	92.3	87.9	87.9
25	89.2	90.0	97.7	88.2
26	86.0	95.2	86.5	87.7
27	93.6	89.3	95.2	89.3
28	99.7	91.1	86.0	88.0
29	88.6	89.2	83.8	88.2
30	91.1	62.7	89.0	89.7
31	83.4	95.2	91.8	92.1
Mittelwert	**89.44**	**89.77**	**90.49**	**90.91**
Standardabweichung	**5.23**	**9.09**	**4.58**	**3.86**
Varianz	**27.30**	**82.65**	**20.94**	**14.92**

Die mittlere Auslastung der Threads verbessert sich mit abnehmender `chunksize`, was auf die Anzahl der zu verteilenden Stücke zurück zu führen ist. Bei konstanter Problemgrösse N und konstanter Prozessoranzahl P gilt:

$$K(\texttt{chunksize}) = \frac{N}{P \cdot \texttt{chunksize}} \Rightarrow K \sim \frac{1}{\texttt{chunksize}} \tag{12}$$

N: Problemgrösse
P: Anzahl eingesetzter Threads
K: Anzahl der Stücke
`chunksize`: Grösse der Stücke bei dynamischer Lastverteilung

Die Anzahl der Stücke ist also umgekehrt proportional zur `chunksize`. Je weniger Stücke an die ausführenden Threads verteilt werden können, desto weniger Einfluss hat die dynamische Lastverteilung auf die Auslastung. Deshalb ist es sinnvoll einen niedrigen Wert für die Stückgrösse festzulegen, der aber aufgrund des Verwaltungsoverheads der dynamischen Lastverteilung nicht allzu klein sein sollte.

5.3.5 Vergleich der Laufzeiten

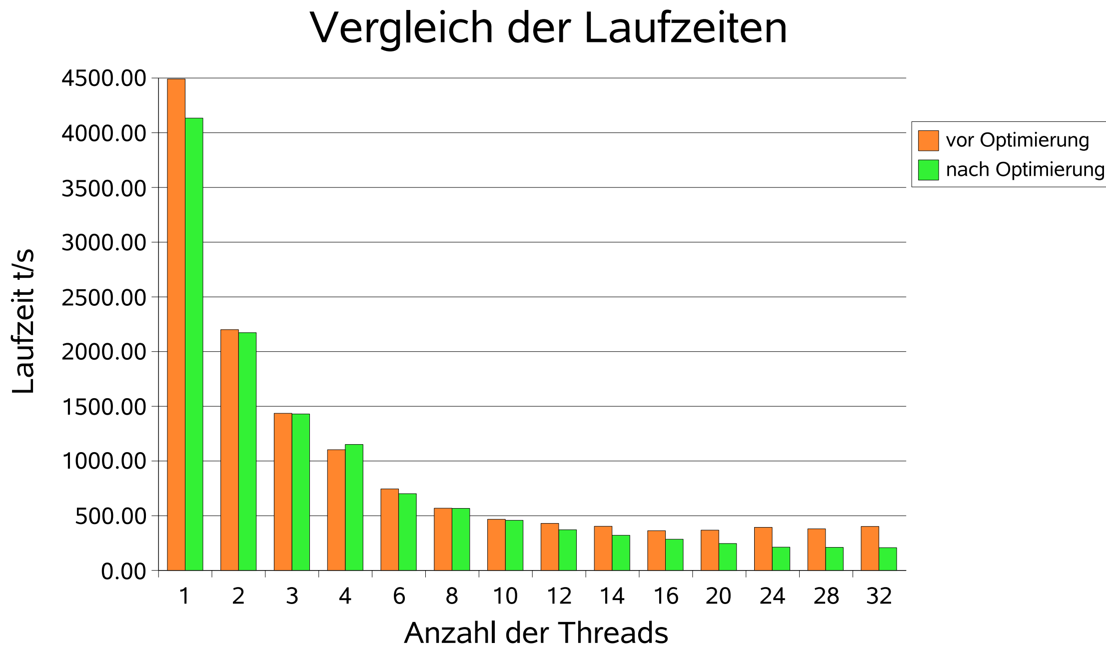


Abbildung 19: Zeitbedarf von ERO nach erfolgter Optimierung

In Abbildung 19 ist nun der Zeitbedarf von ERO vor und nach der Optimierung mit angepasster `chunksize` visualisiert worden. Man sieht fast über den gesamten Verlauf eine Verringerung des Zeitbedarfs nach erfolgter Optimierung. Vor allem aber lässt sich bei hohen Threadanzahlen eine Halbierung der Programmlaufzeit beobachten (20 - 32 Threads), die wesentlich auf die verbesserte Lastverteilung zurückzuführen ist. Leider lässt sich bei kleinerer Threadanzahl kein signifikanter Effekt festestellen, was aber noch Raum für weitere Untersuchungen lässt. Dieses Verhalten lässt sich auch in den gemessenen Speedup-Kurven beobachten (Abbildung 17).

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Das Ziel der Arbeit, die 3D Monte-Carlo-Simulation ERO so zu beschleunigen, dass die notwendige Diskretisierung und die simulierten Effekte, mit einem annehmbaren statistischen Fehler, in einer realistischen Zeitspanne, berechnet werden können, wurde durch die durchgeführte Parallelisierung und Optimierung erreicht.
Obwohl die Parallelisierung eines Monte-Carlo-Codes auf den ersten Blick trivial erscheint, da die simulierten Teilchen statistisch unabhängig sind und auch unabhängig voneinander verfolgt werden, bedingt ihre Wechselwirkung im Plasma und an den Gefässwänden eine Abhängigkeit, auf die bei der Parallelisierung Rücksicht genommen wurde.Die beteiligten Threads arbeiten nämlich im gleichen räumlichen Gebiet und beeinflussen sich so gegenseitig.
Es wurde zunächst eine parametrische Parallelisierung konstruiert, die den Anforderungen bei der Simulation mehrerer Szenarienvariationen entspricht. Die Implementierung wurde dabei durch die Wahl von MPI so gestaltet, dass ein hybrider Betrieb von MPI und OpenMP auf geeigneten Systemen problemlos möglich ist.
Danach wurde die aktive Parallelisierung mit OpenMP in dem vorher festgelegten Szenario „Transport von extern eingeführten Teilchen“ untersucht und ihre Schwachstellen analysiert. Darauf aufbauend war es möglich, den Programmablauf an den entscheidenden Stellen derart zu modifizieren, dass eine bessere Auslastung der beteiligten Threads erzielt werden konnte. Es wurden verschiedene programmiertechnische und auch auf die Rechnerarchitektur bezogene Optimierungen auf den Code angewendet, um dieser Zielsetzung gerecht zu werden, wie etwa die Zusammenlegung von Speicherbereichen und die Optimierung von Speicherzugriffsschemata. Dabei stellte sich heraus, dass vor allem die gleichmässige Auslastung ein wichtiger Parameter bei der Realisierung einer Speedupsteigerung ist.
Durch die beschriebenen Optimierungen ließ sich letztendlich ein Performancegewinn von nahezu 100 % erreichen, womit das Ziel dieser Arbeit in hohem Maße erfüllt wurde. Ein typischer Simulationslauf im Szenario „Transport von extern eingeführten Teilchen „ unter Nutzung von 24 Threads benötigt jetzt nur noch die Hälfte der Zeit, die er vor der Optimierung verbraucht hätte. Dieser Fall ist auch wichtig für die Erforschung von geeigneten Wandmaterialien für ITER. Durch den so gewonnenen Geschwindigkeitsvorteil liefert die Parallelisierung einen wichtigen Beitrag für dieses Projekt.

6.2 Ausblick

Trotz der erfolgreichen Parallelsierung bleibt am Gesamtprogramm ERO noch Raum für Verbesserungen. Für die Fortführung dieser Arbeit wäre beispielsweise eine Programmfluß-steuerung interessant, welche sowohl die parametrische Parallelisierung als auch die aktive Parallelisierung mit OpenMP eng verzahnt und auch in der Lage ist, die LoadLeveler-Batchfiles dementsprechend zu verändern, so dass ein simultaner Betrieb beider Parallelisierungen auf hybriden System bei der Simulation von mehreren Szenarien möglich ist.
Außerdem sollte die Optimierung der anderen Simulationsfälle mit ebenfalls schlechtem Speedup-Verhalten fortgeführt werden, um den globalen Performancegewinn nach oben zu treiben. Dies ist vor allem im Hinblick auf den Fall 5 und 6 (vgl. Tabelle 1) „Transport von chemisch und physikalisch erodierten Teilchen“ interessant, da sich hier durch die richtige Verteilung der Teilchen ebenfalls eine bessere Auslastung ergeben würde. Dies ließe sich beispielsweise durch eine sinnvolle Mischung verschiedenster Teilcheneigenschaften auf alle beteiligten Threads oder durch eine gut optimierte dynamische Lastverteilung erreichen.

Literatur

[1] Forschungszentrum Jülich. *Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks*. `http://www.fz-juelich.de/zam/kojak`.

[2] Forschungszentrum Jülich. *Scalable Performance Analysis of Large-Scale Applications*. `http://www.scalasca.org`.

[3] Horst Stöcker. *Taschenbuch der Physik*. Verlag Harri Deutsch, 2007. 5. Auflage, Seite 540.

[4] Francis F. Chen. *Introduction to Plasma Physics and Controlled Fusion*. Plenum Press, 1984.

[5] F. Wagner and I. Milch. *Der Stellarator Wendelstein 7-X*. Max-Planck-Institut für Plasmaphysik, 2005.

[6] U. Samm. *E05 Kernfusion und Plasmaforschung*. Institut für Plasmaphysik Forschungszentrum Jülich, 2002. `http://www.fz-juelich.de/scientific-report-2002/index.php?_p=12&fe=33`.

[7] F.W. Perkins et al. *ITER Physics Basis*. Nuclear Fusion, Vol.39, No. 12, 1999.

[8] Forschungszentrum Jülich. *IBM p6 575 Cluster JUMP Configuration*. Forschungszentrum Jülich. `http://www.fz-juelich.de/jsc/jump`.

[9] N. Metropolis. *Equation of State Calculations by Fast Computing Machines*. J. Chem. Phys., 1953.

[10] Thomas Rauber und Gudula Rünger. *Parallele Programmierung*. Springer, 2007. 2. Auflage.

[11] A.Kirschner et.al. *Nucl. Fusion 40 No. 5*. 2000. Seiten 989-1001.

[12] D. Naujoks, R. Behrisch, J.P. Coad, and L.deKock. *Nucl. Fusion 33*. 1993. Seiten 581ff.

[13] U. Kögler und J. Winter. *ERO-TEXTOR - 3D-Monte-Carlo Code for Local Impurity-Modeling in the Scrape-Off-Layer of TEXTOR*. Forschungszentrum Jülich, 1997. 2. Auflage.

[14] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skorvia. *Workload Management with LoadLeveler*. IBM International Technical Support Organisation, first edition, 2001. `http://www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf`.

[15] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B. J. N. Wylie. *Scalable Collation and Presentation of Call-Path Profile Data with CUBE*. John von Neumann Institute for Computing, 2007. `http://www.fz-juelich.de/jsc/datapool/KojakPubs/parco2007.pdf`.