Jül-4320

Jülich Supercomputing Centre (JSC)

Entwicklung eines automatischen Validierungssystems für Simulationscodes der Fusionsforschung

Andreas Galonska

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Entwicklung eines automatischen Validierungssystems für Simulationscodes der Fusionsforschung

Andreas Galonska

Berichte des Forschungszentrums Jülich; 4320
ISSN 0944-2952
Jülich Supercomputing Centre (JSC) Jül-4320

Vollständig frei verfügbar im Internet auf dem Jülicher Open Access Server (JUWEL)
unter http://www.fz-juelich.de/zb/juwel

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek, Verlag
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461 61-5220 · Telefax: 02461 61-6103 · e-mail: zb-publikation@fz-juelich.de

Kurzfassung

In der vorliegenden Masterarbeit wird die Entwicklung eines automatischen Validierungssystems für den Simulationscode ERO dokumentiert. Dieser 3D Monte-Carlo Code modelliert den Transport von Verunreinigungen sowie Plasma-Wand-Wechselwirkungs-Prozesse und hat große Bedeutung für die Fusionsforschung. Das Validierungssystem basiert auf JuBE („Julich Benchmarking Environment“), dessen Flexibilität eine leichte Erweiterung des Systems auf andere Codes, z.B solche, die in Rahmen der EU Task Force ITM (**I**ntegrated **T**okamak **M**odelling) betrieben werden, erlaubt. Es wurde zunächst analysiert, welche Anforderungen an ein solches System zu stellen sind. Die gewählte Lösung - JuBE und ein spezielles Programm zum „intelligenten“ Vergleich von aktuellen und Referenz-Ausgabedaten von ERO - wird beschrieben und begründet. Die Benutzung dieses Programms und die Konfiguration von JuBE werden detailliert beschrieben. Simulationen zu unterschiedlichen Plasmaexperimenten, die als Referenzfälle für die automatische Validierung dienen, werden erläutert. Die Arbeit des Systems wird durch die Beschreibung eines Testfalls illustriert. Dieser behandelt die Fehlerlokalisierung und Nachbesserung bei der Parallelisierung eines wichtigen ERO Moduls (Verfolgung von physikalisch erodierten Teilchen). Es wird demonstriert, wie das System bei einer fehlgeschlagenen Validierung reagiert und die anschließend durchgeführte Fehlerbehebung zu einem positiven Ergebnis führt. Zum Schluss wird eine Speedupkurve der Parallelisierung anhand der Ausgabedaten von JuBE erstellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Automatische Validierung

Die Computersimulation entwickelt sich, neben der Theorie und dem experimentellen Nachweis, zum dritten Standbein der Wissenschaft. Theoretische Grundlagen bilden hierbei die Basis,um ein existierendes Problem mit Hilfe numerischer Modelle beschreiben zu können und dadurch in der Lage zu sein, experimentelle Ergebnisse zu reproduzieren oder vorauszusagen. Dies ist oft eine kostengünstige Alternative zu praktischen Versuchen, die einerseits teuer und aufwändig, oder aber praktisch nicht durchführbar sind. Die Computersimulation bietet die Möglichkeit, den Einfluss von einzelnen Effekten zu untersuchen oder das Zusammenspiel von mehreren bekannten Effekten zu verstehen. Ein wichtiges Kriterium für die Simulationsqualität ist dabei die Korrektheit der Ausgabedaten, die schließlich einen real existierenden Sachverhalt möglichst genau wiedergeben sollen.
Dies gewährleistet das zu Grunde liegende Modell, welches jedoch im Laufe des Entwicklungsprozesses immer weiter verbessert wird und an dem möglicherweise viele Personen gleichzeitig arbeiten. Es ist oft sehr kompliziert vorherzusagen, welche Folgen kleinste Änderungen am Code verursachen. Diese Folgen können gewünscht sein oder aber zu Fehlern im Programm führen, welche die Korrektheit des Modells gefährden. Zudem können selbst gewünschte Änderungen an einem Teil unerwarteten Einfluss auf andere Teile des Programms haben. In allen Fällen ist es jedoch hilfreich, Änderungsfolgen automatisch diagnostizieren zu können.
Im Hinblick auf diese Problematik hat die EU Taskforce ITM (**I**ntegrated **T**okamak **M**odelling), die sich mit der Modellierung von Fusionsexperimenten (s. Kap. 2.1) befasst, festgelegt, dass alle innerhalb dieses Projektes eingesetzten Codes Programme oder Module zur automatischen Validierung bereitstellen sollen.
Die beteiligten Codes sollen in einem zentralen Workflow Management System zusammengefasst werden [12]. Dies geschieht im Hinblick auf die Entwicklung des Fusionsexperimentes ITER (**I**nternational **T**hermonuclear **E**xperimental **R**eactor) [15], welches momentan in Cadarache (Frankreich) gebaut wird. Um Aussagen über das Plasma im Inneren dieses Experimentes, die Parameter der einzusetzenden Materialien in den Wänden und schließlich der

Verfügbarkeit von ITER machen zu können, soll es zunächst möglichst realitätsnah modelliert werden. Die dabei eingesetzten Codes sollten den Anspruch haben, über ihren gesamten Entwicklungsprozess korrekte Ergebnisse zu produzieren, welche die Wirklichkeit in einem gewissen Rahmen abbilden. Einer dieser dabei eingesetzten Modellierungs-Codes ist der 3D Monte-Carlo Code ERO, welcher am Forschungszentrum Jülich entwickelt wird.
Es ist notwendig, besonders vor der Aktualisierung des Codes im Versionsverwaltungssystem, sicher zu stellen, dass die Konsistenz der Ausgabedaten weiter gewährleistet ist. Dies geschieht, indem man die mit dem neuen, weiter entwickelten Code erhaltenen Daten mit Daten aus früheren Läufen vergleicht. Die neuen Daten sollten dabei in einem gewissen Rahmen, der sich bei vielen Simulationen auch aus ihrer Monte-Carlo-Basiertheit und dem daraus resultierenden statistischen Fehler ergibt, mit den alten Daten übereinstimmen. Dies trifft natürlich nicht zu, wenn die neuen Ausgabedaten durch gewollte Code-Änderungen bedingt sind. In diesem Fall sollten die neuen Daten als Referenzdaten für spätere Vergleiche dienen.
Da ERO im Workflow System in einem Verbund aus anderen Fusionscodes, die unterschiedliche Aspekte des Experimentes modellieren sollen, eingesetzt wird, sind die produzierten Ausgabedaten vorher zu überprüfen, um den anderen Codes eine korrekte Datenbasis für die weitere Simulation zur Verfügung zu stellen. Die Überprüfung selbst stellt dabei einen nicht zu unterschätzenden manuellen Aufwand dar. Daher wäre es wünschenswert, eine Software einsetzen zu können, die dem Entwickler diese Aufgaben abnimmt, so dass er nur bei fehlerhafter Validierung manuell eingreifen muss.
Da die Ausgabedaten einer Simulation aufgrund ihres Anspruchs, existierende Sachverhalte möglichst genau wiederzugeben, einen großen Umfang haben können, ist es sinnvoller, diese auf wenige charakteristische Werte zu reduzieren und sie anhand dieser zu vergleichen. Bei Monte-Carlo basierten Simulationen ergibt sich zudem durch minimale Abweichungen, die aus den erzeugten unterschiedlichen Pseudo-Zufallswerten resultieren, die Problematik, dass diese überhaupt nicht exakt verglichen werden können. Auch hier ist es von großem Vorteil diese Werte zunächst durch bestimmte Methoden zu reduzieren und diese anhand eines vorher festgelegten maximalen Fehlers zu vergleichen. Im Rahmen dieser Arbeit wird ein automatisches Validierungssystem, inklusive der notwendigen Software-Tools, für ERO entwickelt, welches den Großteil der Validierungsarbeit leistet.

1.2 Motivation

Der manuelle Aufwand bei der Validierung von Ausgabedaten einer Simulation beinhaltet folgende Aufgaben:

- Kompilierung
- Vorbereitung des Ausführungsverzeichnisses
- Ausführung der Simulation
- Analyse der Ausgabedaten

Dabei stellen rein technische Vorgänge, wie beispielsweise die Vorbereitung des Ausführungsverzeichnisses und der Vergleich der Ausgabedaten, manuell durchgeführt, einen erheblichen Zeitaufwand dar. Bei der Vorbereitung des Ausführungsverzeichnisses muss die gewünschte Simulation zunächst kompiliert werden. Hier ergeben sich vor allem Unterschiede hinsichtlich der Plattform, auf der sie läuft. Dies ist zum einen der verwendete Compiler, der von Plattform zu Plattform unterschiedlich sein kann, da er den Maschinencode aus dem Quelltext erzeugt und unterschiedliche Plattformen meist auch unterschiedliche Prozessoren einsetzen. Diese erreichen aufgrund spezieller Maschinenbefehle bessere Ausführungsgeschwindigkeiten, wenn sie mit Binärcode eines passenden Compilers gesteuert werden. Zudem kann die Simulation auf Quelltextebene, durch Nutzung spezieller Makrodefinitionen, in unterschiedlichen Versionen oder Szenarien resultieren. So kann man aus demselben Quelltext Simulationen erhalten, die sich in einem gewissen Rahmen unterscheiden. Diese unterschiedlichen Szenariosimulationen müsste man bei manueller Vorgehensweise alle gesondert kompilieren, um den vollen Funktionsumfang der Simulation testen zu können.
Die verschiedenen Szenarien brauchen, bedingt durch ihre Unterschiede, meist auch speziell zugeschnittene Parameterdatensätze und Eingabedateien, die den Verlauf der Simulation steuern. Diese müssen alle vorgefertigt vorgehalten oder einzeln modifiziert werden, was bei wissenschaftlichen Anwendungen, die eine sehr genaue Spezifikation der Eingabeparameter verlangen, zahlreiche Werte sein können. Diese müssten ohne ein automatisches System, an das jeweilige Szenario angepasst und manuell konfiguriert werden.

Zur Reduzierung des Zeitverbrauchs ist daher ein automatisches Validierungssystem (im Folgenden AVS) unabdingbar, das dem Entwickler die oben genannten Aufgaben abnimmt. Es soll zudem dynamisch und leicht zu konfigurieren sein, damit sich der Zeitaufwand nicht durch eine langwierige Konfiguration wieder erhöht.
Das Jülich Benchmarking Environment, JuBE, ist ein für diese Aufgaben geeignetes System, welches durch seine XML-basierte Konfiguration und seine vollständige Protokollierung der Vorgänge während der Validierung eine ideale Basis bietet. Vorgefertigte Templates, beispielsweise für Batch-Scripts, machen es zudem relativ plattformunabhängig. Während dieses universelle System problemlos in der Lage ist, die ersten drei definierten Aufgaben zu erfüllen,

hat es aufgrund der Unterschiede in den Ausgabedaten verschiedener Programme, respektive Simulationen, keine Möglichkeit die Ergebnisse zu analysieren. Es kann lediglich den Analyseprozess starten. Dafür ist ein separates, ERO-spezifisches, Tool notwendig, das „ERO Testing Tool“, im folgenden ERO.ETT genannt. Dieses Tool dient als „intelligenter Vergleicher“ zwischen den aktuell zu testenden und den ERO-Referenz-Ausgabedaten.
ERO.ETT wird von JuBE im Verifikations- und Analyseprozess ausgeführt und die Analyseergebnisse werden wiederum von JuBE verarbeitet und in einen automatisch erstellten Bericht übernommen. So stellen diese beiden Tools, JuBE und ERO.ETT, die Grundpfeiler des AVS dar.

Kapitel 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 angeregten Elementen abhängt (Abbildung 2.1). Da Atomkerne aus Neutronen (Nukleonen ohne elektrische Ladung) und Protonen (Nukleonen mit positiver elektrischer Ladung) bestehen, sind diese insgesamt positiv geladen. 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 kurzreichweitige Kernkraft, welche die Nukleonen zusammenhält. 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. Der quantenphysikalische Tunneleffekt postuliert eine gewisse Wahrscheinlichkeit, mit der Teilchen in der Lage sind, diese Barriere zu überwinden. Dadurch ist es auch Teilchen mit einer geringeren Energie 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 exotherm ab und ist daher zur Energiegewinnung auch nur hier sinnvoll.
In Abbildung 2.1 ist zu erkennen, dass das bei der Verschmelzung von Deuterium ($^{2}_{1}$H) und Tritium ($^{3}_{1}$H) erzeugte Helium ($^{4}_{2}$He) eine höhere Bindungsenergie als das Ausgangsmaterial besitzt. Diese Energie wird bei der Fusion freigesetzt und äußert sich zudem in einem Massendefekt des entstehenden Reaktionsproduktes. Dessen summierte Masse ist nämlich leichter

als die Summe der Massen der Ausgangsteilchen. Der Massendefekt lässt sich nach der allgemeinen Relativitätstheorie aus der Äquivalenz von Masse und Energie, $E = mc^2$, erklären. Es ist außerdem gut zu erkennen, dass beim Element Eisen (Fe) eine Grenze existiert. Ab dieser Grenze muss man zur Fusion der Elemente mehr Energie aufwenden als durch die Reaktion frei wird, da das Fusionsprodukt eine geringere Bindungsenergie besitzt.

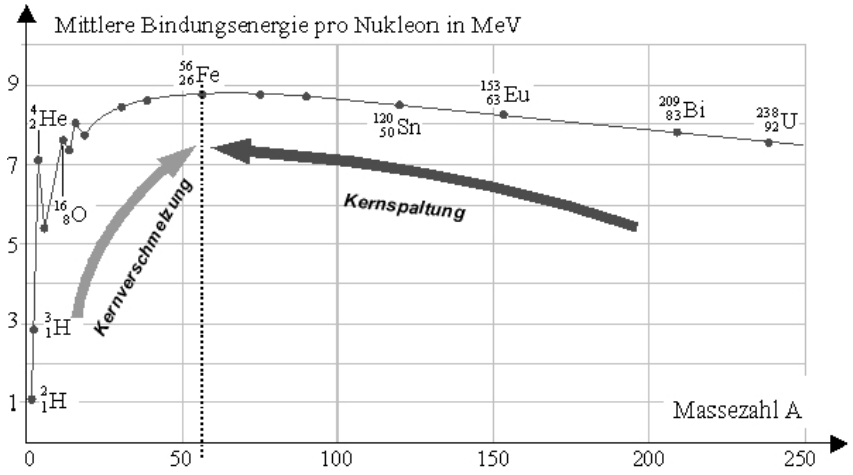


Abbildung 2.1: Bindungsenergie in Abhängigkeit von der Massenzahl[1]

[1] Quelle: `http://commons.wikimedia.org/wiki/File:Bindungsenergie_Massenzahl.jpg` [2]

2.1.1 Kernfusion als Energiequelle

Sterne, wie die 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 [30].

$$
\begin{aligned}
{}^{2}_{1}\mathrm{D} + {}^{3}_{1}\mathrm{T} &\rightarrow {}^{4}_{2}\mathrm{He} + {}^{1}_{0}\mathrm{n} + 17,60MeV \\
{}^{2}_{1}\mathrm{D} + {}^{2}_{1}\mathrm{D} &\rightarrow {}^{3}_{1}\mathrm{T} + {}^{1}_{1}\mathrm{H} + 4,04MeV \\
{}^{2}_{1}\mathrm{D} + {}^{2}_{1}\mathrm{D} &\rightarrow {}^{3}_{2}\mathrm{H} + {}^{1}_{0}\mathrm{n} + 3,27MeV
\end{aligned}
$$

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

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 [7].

2.1.2 Experimentelle Kernfusion

Es gibt bereits mehrere Fusionsexperimente, die beispielsweise nach dem Tokamak-Prinzip arbeiten. Allerdings arbeiten diese Systeme nicht wirtschaftlich (Ausdruck 2.1), da die zu ihrem Betrieb notwendige Energie die Energie, welche bei der Fusion freigesetzt wird, übersteigt.

$$Q = \frac{P_{Fusion}}{P_{Heiz}} \tag{2.1}$$

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

Dabei bedeutet $Q < 1$, dass weniger Energie freigesetzt als dem System zugeführt wurde, was nahezu 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. Bei einer solch hohen Temperatur liegen alle Atome ionisiert vor. Dieses Gas aus ionisierten Atomen nennt man Plasma. Das hitzebeständigste Element, Wolfram, hat einen Schmelzpunkt von ca. 3400 ˚C. Deshalb ist es unmöglich, das Fusionsplasma mechanisch einzuschließen.

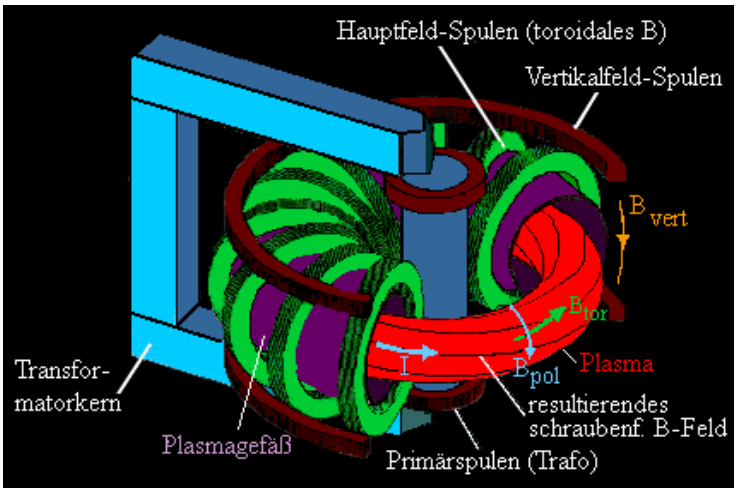


Abbildung 2.2: Prinzip eines Tokamaks[2]

Da Plasma aber aus Ionen besteht, gibt es die Möglichkeit eines helikalen, magnetischen Plasmaeinschlusses, wie dies zum Beispiel in einem Tokamak- oder Stellarator-System geschieht.

[2]Quelle: `http://leifi.physik.uni-muenchen.de/web_ph09_g8/umwelt_technik/09fusion/tokamak.gif` [3]

Dabei wird das Plasma in einem, von torusförmig angeordneten Spulen erzeugten Magnetfeld, in einem Ring „gehalten“. In Abbildung 2.2 ist die Schematik eines Tokamak-Systems dargestellt.
Man sieht, dass die torusförmig angeordneten Spulen das toroidale Magnetfeld B_{tor} erzeugen. Zudem ist es, aufgrund von Effekten, die eine Ladungstrennung bewirken, 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 Magnetfeldlinien durch Induktion eines Stromes in das Plasma erzeugt, wohingegen bei einem Stellarator die geometrische Anordnung der Spulen für die Verdrillung sorgt. Solch ein Stellarator ist etwa Wendelstein 7-X, der derzeit am Max-Planck Institut für Plasmaphysik in Greifswald gebaut wird und bis 2014 fertig gestellt sein soll [33].
Zur Zündung muss das Fusionsplasma aufgeheizt werden. Dies geschieht durch verschiedene Mechanismen:

- Ohmsche Heizung
  Der durch die Transformationsspule in das Plasma induzierte Strom bewirkt gleichfalls eine Erhöhung der Plasmatemperatur, die jedoch zur Zündung einer selbst-tragenden Fusionsreaktion nicht ausreicht.
- 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 der Teilchen sehr effektiv absorbiert wird, erhöht sich die Temperatur des Plasmas.

Das Lawson-Kriterium (Ungleichung 2.2) liefert eine Abschätzung der Zeit, die der Plasmaeinschluss zusammen gehalten werden muss, damit eine sich selbst erhaltende Fusionsreaktion stattfindet, die Energie liefert.

$$n\tau > \frac{12kT}{\overline{\sigma v} \cdot U_f} \sim 10^{15} \frac{s}{cm^3} \tag{2.2}$$

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 α-Teilchen (Heliumkerne) dem dichten Plasma genug Energie zuführen können, um Energieverluste durch Wärmeleitung, Konvektion und Abstrahlung zu kompensieren.

2.1.3 Kernfusionsforschung am Forschungszentrum Jülich

Die Erforschung neuer Energieträger, die eine Unabhängigkeit von fossilen Brennstoffen vorantreiben soll, befasst sich unter anderem mit der Kernfusion zur Energiegewinnung, welche gegenüber der schon seit einem halben Jahrhundert praktizierten Kernspaltung einige Vorteile bietet. Die Vorteile liegen dabei in der höheren Reaktorsicherheit, die ein Kernfusionsreaktor bieten würde und in der beinahe unbegrenzten Verfügbarkeit des Brennstoffes. 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 (**T**okamak **EX**periment for **T**echnology **O**riented **R**esearch) [28], welches insbesondere der Erforschung der Plasma-Wand-Wechselwirkung (im folgenden PWW) und des Verunreinigungstransports 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 und den daraus resultierenden Wärmeflüssen ausgesetzt sind. Dabei sollen diese Materialien bei Beschuss von energiereichen Teilchen möglichst wenige Teilchen selber freisetzen, was zu Verunreinigungen und damit zu einem Strahlungskollaps des Plasmas führen könnte. Die Erosion dieser Stoffe limitiert auch die Lebensdauer der plasma-begrenzenden Komponenten. Außerdem darf die Einlagerung des Brennstoffes, radioaktives Tritium, in der Wand das Sicherheitslimit von ca. 700 g nicht überschreiten. Die Prozesse der PWW können in TEXTOR an speziellen Wandkomponenten, so genannten Testlimitern, untersucht werden. Innerhalb von TEXTOR sind zwei „Limiter Schleusen“ installiert, womit verschiedene Testlimiter an den Plasmarand gefahren werden können. Die Erkenntnisse, die sich aus den Simulationen, den praktischen Experimenten in TEXTOR und weiteren Fusionsexperimenten ergeben, bilden die Basis für die Entwicklung und Konstruktion des internationalen Kernfusionsexperimentes ITER, das ab 2009 gebaut und dann in etwa 10 Jahren fertig gestellt werden soll. 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.

2.2 Eingesetzte Supercomputer

Da ERO, durch seine aufwändigen physikalischen Berechnungen, im sequentiellen Betrieb auf gewöhnlichen Workstation-Computern zu viel Zeit konsumiert, wurde für rechenintensive Teile des Codes eine Parallelisierung durchgeführt [18]. Um diese Parallelisierung effizient und unter Nutzung mehrerer Prozessoren einsetzen zu können, ist die Nutzung eines Supercomputers unabdingbar. Dafür stehen am Forschungszentrum mehrere Systeme bereit, die im folgenden beschrieben werden.

2.2.1 IBM Power6 575 Cluster - JUMP

Die Masterarbeit wurde auf dem Jülicher Multiprozessor (JUelicher Multi Prozessor) nach seiner Migration auf das Power6 System von IBM begonnen.
JUMP besteht aus 14 SMP [3] Knoten mit jeweils 32 SMT [4] Prozessoren, also insgesamt 448 Prozessoren. Innerhalb eines jeden Knotens teilen sich die Prozessoren einen gemeinsamen Adressraum von 128 GByte (1,8 TByte total). Jeder Prozessor hat eine Taktrate von 4,7 GHz und besitzt folgende Caches:

- **L1:** 64kByte
- **L2:** 8 MByte pro Dual-Core-CPU
- **L3:** 32 MByte (extern, nicht auf der Die[5] integriert)

Die einzelnen Knoten sind über ein schnelles Infiniband Netzwerk gekoppelt und erreichen eine Peak Performance[6] von 8,4 TFLOPS [7], respektive 5,4 TFLOPS im Linpack Benchmark [19]. Der Linpack Benchmark ist ein Quasi-Standard zur Leistungsmessung von Hochleistungsrechnern.
Auf JUMP kommen der IBM XL C/C++ Compiler (Version 10.1) und der XL-FORTRAN Compiler (Version 12.1) zum Einsatz und müssen dementsprechend vom AVS zur Kompilierung des Programms verwendet werden.

[3] Symmetrisches Multiprozessing, vgl. Kap. 2.5
[4] Simultaneous Multithreading: Mehrere Berechnungen können hardwareseitig gleichzeitig durchgeführt werden
[5] Die: Prozessorkern
[6] Peak Performance ist die theoretische Spitzenleistung des Systems
[7] 1 TFLOPS $\widehat{=}$ 10^{12} Gleitkommaoperationen pro Sekunde

2.2.2 JuRoPa/HPC-FF

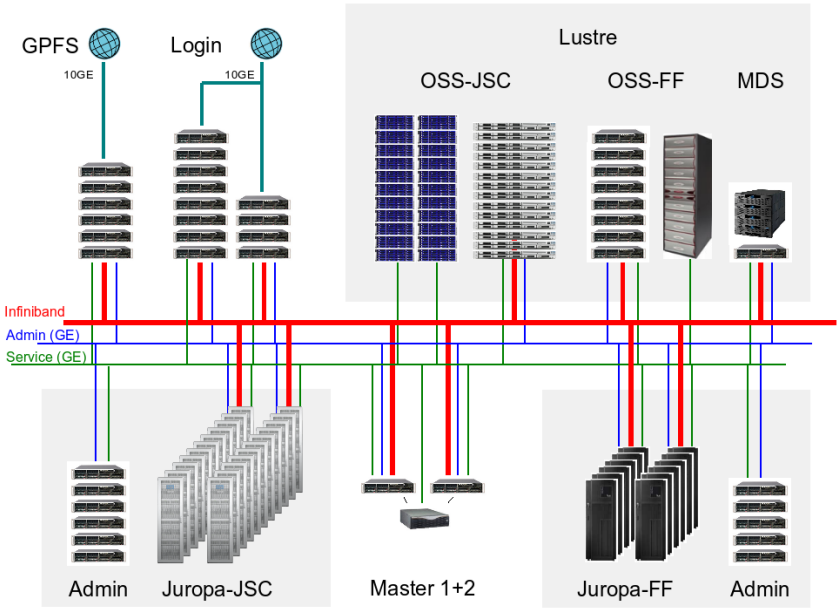


Abbildung 2.3: Architektur des Juropa/HPC-FF Systems[8]

2009 wurde der Cluster JUMP von einem leistungsfähigeren System abgelöst, weshalb die weitere Implementierung, Entwicklung und das Testen des AVS auf dem, im August 2009 in den Produktionsbetrieb gegangenen, Supercomputersystem JuRoPa/HPC-FF (Jülich Research on Petaflop Architectures/High Performance Computing - For Fusion) [20] stattgefunden hat. Das System besteht dabei aus zwei gekoppelten Supercomputern: Juropa und HPC-FF.

2.2.2.1 Juropa-JSC

Juropa-JSC stellt den Nachfolger des JUMP Systems [19] dar, das seinen Betrieb zum Ende des Jahres 2009 eingestellt hat und ist ein Universalrechner, also ein General-Purpose-System. Es besteht aus 2208 Rechenknoten, in denen je 8 Intel Nehalem Prozessoren mit 2.93 GHz Taktfrequenz verbaut sind, die sich 24 GByte DDR3 Hauptspeicher teilen. Die einzelnen Knoten sind durch ein schnelles Verbindungsnetzwerk miteinander gekoppelt. Damit ergibt sich ein Gesamtsystem mit 17664 Prozessoren, was eine Leistung von 207 TFlops Peak Performance und 183.5 TFlops im Linpack Benchmark erreicht.

[8] Quelle: http://www.fz-juelich.de/jsc/datapool/page/4166/architecture.png [8]

2.2.2.2 HPC-FF

HPC-FF ist ein Hochleistungsrechner, der ausschließlich für die Europäische Fusionsforschung zur Verfügung steht. Aufgrund dieses beschränkten Anwenderkreises ist dieser Rechner weniger leistungsfähig als das Juropa-JSC System. HPC-FF besteht lediglich aus 1080 Rechenknoten. Die einzelnen Knoten haben die gleichen Kenndaten wie das Juropa System.
Für das HPC-FF System ergibt sich damit eine Gesamtzahl von 8640 Prozessoren, die 101 TFlops Peak Performance und 87.3 TFlops im Linpack Benchmark erreichen.

2.2.2.3 Gesamtsystem

Werden die beiden System Juropa-JSC und HPC-FF gemeinsam genutzt (s. Abb. 2.3), ergibt sich so ein Gesamtsystem, welches aus 3288 Rechenknoten mit jeweils 8 Prozessoren, also insgesamt 26304 Prozessoren besteht. Diesem stehen 79 TByte Hauptspeicher zur Verfügung und es erreicht eine Peak Performance von 308 TFlops, respektive 274.8 TFlops im Linpack Benchmark.
Das Gesamtsystem ist an einen Storage-Pool angeschlossen, welcher insgesamt 860 TByte Speicherplatz für Benutzerdaten zur Verfügung stellt und auf dem Cluster-Dateisystem Lustre basiert, welches auf schnelle Transferraten bei großen Datenmengen spezialisiert ist. So kann eine aggregierte Bandbreite von ca. 20 GByte/s erreicht werden.
Diese hohe Leistung brachte dem Gesamtsystem Juropa/HPC-FF im Juni 2009 den 10. Platz in der Top 500 Liste der weltweit schnellsten Supercomputer ein.
Auf beiden Systemen kommt der Intel Professional Fortran, C/C++ Compiler zum Einsatz (momentan in der Version 11.0), der bei der Kompilierung von Programmen vom AVS verwendet wird.

2.3 Programmbeschreibung des ERO-Codes

ERO wird am Institut für Energieforschung (IEF-4) des Forschungszentrums Jülich eingesetzt und entwickelt [17]. Es ist eine, größtenteils in C/C++ geschriebene Software, die ursprünglich am Institut für Plasmaphysik in Garching entwickelt wurde [27].
ERO ist ein 3D-Monte-Carlo-Code [24], der die PWW, Transport von Verunreinigungen im Plasma und ihre Lichtemission simuliert. In der Regel wird ein kleiner Bereich eines Fusionsexperimentes simuliert, beispielsweise in der Nähe eines Testlimiters (s. Abb. 2.4). Dieser Bereich wird im Verlauf dieser Arbeit als Simulations-Volumen bezeichnet.
ERO besteht im Prinzip aus zwei Teilen. Es berechnet den Verunreinigungstransport und die Plasma-Wand-Wechselwirkung (PWW). Der Transportteil „verfolgt“ die Bewegung von Testteilchen im Hintergrundplasma unter physikalischen Einflüssen (inklusive Lorentzkraft im elektro-magnetischen Feld, Reibungskraft, Diffusion, atomare Prozesse wie Ionisation und Dissoziation, etc.). Testteilchen repräsentieren dabei eine große Zahl von realen Teilchen.
Trifft ein Testteilchen auf die Wand innerhalb des Simulations-Volumens, wird durch den PWW-Teil „entschieden“, wie viele Atome des verfolgten Testteilchens deponiert oder reflektiert werden und welche weiteren Teilchen chemisch oder physikalisch erodiert werden. Diese erodierten oder reflektierten Spezies werden weiter durch den Transportteil verfolgt.
ERO ist dabei nicht auf die Geometrie eines Tokamak beschränkt, sondern in der Lage beispielsweise auch lineare Plasmaexperimente zu simulieren [6]. Die Simulation arbeitet mit einer Monte-Carlo Annäherung von Testteilchen und verfolgt nur die auftretenden Verunreinigungen. Das Plasma selbst (Dichte, Temperatur) bildet dabei den Simulationshintergrund und wird als Input an das Programm übergeben. In einigen Fällen werden gemessene Plasmaparameter benutzt. Alternativ können von anderen Codes berechnete Werte, beispielsweise von B2-EIRENE [13] für ITER, genutzt werden.

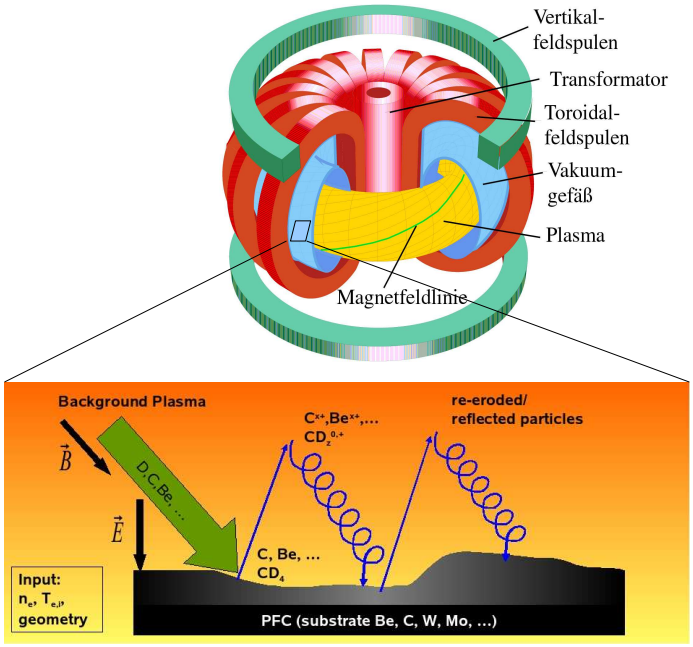
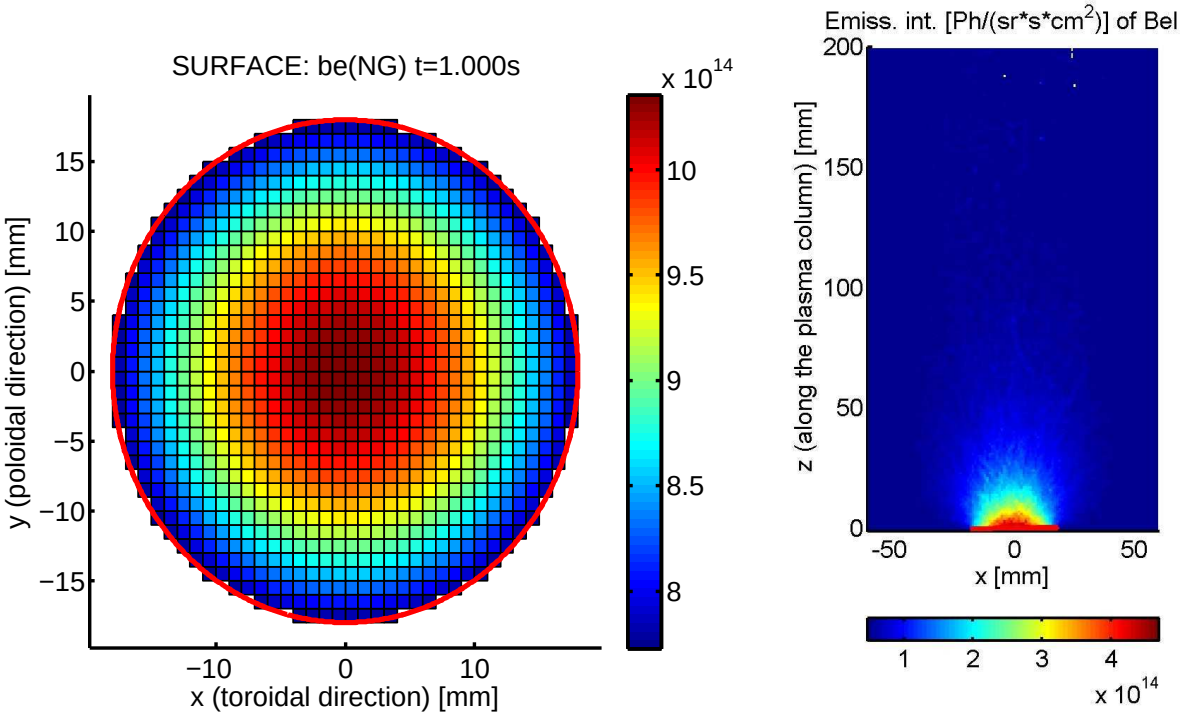


Abbildung 2.4: Beispielhafte Darstellung der physikalischen Prozesse in ERO[9]

Der wichtigsten Teil der Ausgabedaten von ERO sind die Oberflächendaten (Erosion, Deposition, Konzentrationen von Stoffen etc. von verfolgten Elementen in der Plasma-Wand-Interaktionsschicht) und Volumendaten (Dichte der Stoffe, Lichtemission, Anzahl der Ionisationen von Teilchen etc.). In beiden Fällen werden die Daten in einer Gitter-Datenstruktur ausgegeben, die dementsprechend 2 oder 3-Dimensional ist. Jede Zelle des jeweiligen Gitters wird hierbei durch eine komplizierte Datenstruktur charakterisiert, die die oben genannten Daten beinhaltet.
Die so produzierten Daten können mit einem eigens für diesen Zweck entwickelten Matlab-Programm „MERO“ visualisiert werden. MERO hilft dabei, 2D Integrations- oder Querschnittbilder zu erstellen. Ein Beispiel dazu ist in Abbildung 2.5 dargestellt. Die Abbildungen wurden mit MERO erstellt und zeigen eine erodierte Oberfläche, sowie die Lichtemissionen der erodierten Verunreinigungen in einem in y-Richtung integrierten Volumen. Die visualisierten Daten stammen aus einer Simulation des Plasmaexperimentes PISCES-B (vgl. Kap. 3.4.1). Hier sei zunächst nur erwähnt, dass es sich um eine zylindrische Plasmasäule handelt, deren Dichte und Temperaturmaximum im Zentrum der Geometrie liegt. Die Abbildung der Oberfläche zeigt die Dichte der erodierten (**NG**[10]) Beryllium-Teilchen (be) an. Diese ist durch die zuvor erwähnten Eigenschaften der Plasmasäule im Zentrum höher und nimmt zum Rand hin

[9] Quelle: `http://www.dpg-physik.de/dpg/gliederung/fv/p/info/pix/Tokamak.jpg` [5], IEF-4, modifiziert.
[10] entspricht dabei “New Go“, also Teilchen, welche die Oberfläche verlassen

ab. Das nebenstehende Bild zeigt analog dazu die Lichtemission der Beryllium-Teilchen im Simulations-Volumen. Auch hier erkennt man eine höhere Aktivität im Zentrum der Geometrie.



(a) Oberflächendaten (b) Spektroskopie-Volumendaten

Abbildung 2.5: Oberfläche- und Volumendaten

ERO wird unter anderem verwendet, um Experimente an TEXTOR [29], JET [11] und ITER [23] nachzuvollziehen 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 müssen, wie beispielsweise die Implementierung von weiteren physikalischen Prozessen, um einen möglichst realistischen Simulationsablauf zu erhalten. Es existieren mehrere Experimente zur Validierung der in ERO eingesetzten Modelle, Daten und Annahmen. Die Implementierung von weiteren physikalischen Effekten soll dabei möglichst nur in einem gewissen Rahmen die Ausgabedaten von ERO beeinflussen und sich größtenteils mit den

Ausgabedaten aus vorherigen Läufen decken. Während die Implementierung meistens nur in einem gewissen Teil des Codes stattfindet und zur verbesserten Vorhersage von Ergebnissen in diesen Code-Teilen führen soll, sollen sich die Ausgabedaten von anderen Teilen des Codes nicht wesentlich von früheren Läufen unterscheiden. Es sei denn, diese Unterschiede sind gewünscht und die neue Implementation führt zu einem neuen Ergebnis. In einem solchen Fall ist eine Anpassung der Referenzdaten, die zur automatischen Validierung genutzt werden, erforderlich. Schließlich soll ERO zur vorhersagenden Modellierung von zukünftigen Experimenten wie ITER verwendet werden.

2.3.1 Eingabedaten

Die Simulation von ERO ist in sieben Fälle unterteilt, die den unterschiedlichen Szenarien bei der Simulation des Transports von lokalen Verunreinigungen und der Wandwechselwirkung entsprechen (s. Tabelle 2.1). Obwohl alle diese Prozesse, wie die physikalische und chemische Erosion, offensichtlich gleichzeitig stattfinden, werden sie in ERO separat berechnet. Ein typischer ERO-Lauf besteht jedoch aus mehreren Zeitschritten, in denen jedes Mal die plasmabegrenzende Oberfläche modifiziert wird. Ist der Zeitschritt klein genug gewählt, simuliert ERO iterativ das Zusammenspiel der Prozesse korrekt. Jedes dieser Szenarien kann dabei, je

Fall	Beschreibung
0	Physikalische Erosion
1	Chemische Erosion
2	Erosion durch Sauerstoff
3	Deposition von Verunreinigungen aus dem Plasma
4	Transport von extern eingeführten Teilchen
5	Transport von chemisch erodierten Teilchen
6	Transport von physikalisch erodierten Teilchen

Tabelle 2.1: Simulationsfälle in ERO

nach Bedürfnis, wahlweise an- oder abgeschaltet werden, was bei der automatischen Validierung berücksichtigt werden soll. So liegt der Fokus bei der Validierung von simulierten PWW-Prozessen im Wesentlichen in der Überprüfung der erhaltenen Oberflächen-Ausgabedaten, während er bei den Transport-Szenarien in der Untersuchung der Simulations-Volumendaten liegt.
Typischerweise benutzt ERO zur Simulation von PWW-Prozessen das „Homogene Material-Mischungs-Modell“ (HMM). Dies bedeutet, dass für eine Mischung von beispielsweise Kohlenstoff und Wolfram vorkalkulierte Daten von SDTrimSP [10] für reinen Kohlenstoff und reines Wolfram, gemittelt proportional zu den jeweiligen Konzentrationen, benutzt werden.
Eine Weiterentwicklung bei der Simulation der PWW-Prozesse in ERO ist die Möglichkeit,

den C++/C - ERO- Code mit dem Fortran-Code SDTrimSP direkt zu koppeln. SDTrimSP ist dabei für die Simulation des Oberflächenmaterials zuständig und behandelt Schwerpunkte wie Eindringtiefe der Teilchen, Erosion, Materialdurchmischung und physikalische, sowie chemische Zerstäubung von Teilchen aus der Oberfläche. Dazu werden Daten zwischen den beteiligten Codes ausgetauscht. Diese Code-Kopplung soll zur besseren Vorhersage von Plasma-Oberflächen-Interaktionen in Fusionsexperimenten führen.
Als Startparameter dient ERO eine Datei, die alle für die Simulation relevanten Daten enthält. 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 2.1)
- 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.

2.3.2 Programmablauf

Die Diskretisierung in Einzelzeitschritte 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 dieses Zeitschrittes genutzt.
Dieser Vorgang wird auf JUMP unter Nutzung des Batch-Schedulings des IBM LoadLevelers [22] 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 2.6).

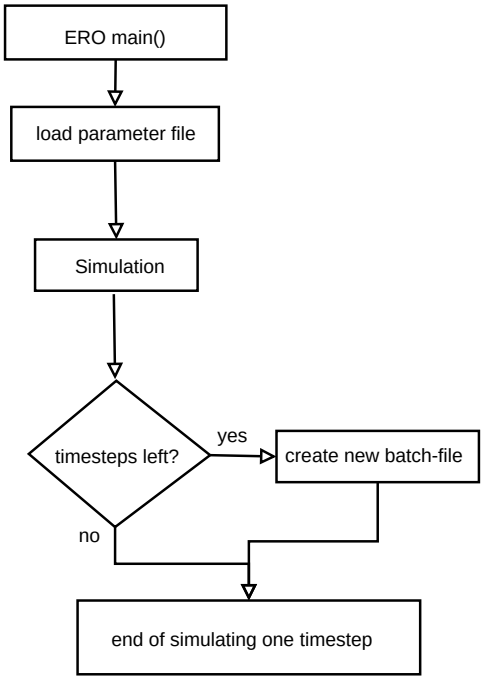


Abbildung 2.6: Serieller Programmablauf von ERO

Unter Juropa/HPC-FF steht ein ähnlicher Mechanismus zur Verfügung, der jedoch ohne die Nutzung mehrerer Batchdateien auskommt und die Verkettung der Aufrufe von ERO über, voneinander abhängige, Batch-Jobs realisiert.

2.4 Besonderheiten der Monte-Carlo-Simulation

2.4.1 Das Monte-Carlo-Modell

Die MC-Simulation ist ein häufig eingesetztes Rechenmodell in der Plasmaphysik, um Vorgänge zu simulieren, die „zufällig“ ablaufen sollen [26]. Dabei werden Testteilchen generiert, deren Eigenschaften einer vorher festgelegten statistischen Wahrscheinlichkeitsverteilung gehorchen. Es könnte aber 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 großen Zahlen“ vollkommen aus (Ausdruck 2.3).

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

$$P(\lim_{n\to\infty} \overline{X}_n = \mu) = 1 \tag{2.3}$$

mit

$$\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i \tag{2.4}$$

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 ihr empirischer Mittelwert 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 (2.3) wird aber für eine genügend große Anzahl an Monte-Carlo-Teilchen garantiert, so dass diese die realen Teilchen mit hinreichender Genauigkeit repräsentieren können.

2.4.2 Deterministische und nicht-deterministische Simulation

In der 3D Simulation ERO werden Testteilchen generiert, die eine repräsentative Untermenge der in der Realität auftretenden Teilchen bilden. Dabei gibt es einen bestimmten, vorher festgelegten Eingabeparameter, der angibt, wie viele reale Teilchen ein Monte-Carlo Testteilchen repräsentieren. Die Teilcheneigenschaften und andere Werte, wie der Winkel, unter dem die Teilchen beispielsweise von einer Wand eines Fusionsexperimentes reflektiert werden, lassen sich anhand von generierten Pseudo-Zufallszahlen berechnen, oder anders gesagt: Die mit einer entsprechenden Verteilung generierten Zufallszahlen werden auf die Teilcheneigenschaften abgebildet. Dazu ist es notwendig einen Startwert für die Folge der generierten Zufallszahlen zu bestimmen, von dem die erzeugten Zahlen in gewisser Weise abhängen. Dieser Startwert wird auch „Random Seed“ genannt. In ERO bedeutet ein Random Seed $\neq 0$, dass die angegebene Zahl als Startwert zur Generierung der Zufallszahlen herangezogen wird, wohingegen ein Random Seed von 0 die Erzeugung eines pseudo-zufälligen zeitabhängigen Random Seeds zur Folge hat. In der Praxis bedeutet dies, bei Verwendung eines gleichen Random

Seeds auch eine Liste gleicher Zufallszahlen zu erhalten. Dieser Sachverhalt ist bedingt durch den in ERO integrierten deterministischen, rekurrenten Zufallszahlengenerator. Dieser arbeitet ohne Entropie. Fehlende Entropie bedeutet hier, dass zur Berechnung der Zufallszahlen lediglich mathematische Modelle angewandt und keine äußeren Einflüsse wie Benutzereingaben oder Netzwerkverkehr hinzugezogen werden, weshalb die so generierten Werte auch lediglich „Pseudo"-Zufallszahlen genannt werden. In diesem Fall nennt man die Erzeugung der Zufallszahlen „deterministisch"
Bei der Nutzung von unterschiedlichen Random-Seeds in einer Simulation ergibt sich das Problem, die Ausgabedaten nicht exakt vergleichen zu können. Da hier unterschiedliche Pseudo-Zufallszahlen generiert werden, ist der Simulationsverlauf in einem gewissen Rahmen anders als bei der Nutzung eines gleichen Random Seeds. In diesem Fall ist die Erzeugung der Zufallszahlen nicht deterministisch. Fließt beispielsweise eine Zufallszahl in den Reflexionswinkel eines Testteilchens ein, wird das Testteilchen bei unterschiedlichen Zufallszahlen anders reflektiert. Dies schlägt sich dann wiederum im Verlauf der Trajektorie des Teilchens nieder.
Für die automatische Validierung bedeutet dieser Umstand, die Ausgabedaten unter Nutzung verschiedener Random Seeds nur bedingt vergleichen zu können. Die Anzahl der beobachteten Testteilchen, sowie die Verunreinigungsdichte im gesamten Simulationsvolumen als auch die Teilchenbelegung auf der Oberfläche, darf sich in der Summe nur geringfügig unterscheiden. Dieser Rahmen hängt maßgeblich mit der Anzahl der Testteilchen zusammen, die simuliert werden. Er wird anhand eines relativen Fehlers festgelegt, mit dem sich die zu validierenden Daten mit den Referenzdaten, die aus früheren Läufen erhalten wurden, unterscheiden dürfen. Da die Simulation aufgrund der Vielzahl an simulierten physikalischen Effekten eine hohe Komplexität aufweist ist der zu definierende relative Fehler lediglich durch Erfahrungswerte zu begründen. Eine möglicherweise existierende Berechenbarkeit wird daher in dieser Arbeit nicht behandelt. Für alle Validierungen wurde ein maximaler relativer Fehler von 10^{-2} zugrunde gelegt.

2.5 OpenMP

Die in ERO implementierte, OpenMP-basierte, Parallelisierung [18] ist ein Weg die Vorteile von symmetrischen Multiprozessorsystemen zu nutzen. Symmetrisch bedeutet dabei, dass alle Prozessoren auf einem gemeinsamen Hauptspeicher arbeiten und daher auf die gleichen Daten zugreifen können. Dies hat 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*) [31] 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.
Die Parallelisierung von ERO basiert auf der Verteilung von Testteilchen in Untermengen, genannt „chunks“, die gleichzeitig und unabhängig voneinander verfolgt werden können. Um die Lastverteilung der chunks 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[11] `chunksize`
  - Zuweisung im „Round-Robin-Verfahren“ [12]
  - Fehlt chunksize → Aufteilung in $\frac{N}{P}$ Stücke
    (N: Problemgrösse, P: Anzahl der Threads)

[11] Ist im Fall von ERO entweder die Anzahl der Zellen pro Thread von denen Teilchen starten oder, bei externer Injektion, die Anzahl der Testteilchen pro Thread
[12] 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)

Das Scheduling wird hierbei maßgeblich von der chunksize beeinflusst. Diese gibt die Größe der Stücke an, die auf die beteiligten Threads verteilt werden. Im einfachsten Fall entspricht die Größe der Stücke dem festen Quotienten

$$chunksize = \frac{Arbeitslast}{\#Threads} \tag{2.5}$$

Allerdings macht derart dynamisches Scheduling keinen Sinn, da es genauso viele Stücke wie beteiligte Threads gibt. Daher ist es notwendig, den Quotienten abzuändern in

$$chunksize = \frac{Arbeitslast}{\#Threads \cdot \#Factor} \tag{2.6}$$

wodurch die Anzahl der chunks $\#Threads \cdot \#Factor$ beträgt. Damit stehen mehr Stücke als Threads bereit und diese können anschließend dynamisch auf die Threads verteilt werden.

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. Die Auswahl der Scheduling-Verfahrens ist eine Optimierungsfrage. Bei mehreren Testläufen von ERO im Rahmen der Optimierung von Case 4 (vgl. Tabelle 2.1) [18] wurde festgestellt, dass eine chunksize von $\frac{\#Testteilchen}{\#Threads \cdot 10}$ für diesen Fall ein Optimum darstellt.
OpenMP Programme können meist ohne große Veränderungen auf einem, wie auch auf mehreren Prozessoren ablaufen.

Kapitel 3

Das automatische Validierungssystem

Um die in Kap. 1.2 definierten Ziele eines automatischen Validierungssystems erfüllen zu können, wurde ein schon vorhandenes System zum Benchmarking von parallelen Anwendungen verwendet: das Jülich Benchmarking Environment [1] - „JuBE“.

3.1 JuBE

JuBE ist in der Skriptsprache PERL (Practical Extraction and Report Language) geschrieben, welche durch ihre hohe Funktionalität im Bereich der Textverarbeitung auffällt. Reguläre Ausdrücke ermöglichen ein hohes Maß an Komplexität im Bereich der Analyse und Substitution von Textdaten.
Die Konfiguration von JuBE geschieht mit Hilfe von XML-Dateien [32], welche von JuBE ausgelesen und interpretiert werden. XML (e**X**tensible **M**arkup **L**anguage) eignet sich hierbei besonders, da es einen übersichtlichen Weg der Konfiguration bietet. Des weiteren kann es aufgrund seiner ASCII-Basiertheit leicht von PERL Programmen verarbeitet werden. JuBE benutzt zudem eine in PERL geschriebene Bibliothek (XML-Simple [25]), welche die Handhabung von XML-Dateien besonders vereinfacht.

3.1.1 Aufgaben von JuBE

JuBE ist fähig, die 4 Hauptaufgaben im Bereich des Benchmarking zu erfüllen, welche im Folgenden aufgeführt werden. In Klammern dahinter ist jeweils die dafür zuständige XML-Konfigurationsdatei angegeben.

- Übersetzen des Programmcodes in ein ausführbares Programm (*compile.xml*)
- Aufbereitung der Eingabedaten und das Kopieren von benötigten Dateien (*prepare.xml*)
- Ausführung des Programms (*execute.xml*)
- Analyse der Ausgabedaten durch externe Programme (*analyse.xml* & *verify.xml*)

3.1.1.1 Das Top-Level XML File

JuBE wird mit dem Top-Level XML File der Konfiguration gestartet. Das Wurzel-Element *<bench> ... </bench>* enthält die Definition zu einer Sammlung von Benchmarks, die jeweils mit *<benchmark> ... </benchmark>* eingeleitet werden. Diese können wahlweise aktiviert oder deaktiviert und so bei einem Lauf von JuBE ausgeführt werden oder nicht. Im Folgenden wird die Struktur dieser Datei am Beispiel der Konfiguration des Validierungssystems für ERO erläutert:

- **<compile cname="PISCES"version="new"/>**

  Hier wird die Aufgabe zur Kompilierung des Programms definiert, die über einen gemeinsamen Namen (cname) in *compile.xml* referenziert wird. Diese Zeile zeigt JuBE an, dass es das Ziel PISCES in *compile.xml* kompilieren und die darin enthaltenen Anweisungen ausführen soll. Erläuterungen dazu werden im entsprechenden Abschnitt 3.1.1.2 gegeben.

- **<tasks threadspertask="1"taskspernode="1"nodes="1"/>**

  Dieses Tag definiert, mit wie vielen Knoten, Prozessoren und Threads das kompilierte Programm ausgeführt werden soll. Hier sind jeweils mehrere Werte zulässig, die bedingen, dass aufgrund dieser Werte ein minimal spannender Baum erstellt wird und das Programm jeweils mit einer Kombination dieser Werte ausgeführt wird. Beispielsweise würde folgende Konfiguration zu einer Ausführung des Programms mit 2, 4, 8 und 16 Threads auf einem Knoten führen:

  <tasks threadspertask="2,4,8,16"taskspernode="1"nodes="1"/>

  Dies ist von großem Nutzen, falls man den Zeitverbrauch messen, also ein Time-Benchmarking durchführen will. Definiert man im Analyse und Verfikationsschritt entsprechende Optionen, erhält man so eine übersichtliche Tabelle, die zu jeder Anzahl von genutzten Prozessen/Threads einen Zeitwert ermittelt, aus dem man die Performancesteigerung des Programms errechnen kann.

- **<params eroroot="/home6/jzam04/jzam0410/ero"... />**

  In diesem Tag werden selbst definierte Parameter eingetragen, die beispielsweise später bei der Substitution genutzt werden können. Sie sind global in allen nachfolgenden Schritten verfügbar und können so dazu eingesetzt werden, den Verlauf des Benchmarks zu gestalten. Im Beispiel wird der Pfad zum ERO-Hauptverzeichnis gesetzt, um später auf das ERO Testing Tool zugreifen zu können, welches sich ebenfalls dort in einem Unterordner befindet. Eine Auflistung aller Parameter gibt Tabelle 3.1

- **<prepare cname="PISCES"/>**

  Das prepare-Tag greift wieder über einen gemeinsamen Namen auf ein Element zu, welches in *prepare.xml* definiert ist und in diesem Fall so heißt wie der zugehörige Benchmark: PISCES. Erläuterungen hierzu befinden sich im dem entsprechenden Abschnitt zu *prepare.xml*.

- **<execution iteration="1"cname="$platform"/>**

  Hiermit wird JuBE mitgeteilt, auf welcher Platform und wie oft JuBE das Programm ausführen soll. Die $platform - Variable muss im Root-Tag <bench /> definiert sein. Anhand dieses Namens sucht sich JuBE ein entsprechendes Muster einer Batch-Datei, über die es das Programm auf dem genutzten Rechner startet. JuBE stellt dabei eine Vielzahl an Batchdatei-Mustern für unterschiedliche Supercomputer-Systeme zur Verfügung (vgl. Kap. 2.2), so dass man nur die Platform definieren und sich nicht um die Definition eines neuen Musters für jeden Supercomputer kümmern muss. In diesem Fall heißt die zugehörige Platform *Intel-Nehalem-Juropa*.

- **<verify cname="ERO"/>**

  Durch das verify-tag wird JuBE angeleitet, den in *verify.xml* definierten Verifikationsschritt auszuführen. Dies beinhaltet hauptsächlich den Aufruf des entsprechenden Validierungs-Tools, welches einzigartig für jede Simulation sein muss, und wird im Abschnitt zur Verifikation ausgeführt.

- **<analyse cname="$platform"/>**

  Hier steht der Name des Analyseschritts, der in *analyse.xml* definiert wurde. Auch hier wird eine genaue Erläuterung im entsprechenden Abschnitt gegeben. Es werden hierbei die Ausgabedaten der Simulation und des Validierungstools eingelesen und verarbeitet.

Damit sind in der Top-Level XML-Datei alle, für das automatische Testen notwendigen, Ziele definiert. JuBE wird mit dieser Datei als Programmargument gestartet und beginnt daraufhin mit der Verarbeitung. Für jede einzelne Simulation wird eine Sandbox-Umgebung [1] erstellt.

[1] Ein Verzeichnis, in dem alle benötigten Dateien liegen und aus dem heraus das zu testende Programm keinen Schaden anrichten kann

Parametername	Beschreibung
$eroroot	Hauptverzeichnis, in dem sich ERO, sowie ERO.ETT befinden
$parfile	Parameterfile für das jeweilige Szenario
$parts	Anzahl der extern eingeblasenen Testteilchen
$partspercell	Anzahl der zu erodierenden Testteilchen pro Zelle
$totaltime	Gesamtzeit, die simuliert werden soll
$timeperstep	Zeit pro Zeitschritt
$error	Relativer Fehler für den Vergleich
$mailaddress	Mailadresse, an die der Testreport geschickt werden soll
$etttestname	Name zur Identifizierung des jeweiligen Szenarios
$sputterdat	Name der zu benutzenden Zerstäubungsdatenbank
$taskname	Name zur Identifizierung des jeweiligen Tests
$target	Target im Makefile, welches kompiliert werden soll

Tabelle 3.1: Globale Parameter

3.1.1.2 Kompilierung

JuBE kann schon in der Kompilierungsphase Textdateien wie *makefiles*, die zur automatischen Erstellung von Programmcode verwendet werden oder gar ganze Code-Dateien, in denen die Eingabedaten direkt codiert sind, einlesen und verarbeiten. Die hierzu notwendige Vorgehensweise wird in der Datei *compile.xml* definiert. Im Falle der *makefiles* lässt sich beispielsweise der, von der genutzten Platform verwendete, *compiler* substituieren, oder aber es lassen sich bestimmte Präprozessormakros setzen, die eine plattformabhängige Übersetzung bedingen. Dazu können in der dafür zuständigen Konfigurationsdatei *compile.xml* Substitutionsvariablen definiert werden, die dann automatisch beim Parsen der jeweiligen Datei ersetzt werden:

Listing 3.1: Substitution im *makefile*

```
<src directory="src" files="*" />
 <substitute infile="makefile" outfile="make_ero">
   <sub from="#LIBS#"        to="-lm" />
   <sub from="#EXECNAME#"    to="$execname" />
   <sub from="#MYDEF#"       to="$mydef" />
   </substitute>
</src>
```

Im Listing 3.1 wird definiert, dass alle Dateien (“*“) aus dem Verzeichnis *src* in das jeweilige Verzeichnis kopiert werden sollen, in dem die Kompilierung stattfindet. Dort soll die Datei *makefile* geöffnet und die zu ladenden Bibliotheken (`#LIBS#`), sowie der Name der ausführbaren Datei (`#EXECNAME#`) und eigene Definitionen (`#MYDF#`) ersetzt werden. Nach diesen Ersetzungen wird die eigentliche Kompilierung gestartet und anschließend die ausführbare Datei

in das Ausführungsverzeichnis kopiert. Um den Kompilationsprozess nicht zu beeinflussen, wird zunächst eine Kopie des Quellen-Verzeichnisses in der Sandbox-Umgebung angelegt.

3.1.1.3 Vorbereitung

In der Vorbereitungsphase von JuBE werden alle zur Ausführung des Programms benötigten Maßnahmen getroffen, welche in der Datei *prepare.xml* konfiguriert sind. Dies beinhaltet hauptsächlich das Kopieren von Konfigurations- und Eingabe-Dateien, bei dem ebenfalls Substitutionsvariablen zur Modifizierung genutzt werden können.
Für ERO wird hier beispielsweise die Eingabe-Parameter-Datei modifiziert:

Listing 3.2: Substitution in der Parameterdatei

```
<substitute infile="par_psc.in" outfile="par_psc">
     <sub from="#PARTS#" to="$parts" />
     <sub from="#PARTSPERCELL#"  to="$partspercell" />
     <sub from="#TOTALTIME#" to="$totaltime" />
     <sub from="#TIMEPERSTEP#" to="$timeperstep" />
</substitute>
```

Die Anzahl der zu simulierenden Testteilchen, die totale Simulationszeit und die Simulationszeit pro Zeitschritt werden hier an den entsprechenden Stellen ersetzt, die durch die Platzhalter (#PARTS#, #TOTALTIME#, #TIMEPERSTEP#) gekennzeichnet sind. Alle Dateien, die hier angegeben und verarbeitet werden, kopiert JuBE anschließend in das Verzeichnis, in dem der Lauf der Simulation stattfinden soll.

3.1.1.4 Ausführung

Die Konfigurationsdatei *execute.xml* legt fest, welche Werte in der zur Platform gehörenden Batch-Datei ersetzt werden. Diese Batchdatei ist ein Template, in dem alle für die Plattform wichtigen Platzhalter eingetragen sind und anschließend substituiert werden können. Im Fall von Juropa heißt dieses Template intel_nehalem_MSUB.in. Es werden die notwendige Anzahl der Knoten, Prozessoren und Threads substituiert, sowie der Name der Ausführbaren Datei, welche über die Batch-Datei gestartet wird. Umgebungsvariablen, die für den Lauf notwendig sind, können ebenfalls gesetzt werden. Nach Abarbeitung dieser Datei wird die Batch-Datei an die Warteschlange der Platform übergeben und bei Verfügbarkeit von ausreichenden Ressourcen das Programm so gestartet.

3.1.1.5 Analyse

Im Verfikationsschritt, welcher in der Datei *verify.xml* definiert ist, werden die Ausgabedaten des Programms validiert. Dies geschieht durch Aufruf eines externen Programms, was in diesem Fall das ERO Testing Tool - „ERO.ETT“ - ist. Auf das ERO Testing Tool und seine Funktions- und Arbeitsweise wird in späteren Abschnitten dieser Arbeit eingegangen. Es ist jedoch notwendig anzumerken, dass dieses Tool eine Ausgabedatei erzeugt, welche von JuBE verarbeitet werden kann. Dies geschieht mit Hilfe von regulären Ausdrücken, nach denen die Ausgabedatei durchsucht und die darin enthaltenen Informationen abgespeichert werden.
Die regulären Ausdrucke werden in der Konfigurationsdatei *patterns.xml* definiert. An dieser Stelle werden den gefundenen Werten auch Namen zugeordnet, um später auf sie zugreifen zu können. Dies beinhaltet, am Beispiel von ERO, die benötigte Rechenzeit, die Anzahl der simulierten Testteilchen und eine Information, ob die Validierung erfolgreich bestanden wurde oder nicht. Diese Werte werden nach Abschluss der Verifikation und Validierung, gesteuert durch die Datei *analyse.xml*, übersichtlich nach Benchmarks, in diesem Fall Szenarien, geordnet in einer Tabelle wiedergegeben, die eine kurze Übersicht über den gesamten Validierungsprozess liefert.

3.1.2 Gesamtsystem

In Abbildung 3.1 ist die Vorgehensweise des AVS systematisch dargestellt. Der erste Teil der Validierung wird von JuBE übernommen. Die Sandbox-Umgebung wird erstellt, das Programm wird kompiliert, Eingabedaten werden vorbereitet und schließlich wird das Programm auf dem genutzten Rechnersystem ausgeführt. Die erzeugten Ausgabedaten werden an ERO.ETT weitergeleitet, wo sie auf charakteristische Werte reduziert und mit den Referenzdaten verglichen werden. Die Ausgabe von ERO.ETT, sowie die des zu validierenden Programms wird wiederum von JuBE im Analyseschritt eingelesen und in einer übersichtlichen Tabelle zusammengefasst, die ausgegeben wird.

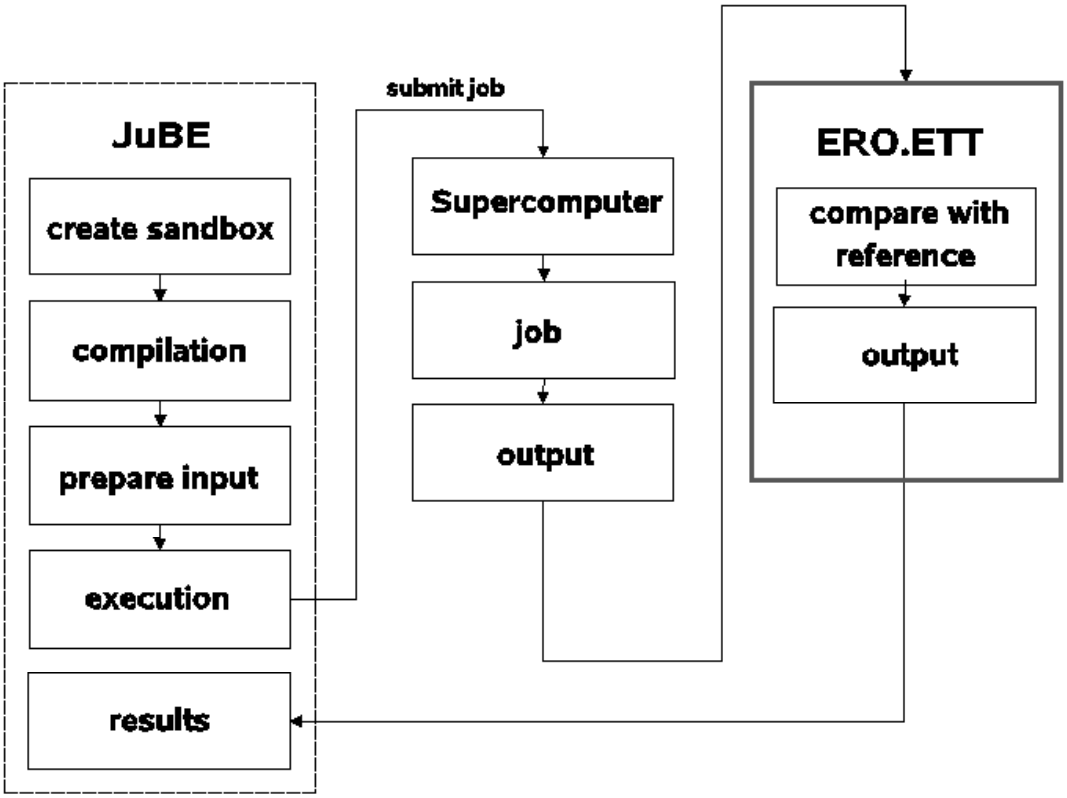


Abbildung 3.1: Schematische Darstellung des automatischen Validierungssystems

3.2 ERO Testing Tool

Das ERO Testing Tool ist eine speziell auf die 3D Monte-Carlo-Simulation ERO zugeschnittene Validierungsanwendung, die es ermöglicht, zu überprüfen, ob die Ausgabedaten von ERO reproduzierbar sind oder nicht.
Dazu benötigt es eine Konfigurationsdatei sowie 2 Ausgabeverzeichnisse von ERO, die es auf Stimmigkeit überprüfen soll. Das erste Ausgabeverzeichnis dient dabei als Referenz, während das zweite die Probe darstellt, die mit der Referenz verglichen werden soll. ERO.ETT ist momentan in der Lage, die beiden für die Simulation relevantesten Daten einzulesen und zu vergleichen. Dabei handelt es sich zum einen um Volumendaten und zum anderen um Oberflächendaten. ERO.ETT wird wie folgt gestartet:

ERO.ETT.prg <Konfigurationsdatei> <Referenz-Verzeichnis> <Probenverzeichnis>

3.2.1 Konfiguration

Die Konfigurationsdatei von ERO.ETT ist in einem Matlab ähnlichen Format gehalten und orientiert sich von der Syntax an den Parameterdateien von ERO. Dadurch ist eine einheitliche Handhabung gewährleistet, ohne die Nutzer von ERO und ERO.ETT mit einem neuen Format zu konfrontieren. Es können folgende Optionen gesetzt werden.

- Name des Testlaufs
- Ausgabeverzeichnis für die Zusammenfassung der Testergebnisse
- E-Mail Adresse
- Erlaubter maximaler relativer Fehler
- Funktionalität

Listing 3.3: config.ett

```
% ====================================
% An example of ETT configuration file
% ====================================
%
% name of the test, modifys name of output file
name = 'PISCES';

%output path, where ERO.ETT should write the testresults to
output = '/home/galonska/TEST';

%mail address to send the testresults
mail = 'a.galonska@fz-juelich.de';

% define the relative error to be used, default is 10E-4
relative_error=10E-3;

%comparing sum of dens all elements
CmpSum('all', 'dens');

%comparing surface data of all elements
CmpSurf('all');

%comparing the avg penetration depth
%params: element, charge, plane, dens or spec, start_x, start_y, width,
     length, angle, sections
CmpAvgPenDep('be__', 0., 'xy', 'dens', 0., 0., 20.0, 100.0, -90.1, 10.)
    ;


%comparing spec average for element be
%CmpAver('be' , 'spec');

%comparing spec average for element _c
%CmpAver('_c' , 'spec');
```

In Listing 3.3 ist eine Beispielkonfiguration angegeben. Die Methoden im unteren Teil der Datei sind in ERO.ETT fest registrierte Methoden, können aber erweitert werden, so dass eine Flexibilität in Hinsicht der Integration neuer Funktionalität gewährleistet ist. ERO.ETT erkennt anhand der Methoden-Signaturen, welche Funktion es intern aufrufen soll. So reduziert sich der Arbeitsaufwand bei der Integration neuer Vergleichsmethoden auf ein Minimum. Man implementiert lediglich eine neue Methode und macht die Signatur, welche in der Konfigurationsdatei dafür verwendet werden soll, in ERO.ETT bekannt. So lassen sich auch die Parameter der jeweiligen Methode auslesen. Dies sorgt für einen geringen Arbeitsaufwand bei der weiteren Entwicklung der Software. Momentan sind folgende Methoden für den Vergleich von Ausgabedaten in ERO.ETT implementiert:

Funktion	**Argumente**
compareEmissionSum	Element Ladung Dichte oder Spektroskopie
compareSurface	Element Ladung
compareAvgPenetrationDepth	Element Ladung Ebene Dichte oder Spektroskopie Element, dessen Eindringtiefe berechnet werden soll Ladung des Elements Ebene, in der die Eindringtiefe berechnet werden soll Dichte oder Spektroskopie Startpunkt der Box Breite der Box Länge der Box Winkel Anzahl der Abschnitte

Tabelle 3.2: Vergleichsmethoden des ERO Testing Tools

3.2.2 Behandlung von Volumen-Daten

Der Simulationsbereich ist in ERO mit einem 3D-Gitter bedeckt. ERO registriert, wie viele Zeitschritte jedes der verfolgten MC-Testteilchen in einer Zelle verblieben ist. Daraus berechnet ERO die Dichte ρ. Sind die Plasmaparameter n_e (Elektronendichte) und T_e (Elektronentemperatur) bekannt, lässt sich daraus für jede Zelle die entsprechende Lichtemission berechnen:

$$I \propto \langle v\sigma \rangle (n_e, T_e) \cdot n_e \cdot \rho \cdot \Delta V \tag{3.1}$$

mit

$\langle v\sigma \rangle (n_e, T_e)$: Ratenkoeffizient (z.B.: aus ADAS-Datenbank [16])
n_e: Elektronendichte
T_e: Elektronentemperatur
ρ: Verunreinigungsdichte
ΔV: Volumen

Die Messung der Lichtemission von Teilchen nennt man auch Spektroskopie, weshalb die so simulierten und berechneten Daten Spektroskopiedaten genannt werden.
Bedingt durch seine Monte-Carlo-Basiertheit, produziert ERO eine große Menge unterschiedlicher Ausgabedaten, die statistisch verteilt sind. Beispielsweise kann sich die Teilchendichte in einer Zelle, rein zufällig, sehr von den Referenzdaten unterscheiden. Zur Beurteilung der Reproduzierbarkeit der Daten von 2 Versionen des Codes ist es daher notwendig, die Menge der Daten auf „charakteristische Werte“ zu reduzieren, die sich vergleichen lassen.
Die Gesamtzahl der Teilchen im Simulationsvolumen (Integral der Teilchendichte) muss im Rahmen des statistischen Fehlers der Monte-Carlo Simulation, also $\frac{1}{\sqrt{N}}$, übereinstimmen (mit N: Teilchenanzahl).

3.2.2.1 Charakteristische Werte der Volumen-Daten

In Abbildung 3.2 ist eine Verteilung von Verunreinigungen in einer Ebene exemplarisch dargestellt. Es gibt Zellen mit und ohne Verunreinigungen. Die roten Punkte sollen Testteilchen innerhalb des Simulations-Volumens darstellen. Aus dieser Verteilung lassen sich die Volumen-Daten berechnen, welche im Fall von ERO, aus Spektroskopie und reinen Dichtedaten bestehen. Diese werden aus historischen Gründen in ERO Emissionsdaten genannt.

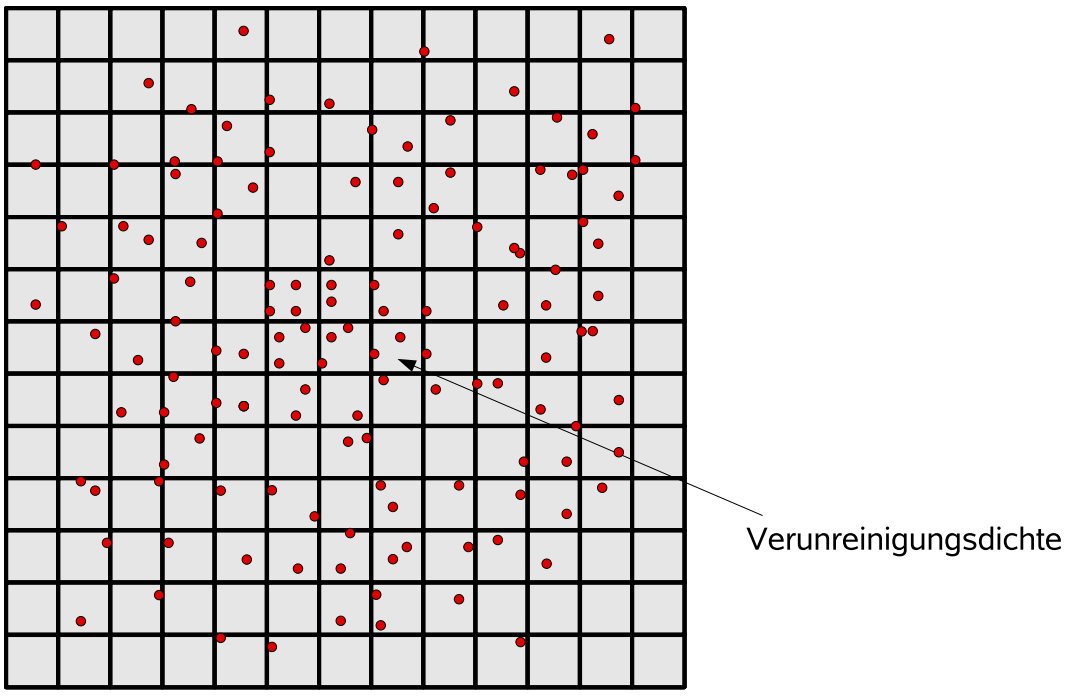


Abbildung 3.2: Exemplarische Verunreinigungsdichte im Gitter - 2D Repräsentation

Diese Emissionsausgabedaten werden in ERO.ETT mit der Methode **CmpEmissionSum** behandelt. Als erstes Argument verlangt diese Methode das zu vergleichende Element sowie dessen Ladung, da in ERO Emissionsdaten elementweise abgespeichert werden. Hier wird der Einfachheit halber der in ERO genutzte Name, beispielsweise **_be** für Beryllium, und dessen Ladung angegeben. Das Schlüsselwort **all** steht dabei für alle Elemente. Als dritten Parameter erwartet es entweder die Schlüsselwörter **spec** oder **dens** und wird damit angewiesen Spektroskopiedaten oder reine Dichtedaten zu vergleichen. Die Vorgehensweise der Methode bleibt jedoch unabhängig vom Element gleich:
Zunächst wird die generelle Konsistenz der Daten überprüft. Globale Daten wie die Struktur des Gitternetzes, die Anzahl der beobachteten Elemente, sowie der berechnete Zeitschritt müssen exakt übereinstimmen. Andernfalls ergibt die Validierung der restlichen Daten keinen Sinn. Somit werden unnötige Berechnungen vermieden. ERO rechnet intern in einem 3D-Volumen. Um die große Menge an Ausgabedaten zu reduzieren, werden diese aber meistens vorher in eine Richtung integriert und jeweils die Zellen in der **xy**-, der **xz**- und der **yz**-Ebene ausgegeben. Dies ist insofern sinnvoll, da man in realen Experimenten ebenfalls ein entlang einer Beobachtungslinie integriertes 2D-Bild betrachtet. Registriert ERO, dass sich während der Simulation eines Zeitschrittes N verfolgte Testteilchen im kartesischen Koordinatensystem in Zelle (i,j,k) aufgehalten haben, wird angenommen, dass die Dichte ρ in dieser Zelle

$$\rho = \frac{N}{dV} = \frac{N}{dxdydz} \tag{3.2}$$

beträgt. Durch Integration in einer Richtung, beispielsweise z, kann man die 3D Dichte ρ auf eine 2D „Flächendichte“ σ reduzieren:

$$\sigma_{xy} = \int \rho dz = \sum_k \rho_{(i,j,k)} \tag{3.3}$$

Damit gilt folgendes:

$$N = \int \rho dV = \int \sigma_{xy} dxdy \tag{3.4}$$

Der charakteristische Wert, beispielsweise für **xy**-Ebene, ist damit

$$\sum_i \sum_j \sigma_{(i,j)} \tag{3.5}$$

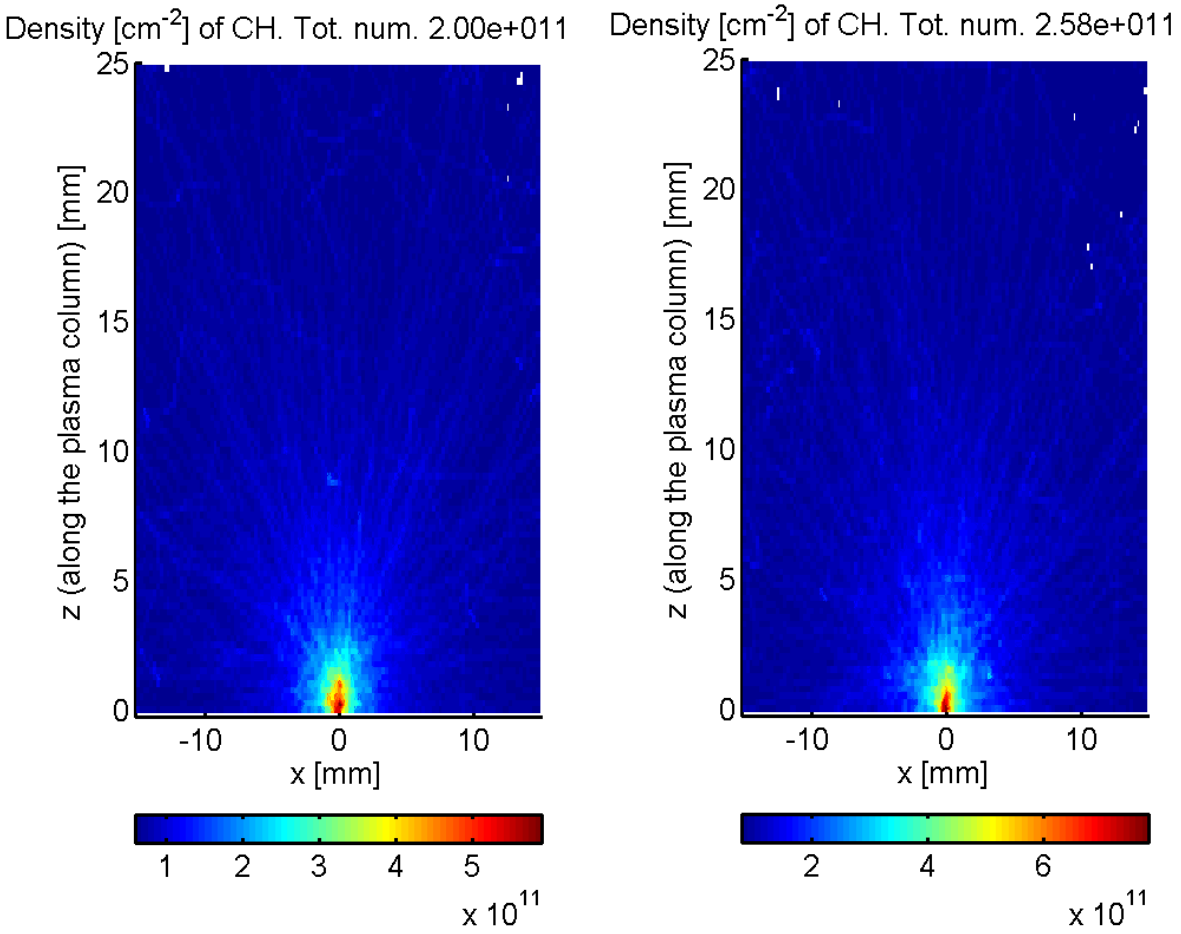
Die Daten werden von ERO.ETT über alle Zellen der Ebene aufsummiert und es reduziert so eine Vielzahl von Einzelwerten auf einen charakteristischen Wert, der die Gesamtladungsdichte des Simulations-Volumens repräsentiert. Die Dichte-Daten sollten aufsummiert in jeder Ebene den gleichen Wert ergeben, es sollte also gelten:

$$\sum_i \sum_j \sigma_{(i,j)} = \sum_i \sum_k \sigma_{(i,k)} = \sum_j \sum_k \sigma_{(j,k)} = const \tag{3.6}$$

Auch hier ist eine nicht gegebene Übereinstimmung der drei Werte wieder ein Abbruchkriterium für die Validierung. Der ermittelte charakteristische Wert wird mit dem von der Referenz erhaltenen Wert anhand des vorgegebenen relativen Fehlers verglichen. Ist der errechnete Fehler kleiner, gelten die Daten als validiert und das Programm macht mit der nächsten Funktion weiter. Bei einem größeren Fehler ist der gesamte Validierungsprozess gescheitert, was festgehalten und später bei der Ausgabe des Testreports ausgegeben wird.

3.2.2.2 Berechnung der mittleren Eindringtiefe

Die Eindringtiefe der Verunreinigungen in das Plasma ist ebenfalls ein sinnvoller charakteristischer Wert, anhand dessen man zwei Läufe der Simulation miteinander vergleichen kann. In Abbildung 3.3 sind zwei Verteilungen von Teilchen abgebildet, welche auf simulierten Dichtedaten basieren und die Verteilung von Kohlenwasserstoff, also CH, in einem Simulations-Volumen einer Linearmaschine visualisieren. Gezeigt ist hier die in **y**-Richtung integrierte **xz**-Ebene. Die Abbildungen zeigen ein für Ionisationsprozesse charakteristisches Bild.
In der Realität ist der Ort mit der höchsten Konzentration, unten in der Mitte der Abbildungen, ein Einlassloch innerhalb eines Fusionsexperimentes, durch das CH_4-Moleküle in das Plasma eingeblasen werden. Dort zerfallen diese in die CH-Moleküle, welche von Interesse sind. Die beiden Abbildungen stammen aus einem seriellen und einem parallelen ERO-Lauf und sind visuell kaum unterscheidbar. Trotzdem lässt sich schon anhand der totalen Anzahl an CH-Teilchen ein Unterschied feststellen, der aber rechnerisch bestimmt werden will.



(a) CH Dichte

(b) CH Dichte

Abbildung 3.3: In y-Richtung integrierte Dichte der CH-Teilchen

Zur Berechnung der mittleren Eindringtiefe der Verunreinigungen wird der in Abbildung 3.4 visuell dargestellte Ansatz gewählt. Man sieht eine Teilchenwolke in einem exemplarischen Simulations-Volumen, deren Teilchendichte schon in eine Richtung integriert wurde. Um die Teilchenwolke wird eine rechteckige Box konstruiert. Die Implementierung ist in Abbildung 3.6 als Nassi-Shneidermann-Diagramm aufgeführt.

Die Box hat dabei folgende Eigenschaften:

- Startpunkt der Box (x_0, y_0)
- Breite(w)
- Länge(l)
- Winkel(α)
- Anzahl der Abschnitte(n)

Der Winkel ist beispielsweise zu gebrauchen, falls die Testteilchen in der Simulation nicht senkrecht oder waagerecht in das Simulations-Volumen, sondern unter einem bestimmten Winkel eingeblasen werden.
Die Box wird gemäß dem konfigurierten Wert, in mehrere Abschnitte $(\Omega_0, \Omega_1, \Omega_2, ...\Omega_n)$ unterteilt, welche die Breite der Box und die Länge $\frac{Laenge_{Box}}{n}$ haben.

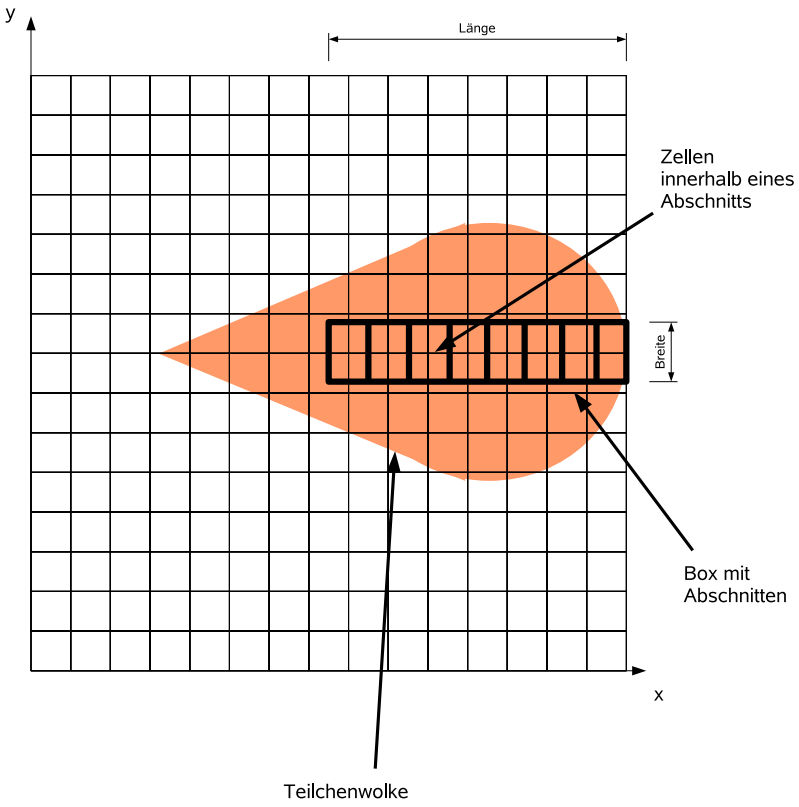


Abbildung 3.4: Schematik zur Berechnung der mittleren Eindringtiefe

Nun werden zunächst die Mittelpunktskoordinaten der Zellen bestimmt, die innerhalb dieser Box liegen, denn nur diese sind von Interesse. Es wird anschließend bestimmt in welchem

Abschnitt der Box sich diese Zellen befinden. Für jeden Abschnitt der Box wird anhand der innerhalb befindlichen Zellen der Mittelwert der Verunreinigungsdichte, sowie die zugehörige Eindringtiefe, bezogen auf die Ausrichtung und den Startpunkt der Box, berechnet. Damit erhält man Paare von Werten mit jeweils einem X-Anteil, welcher die Eindringtiefe, respektive die x-Koordinate des Mittelpunktes des Abschnittes, darstellt und einem Y-Anteil, welcher die gemittelte Verunreinigungsdichte in diesem Abschnitt symbolisiert. Somit erhält man ein charakteristisches Profil, welches im folgenden vorgestellt wird.
In Abbildung 3.5 ist auf 1 normiert aufgetragen, wie hoch der Anteil der Verunreinigungen, bezogen auf eine bestimmte Eindringtiefe, in z-Richtung ist. Hier ist die mittlere Eindringtiefe in z-Richtung aufgetragen, da dieses Profil zu den beiden Bildern in 3.3 gehört, wo die Testteilchen senkrecht von unten eingeblasen wurden. Über dieses Profil lässt sich nun die mittlere Eindringtiefe berechnen, indem man zunächst die zusammengehörenden Paare, mittlere Eindringtiefe und Verunreinigungsdichte, multipliziert aufsummiert und anschließend durch die Summe der mittleren Verunreinigungsdichten teilt. Man kann gut erkennen, dass die mittlere Eindringtiefe, symbolisiert durch den senkrechten Balken in beiden Fällen leicht unterschiedlich ist.

$$\frac{\sum_{i=0}^{n} \sigma_i \cdot x_i}{\sum_{i=0}^{n} \sigma_i} \tag{3.7}$$

mit
σ_i: mittlere Verunreinigungsdichte im Abschnitt Ω_i
x_i : x-Koordinate des Mittelpunkts des Abschnittes Ω_i

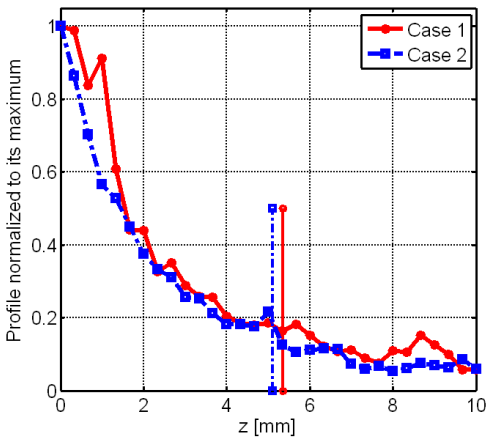


Abbildung 3.5: Eindringtiefe, bezogen auf eine normierte Anzahl von Teilchen

Die mittlere Eindringtiefe wird sowohl für die Referenzdaten als auch für die eigentlichen Testdaten berechnet. Durch die Monte-Carlo-Basiertheit der Simulation ergibt sich die Tatsache, dass die mittlere Eindringtiefe der Referenz und der Testdaten in einem gewissen Rahmen unterschiedliche Werte aufweisen kann. Daher wird der relative Fehler zwischen Referenz und Probe berechnet und mit dem vorher festgelegten relativen Fehler verglichen. Ist dieser Vergleich erfolgreich, gilt die mittlere Eindringtiefe als erfolgreich validiert.

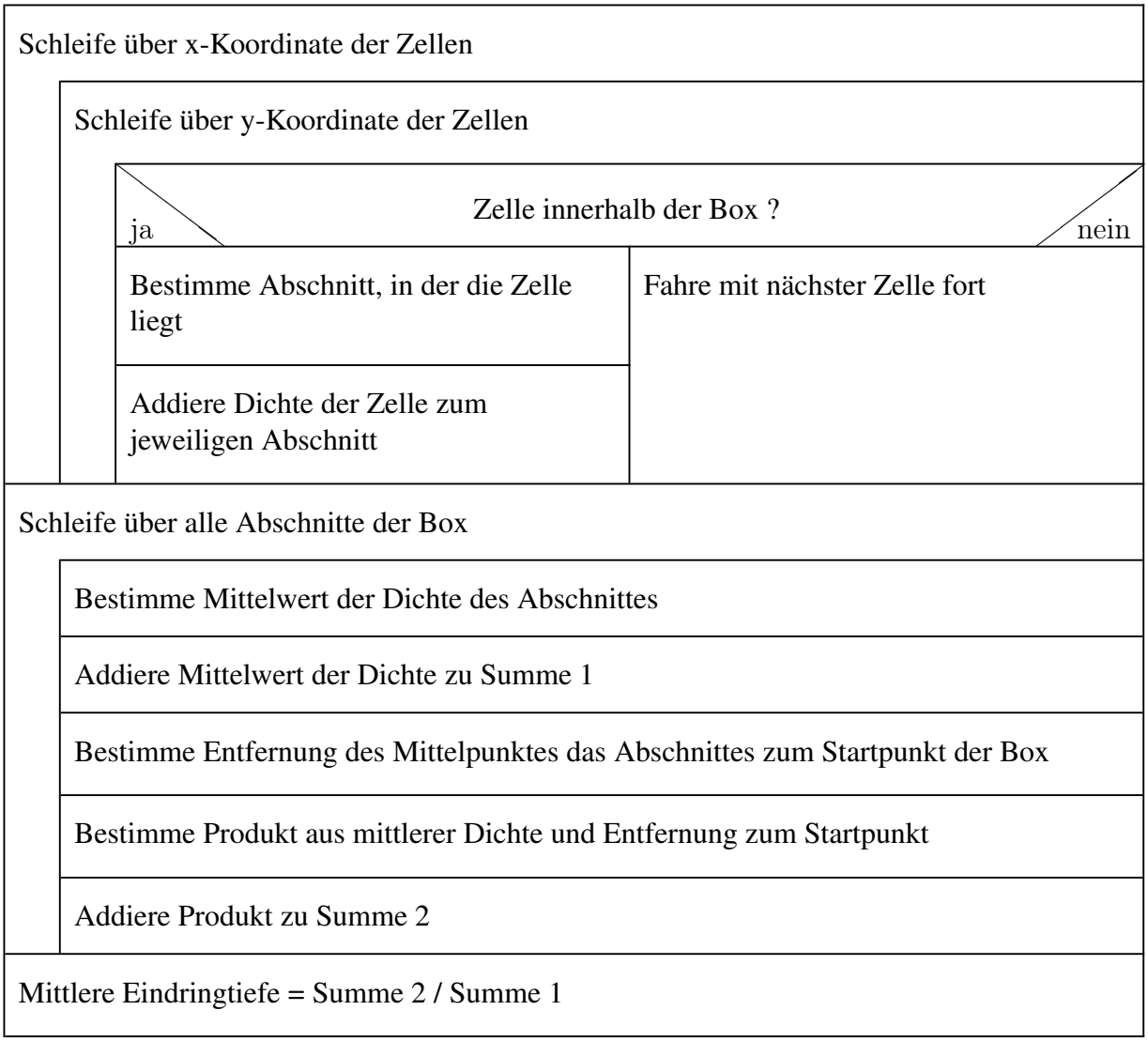


Abbildung 3.6: Nassi - Shneidermann Diagramm zur Berechnung der mittleren Eindringtiefe.

3.2.3 Behandlung von Oberflächendaten

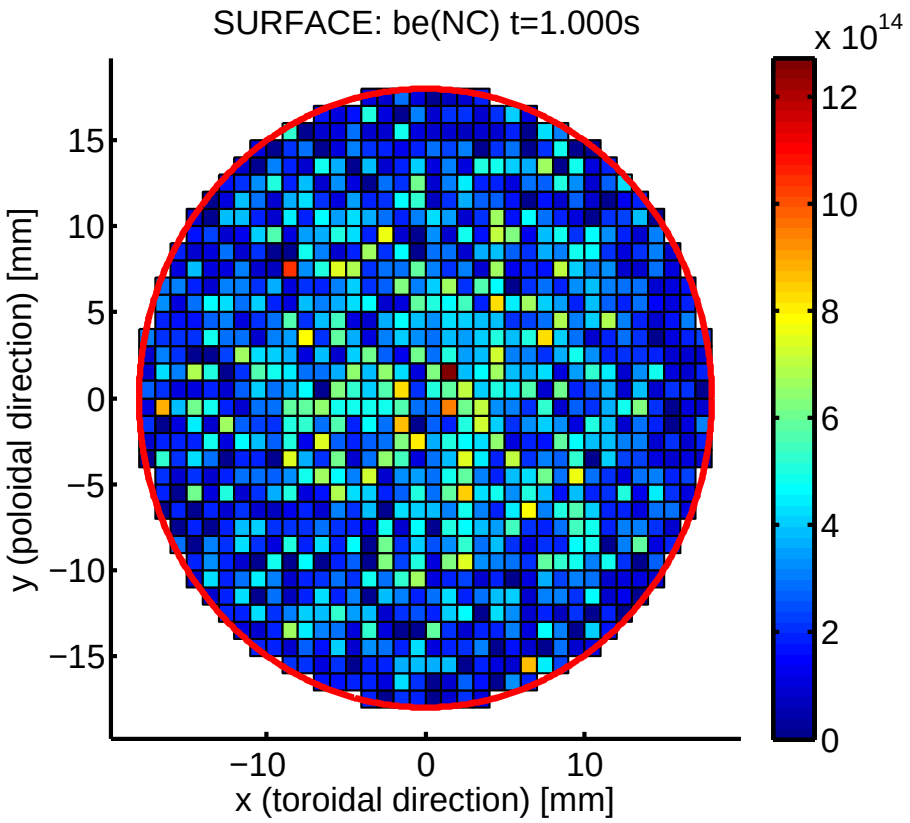


Abbildung 3.7: Oberflächendaten deponierter Teilchen (**NC**)

Unter Oberflächendaten versteht man in ERO solche Simulationsdaten, welche am Material innerhalb des Simulations-Volumens anfallen. Da in ERO die Plasma-Wand-Wechselwirkung (PWW) durch Prozesse wie Erosion, Deposition und Re-Deposition simuliert wird, finden sich diese Daten im verwendeten Oberflächennetz wieder. ERO verfolgt die Konzentration von Elementen in der obersten Oberflächenschicht („Interaction Layer“). Zur Kontrolle werden hier pro Zelle des Oberflächennetzes in jedem Zeitschritt die Anzahl der neu deponierten Elemente (**NC**[2]), die Anzahl der erodierten und die Anzahl der re-deponierten Elemente gespeichert. Dies ist lediglich eine kleine Auswahl von insgesamt 15 gespeicherten Zelleigenschaften, um eine ungefähre Vorstellung davon zu bekommen, wie umfangreich die Daten sind, die bei der Simulation anfallen. In Abbildung 3.7 ist beispielsweise die Anzahl der netto deponierten Teilchen auf dem Oberflächenmaterial zu sehen.
Wie schon im Fall der Spektroskopie- und Elektronendichtedaten überprüft ERO.ETT auch

[2] entspricht dabei “New Come“

hier wieder zunächst die globalen Daten, also Daten, die exakt übereinstimmen müssen und nicht von der Statistik der MC-Simulation abhängen. Dies ist wiederum die Dimension des Oberflächennetzes, der aktuelle Zeitschritt und die Anzahl der verfolgten Elemente. Stimmen diese Daten nicht exakt mit den Referenzwerten überein, macht es keinen Sinn, die weiteren Ausgabedaten zu vergleichen. Es gab damit grundlegend unterschiedliche Rahmenbedingungen für die Simulation. Hier wird wie bei einer fehlgeschlagenen Prüfung sowohl der abweichende, als auch der Vergleichswert ausgegeben und die Validierung als fehlgeschlagen gekennzeichnet.
Darauf hin überprüft ERO.ETT jede Zelle des Netzes, indem es die in ihr enthaltenen Werte aufsummiert, um den charakteristischen Wert für das jeweilige Element zu erhalten. Hier findet wieder ein Vergleich mit dem maximalen relativen Fehler, bezogen auf die Referenzdaten, statt.
Bei der Behandlung der Oberflächendaten werden zudem zu den einzelnen Gitterzellen gehörige Daten exakt verglichen, da sie unabhängig von der Monte-Carlo Eigenschaft der Testteilchen sind. Dies sind beispielsweise die Zellgröße oder die Oberflächentemperatur, welche als Inputparameter an ERO übergeben werden.

3.2.4 Programmablaufplan

Lade Konfigurationsdatei
Lade Referenzdaten
Lade Probedaten
Initialisiere Funktionsprototypen
Weitere Funktionen in Konfigurationsdatei vorhanden?
Führe gefundene Funktion aus
Schreibe Ausgabedatei
Versende Ausgabedatei per Mail

Abbildung 3.8: Programmablaufplan von ERO.ETT.

In Abbildung 3.8 wird die generelle Arbeitsweise von ERO.ETT beschrieben. Zunächst wird die Konfigurationsdatei geladen. Es werden globale Daten wie etwa der zu nutzende relative Fehler und das Ausgabeverzeichnis zur weiteren Nutzung intern abgespeichert. Im nächsten Schritt lädt ERO.ETT die Referenz- und Probedaten und prüft diese zunächst auf generelle Konsistenz, die nicht vom verwendeten relativen Fehler abhängt. So muss beispielsweise die Anzahl der simulierten Zeitschritte, die Größe des Simulationsvolumens und die Zellgrößen des Volumen- und Oberflächennetzes exakt übereinstimmen. Danach werden die Funktionsprototypen initialisiert. Diese dienen als Templates für die intern genutzten Funktionen. Sie erleichtern die Zuordnung der in der Konfigurationsdatei gesetzten Funktionen mit ihren Eingabeparametern zu den fest in ERO.ETT implementierten, welche die charakteristischen Werte berechnen und vergleichen (vgl. Tabelle 3.2). Für jede in der Konfigurationsdatei gefundene Funktion wird das entsprechende Pendant in ERO.ETT mit den gesetzten Parametern gestartet.
Die Ergebnisse des Vergleichs werden anschließend in eine Ausgabedatei geschrieben, den Report. Wurden alle, in der Konfigurationsdatei gefundenen Funktionen, aufgerufen, wird der Report geschlossen und an die angegebene E-Mail-Adresse gesendet. Dies funktioniert jedoch lediglich auf Systemen, die das Versenden von Mails auch erlauben, was beispielsweise bei Juropa / HPC-FF nicht der Fall ist.

3.2.5 Testreport und Benachrichtigung des Benutzers

Die Hauptaufgabe eines Testreports bei einem automatischen Validierungssystems sollte darin bestehen, die Ausgabe bei einer erfolgreich bestandenen Validierung möglichst gering zu halten, während im Fall einer Unstimmigkeit genauere Informationen erforderlich sind, um den Fehler möglichst rasch eingrenzen zu können. ERO.ETT erfüllt diese Voraussetzung, indem es für jeden bestandenen Test nur eine kurze Status-Information ausgibt, um welchen bestandenen Test es sich handelt (vgl. Listing 3.4).

Listing 3.4: Erfolgreicher Test

```
**************************************
* SUCCESSFUL CHECK!!!!!
* Element: all
* Checking <Compare Sum of dens> successful!
**************************************
```

Tritt jedoch eine Unstimmigkeit auf, gibt ERO.ETT detailliertere Informationen aus, wie den Wert, welcher nicht im Rahmen des relativen Fehlers mit der Referenz übereinstimmt sowie den eigentlichen Referenzwert (vgl. Listing 3.5). Zudem wird auch der Ort der Datei angegeben, wo diese Unstimmigkeit auftrat. Dies ermöglicht dem Nutzer des Programms, den Suchprozess einzuschränken, da nur solche Teile des Codes überprüft werden müssen, die Einfluss auf die überprüften Ausgabedaten haben.

Listing 3.5: Fehlgeschlagener Test

```
**************************************
* FAILED CHECK!!!!!
* Checking <Compare StatusSums> failed!
* File: /home/galonska/ero/eroTestBed/PISCES_probe/BiasPot1p2_Surface_
    step0.m
* ValueName: be__ -> _T
* ValueReference: -8.985728e+16
* ValueProbe:  -4.029777e+16
**************************************
```

Die Ausgaben des ERO Testing Tools werden während des Validierungsprozesses in ein Testprotokoll geschrieben, dessen Dateiname den Namen des Tests (s. $taskname in 3.1) sowie das aktuelle Datum und die Uhrzeit enthält, um diese später leichter identifizieren zu können. Gleichzeitig sendet ERO.ETT nach Ende der Validierung eine E-Mail mit diesem Testprotokoll als Anhang an die, in der Konfigurationsdatei angegebene, E-Mail-Adresse (vgl. Kap. 3.2.1). In der Betreffzeile der Mail steht dabei der Testname und eine Statusmeldung, die angibt ob der Test fehlgeschlagen ist oder nicht. Dadurch wird im Falle einer erfolgreichen Validierung eine manuelle Kenntnisnahme des Testprotokolls vermieden. Dies ermöglicht ein direktes Feedback der Anwendung an den Nutzer und dieser muss das Testprotokoll nicht extra lokalisieren, um es analysieren zu können. So wird der Nutzer auch zeitnah über einen abgeschlossenen Test informiert. Um JuBE die Verarbeitung des Testreports zu erleichtern, schreibt ERO.ETT an den Schluss des Reports noch in einem einheitlichen Format, ob die Validierung erfolgreich bestanden wurde.

3.3 Zusammenfassung der Testergebnisse durch JuBE

Da ERO.ETT in JuBE eingebunden ist, können die erhaltenen Testergebnisse nach Ende der Ausführung von ERO.ETT wiederum von JuBE verarbeitet werden. Dies hat den Vorteil, dass JuBE alle Ereignisse und Daten, die während eines Validierungslaufs anfallen, abspeichert und so bei späteren Tests leicht darauf zurückgegriffen werden kann. JuBE extrahiert dabei, mit Hilfe von regulären Ausdrücken, die gewünschten Daten aus den Ausgabedaten und kann diese anschließend in einer übersichtlichen Tabelle darstellen. Ein Beispiel einer solchen Tabelle ist in Listing 4.2 abgebildet. Für einen Validierungslauf von ERO sind dabei auch technische Daten wie etwa die Anzahl der Testteilchen, die benötigte Laufzeit, sowie die Laufzeiten der einzelnen in ERO enthaltenen Module (vgl. Kap. 2.3) oft relevant. JuBE extrahiert außerdem die von ERO.ETT extra zu diesem Zweck herausgeschriebene Information, ob die Validierung erfolgreich war.

3.4 Grundkonfiguration und Referenzfälle

Das AVS soll eine möglichst breite Palette an Szenariovariationen von ERO testen, da bestimmte Szenarien auch nur bestimmte Teile des Codes ansprechen. Dazu wurden zunächst drei möglichst unterschiedliche Szenarien ausgewählt, deren Validierung simultan ablaufen soll. Die Unterschiede zwischen diesen Szenarien liegen hauptsächlich in folgenden Merkmalen:

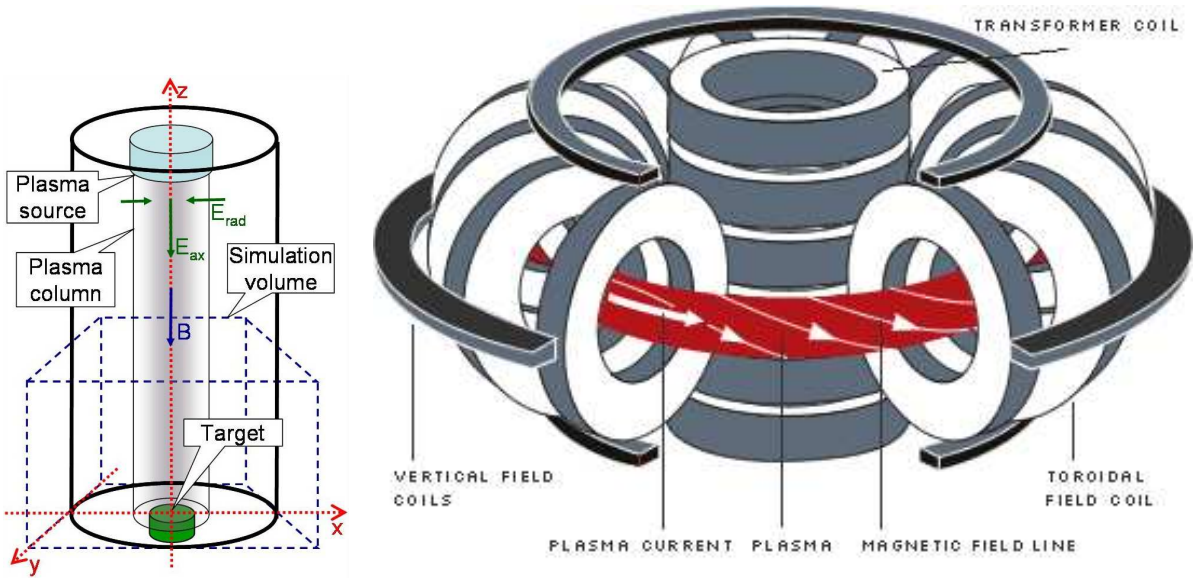
- Eingabeparameter
- Verschiedene Datenbanken für atomare/molekulare und Oberflächen-Daten
- Kompilierung
  - Kopplung mit anderen Softwaremodulen
  - Verschiedene Fusionsexperimente (Limiter, Divertor, Linear-Maschine)
  - Verschiedene Computerarchitekturen
- Analyse der Ausgabedaten während der Validierung

Die Vorstellung von drei verschiedenen Szenarien erfolgt in den nächsten Abschnitten

3.4.1 PISCES-B

Bei PISCES-B [6] handelt es sich um ein Plasmaexperiment mit einer linearen, zylindrischen Geometrie.

Wie in Abbildung 3.9 ersichtlich ist, unterscheidet sich der Aufbau von PISCES-B grundlegend von dem eines Tokamak-Fusionsexperimentes. Während bei einem Tokamak das Fusionsplasma einen geschlossenen Torus bildet, kann man gut erkennen, dass bei PISCES-B lediglich eine Plasmasäule existiert.
Die Konfiguration des elektromagnetischen Feldes, der Geometrie und der Position von Schlüsselkomponenten und weiteren Besonderheiten führen zu der Notwendigkeit, eine separate Version von ERO zu entwickeln, welche jedoch die Kernkomponenten bereitstellt. Diese Version enthält große, alternative Codeblöcke, deren Nutzung durch den Präprozessor gesteuert wird. Es entstehen so in der Simulation andere Ausgabedaten als dies bei einem Tokamak-Experiment der Fall wäre. Diese müssen also anders ausgewertet und validiert werden als bei einer Tokamak Simulation.
PISCES-B ist momentan praktisch das einzige fusions-relevante PWW-Experiment, welches Versuche mit dem giftigen Beryllium erlaubt. Beryllium kann dabei entweder injiziert oder von einem Target erodiert werden, welches mit Beryllium beschichtet ist. Die verwendeten Targets sind austauschbar.



(a) Schematik von PISCES-B (b) Schematik eines Tokamak

Abbildung 3.9: Unterschiede zwischen den Geometrien von PISCES-B und Tokamak[3]

3.4.2 Höhere Kohlenwasserstoffe

Das Plasma in ITER soll aus Deuterium (D / ^2H) und Tritium (T / ^3H) bestehen, also schweren Wasserstoffisotopen, die zur Fusion gebracht werden. Wandmaterialien in Fusionsexperimenten aus Kohlenstoff sind momentan Gegenstand der Forschung, da diese dafür vorgesehen sind, in den Divertor-Platten von ITER eingesetzt zu werden [15]. Divertoren sind Teile des Tokamaks oder Stellarators, wo die Haupt-PWW-Prozesse stattfinden, da die Verunreinigungen dorthin geleitet werden. Durch zusätzliche spezielle magnetische Spulen wird der Plasmafluss in eine so genannte „Divertor-Kammer“ auf „Prallplatten“ abgeleitet. Die Plasmaverunreinigungen, inklusive das Fusionsprodukt Helium, werden dort effektiv abgepumpt. Der Wasserstoff im Plasma kann in die kohlenstoff-haltige Wand eindringen und dort effektiv Kohlenwasserstoff-Moleküle bilden. Diese wiederum können ins Plasma gelangen, was chemische Erosion genannt wird. Die chemische Erosionsrate von Kohlenstoff hängt dabei hauptsächlich von der Energie des auftreffenden Teilchens und der Temperatur der Oberfläche ab, mit der die Wasserstoffe reagieren. In Experimenten werden Teilchen mittels Spektroskopie, also ihrer spezifischen Lichtausstrahlung verfolgt.
An der Wand können nun mehrere unterschiedlich konfigurierte Kohlenwasserstoff-Moleküle gebildet werden, inklusive Methan (CH_4) und „höhere Kohlenwasserstoffe“, die mehr als ein C-Atom besitzen. Diese Moleküle dissoziieren (zerfallen) wiederum im Plasma zu „kleine-

[3]Quelle: (a) [6], (b) `http://www.ipp.mpg.de/ippcms/eng/images/pic/images_bereiche/allgemein/380x265/tokamak_schema.gif` [4]

ren“ Molekülen (vgl. Abbildung 3.10). Nur wenige davon (z.B. CH, C_2) haben Spektrallinien, die gut beobachtet werden können. Meistens ist die von ihnen emittierte Strahlung zu schwach oder ihre Linien werden von den Linien anderer Spezies überdeckt.

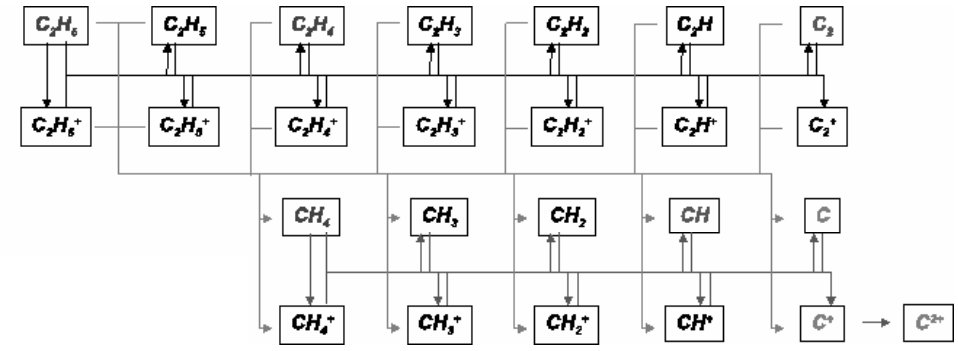


Abbildung 3.10: Zerfallsketten höherer Kohlenwasserstoffe[4]

Die Kohlenwasserstoff-Moleküle werden in ERO in einem gesonderten Szenario verfolgt [9] und die Ausgabedaten sind entsprechend der Vielzahl der Moleküle umfassender. Dazu ist eine entsprechend große Molekül-Datenbank notwendig, die die verschiedenen Konfigurationen aus Kohlenstoff und Wasserstoff, sowie entsprechende Dissoziationsraten für jede Reaktion, abhängig von der Elektronen-Temperatur, beinhaltet. Die Eingabeparameter von ERO müssen so modifiziert werden, dass ERO auf diese Datenbank zugreifen kann. Das Szenario selbst wird über eine C/C++ - Präprozessor-Makrodefintion aktiviert, so dass hier der Kompilierungsprozess ebenfalls modifiziert werden muss.

3.4.3 ERO-SDTrimSP Kopplung

Bei SDTrimSP [10] (**S**tatic and **D**ynamic **TR**ansport of **I**ons in **M**atter for **S**equential and **P**arallel Computers) handelt es sich um einen in der Programmiersprache Fortran verfassten Code, der die physikalischen und chemischen Reaktionen innerhalb der Oberfläche einer Wand simuliert. Es kann mehrere Schichten an Materialien simulieren und verfolgt die Teilchen und ihre Interaktionen mit den Materialien innerhalb dieser Oberfläche. Dazu bedient sich der Code der binären Kollisions-Approximation (BKA), also der Kollision von zwei Partikeln. Während der Kollision werden die Wechselwirkungen mit allen anderen Partikeln vernachlässigt, allerdings kann das gestoßene Teilchen, abhängig von seiner Energie und der Eindringtiefe, wiederum ein anderes Teilchen anstoßen. Dadurch entsteht eine so genannte „Stoß-Kaskade“.

[4] Quelle: IEF-4

Es werden folgende Vorgänge, bezogen auf die Testteilchen, simuliert:

- Zerstäubung
- Reflexion
- Ablagerung
- Chemische Erosion

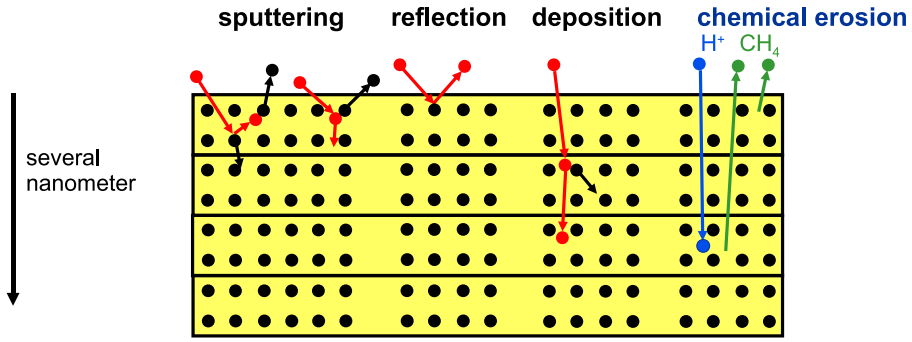


Abbildung 3.11: Schematische Darstellung der Vorgänge in SDTrimSP

Die einzelnen Prozesse sind in Abbildung 3.11 schematisch dargestellt. Von besonderem Interesse bei der Verwendung von SDTrimSP sind die Reflexionskoeffizienten (3.8) und die Zerstäubungsausbeute (3.9).

$$R = \frac{n_{\text{reflektiert}}}{n_{\text{eingetroffen}}} \tag{3.8}$$

$$Y = \frac{n_{\text{zerstaeubt}}}{n_{\text{eingetroffen}}} \tag{3.9}$$

mit
n : Anzahl der Testteilchen

Der Reflexionskoeffizienten gibt hierbei an, welcher Anteil der Testteilchen, die die Oberfläche erreicht haben, wieder reflektiert wird. Respektive macht die Zerstäubungsausbeute eine Aussage darüber, wie hoch der Anteil der zerstäubten Teilchen ist.
Im Zusammenspiel der beiden Codes wird bei jedem Eindringen eines ERO-Testteilchens in die Wand ein SDTrimSP-Lauf gestartet. Das ERO-Testteilchen wird dabei durch mehrere SDTrimSP-Testteilchen repräsentiert, um einen akzeptablen statistischen Fehler in 3.8 und 3.9

zu haben.
Eine weitere Besonderheit dieses Simulations-Szenarios besteht aus technischer Sicht in der Kopplung aus Fortran (SDTrimSP) und C/C++ (ERO). Daher müssen hier sowohl Fortran, als auch C/C++ Quellen gesondert kompiliert und anschließend zu einem einzelnen Programm gelinkt werden. Dieser Vorgang ist insofern problematisch, als dass auf unterschiedlichen Plattformen verschiedene Compiler, sowohl auf C/C++ als auch auf Fortran Seite zum Einsatz kommen. Die verschiedenen Fortran Compiler erzeugen unterschiedliche Namen für die implementierten Funktionen, so dass die Schnittstelle zwischen den Codes, je nach Plattform, auch die Funktionen unter diesen unterschiedlichen Namen aufrufen können muss. Dafür sind wiederum Präprozessor-Makros notwendig, die den jeweiligen Namen dem Compiler nach entsprechend verändern. Andernfalls erkennt der Linker die Funktionen auf Fortran-Seite nicht und der Link-Vorgang schlägt fehl.
Auch dies ist entsprechend im AVS konfiguriert worden.

Kapitel 4

Verwendung des AVS zur Analyse der Parallelisierung

4.1 Fallbeschreibung von Case 6

In Case 6 von ERO (vgl. Tabelle 2.1) wird die physikalische Erosion von Teilchen aus der Oberfläche behandelt. Es werden alle Zellen des Oberflächengitters durchlaufen und jeweils eine bestimmte Anzahl Testteilchen generiert. Diese starten nun unter einem Monte-Carlo verteilten Winkel und einer Anfangsgeschwindigkeit in das Simulations-Volumen. Von diesen Testteilchen wird eine Trajektorie aufgezeichnet und der entsprechende Weg durch das Volumen so verfolgt. Dies soll den Effekt der physikalischen Erosion möglichst realitätsnah modellieren (siehe Abbildung 4.1).

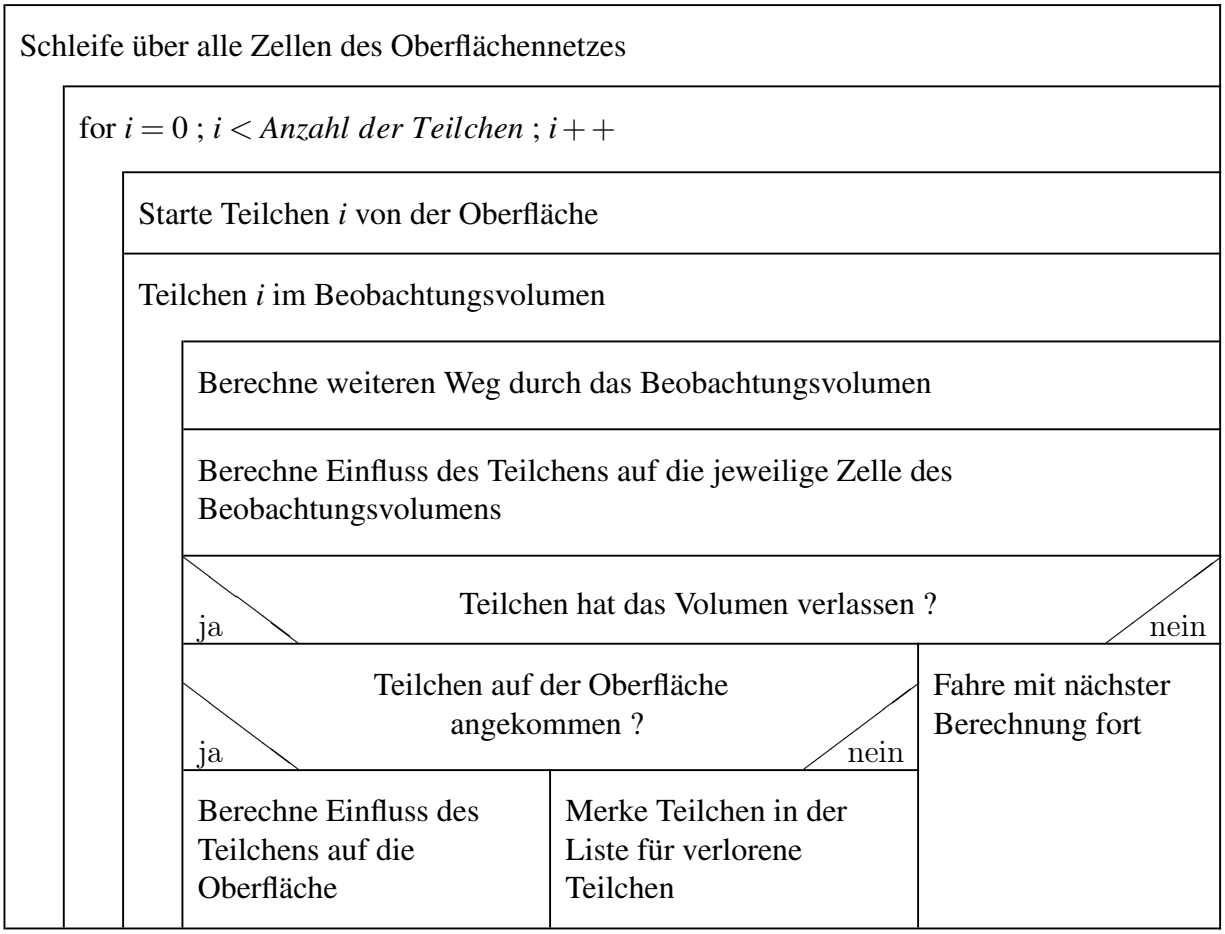


Abbildung 4.1: Nassi - Shneidermann Diagramm für Case 6 (Seriell)

Dieses Szenario ist in ERO mit OpenMP parallelisiert worden, da es sehr zeitaufwändig ist und die Ausführungszeit auf Shared Memory Systemen dadurch verringert werden soll. Der hohe Zeitaufwand resultiert aus der Anzahl der Zellen des Oberflächengitters und der zu startenden Anzahl von Testteilchen pro Zelle. Die Anzahl der Testteilchen pro Zelle beträgt im Referenzfall mindestens 100 und wird durch die Eingabeparameterdatei von ERO festgelegt, so dass dieser Wert auch deutlich höher ausfallen kann. Durch den Einsatz des AVS soll gewährleistet werden, dass ERO auch auf mehreren Prozessoren in einem bestimmten Rahmen die gleichen Ergebnisse liefert, wie bei einem seriellen Lauf. Der Ansatz, welcher für die Parallelisierung gewählt wurde, ist in Abbildung 4.2 dargestellt.

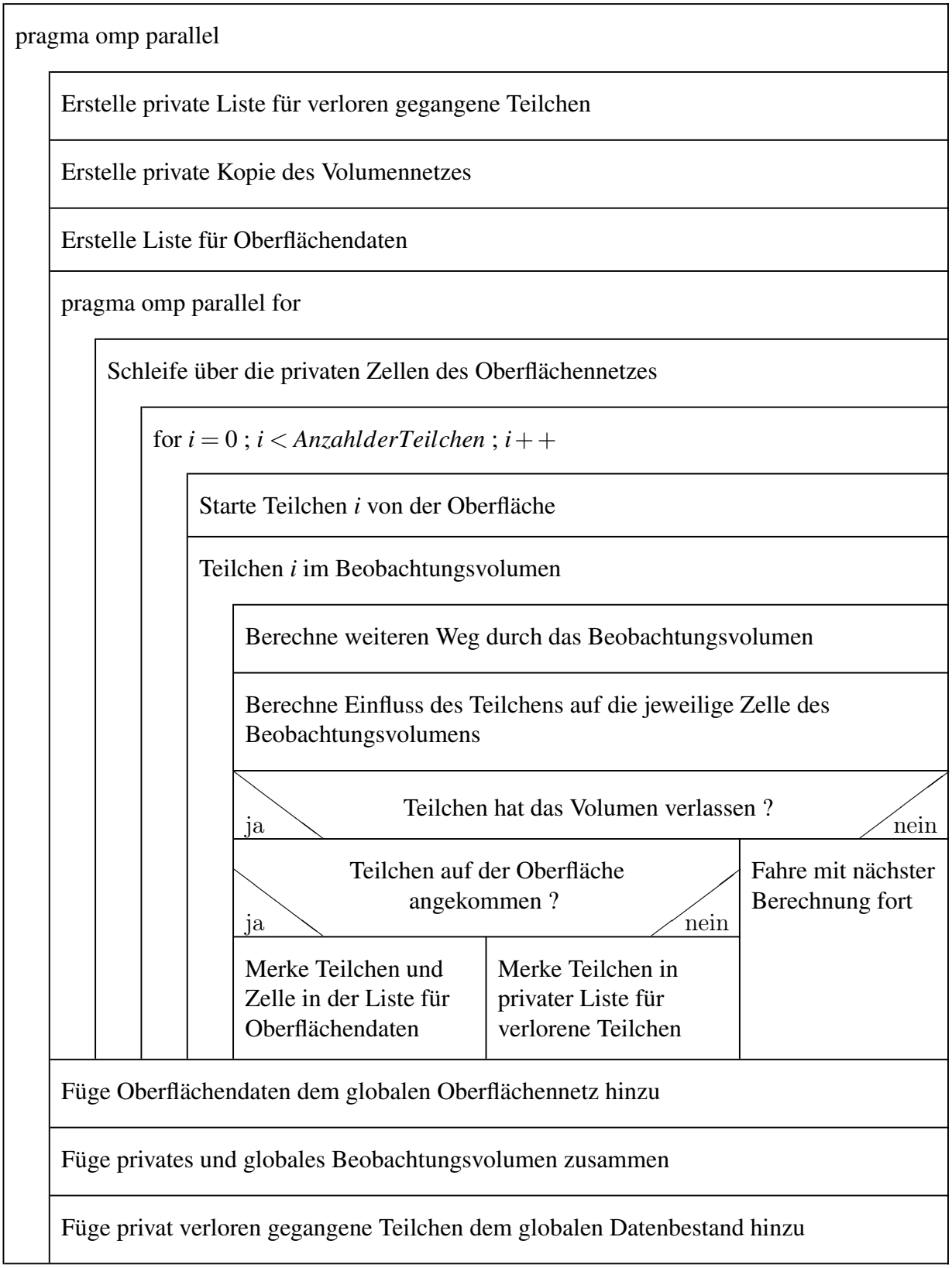


Abbildung 4.2: Nassi - Shneidermann Diagramm für Case 6 (Parallel)

4.2 Validierung

Zur Vorbereitung der Validierung wurde zunächst das entsprechende Modul in ERO (Case 6) aktiviert und ein serieller Lauf gestartet, welcher die Referenzdaten ausgibt. Die Referenzdaten wurden anschließend in JuBE abgelegt.
Im Top-Level XML File von ERO wurde ein neuer Eintrag gesetzt und so konfiguriert, dass die Eingabedaten für ERO zu denen des seriellen Laufs passen. Dies beinhaltet eine zum seriellen Lauf passende Eingabeparameterdatei, in der die Platzhalter für Testteilchen und Zeitschritte durch adäquate Werte ersetzt werden.
Bei der anschließenden Validierung stellte sich heraus, dass die Werte des Probelaufs nicht mit den Referenzdaten aus dem seriellen Lauf übereinstimmen, weshalb von einer Nutzung der Parallelisierung abzuraten war. Aufgrund der detaillierten Ausgabe von ERO.ETT konnten daraufhin die Fehler ermittelt werden, die zu den Abweichungen geführt haben. Um das Auffinden der Abweichung zu erleichtern, wurde zunächst ein Mechanismus entworfen, der die Ergebnisse des seriellen Laufs **exakt** auf einer vorher festgelegten Anzahl an beteiligten Threads reproduzierbar macht.
Die Reproduzierbarkeit hängt dabei, wie in Kapitel 2.4.2 beschrieben, von der Verwendung des gleichen Random-Seeds ab. Nur wenn dieser gleich ist, sind die produzierten Pseudo-Zufallswerte gleich, die den Weg des Teilchens durch das Beobachtungsvolumen bestimmen. Da bei Verwendung eines einzelnen Random-Seeds in einem parallelisierten Lauf der Zugriff auf die Zufallsvariable von der Last der eingesetzten Threads abhängt und diese auch gleichzeitig, bzw. in unbestimmter Reichenfolge auf die Zufallsvariable zugreifen und so unterschiedliche Werte generieren, wäre der Simulationsverlauf und somit auch die Ausgabedaten jedes Mal unterschiedlich. Deshalb wurde ein Feld von Zufallsvariablen eingeführt, bei dem jedes Element nur für einen Chunk zuständig ist (vgl. Kap. 2.5).

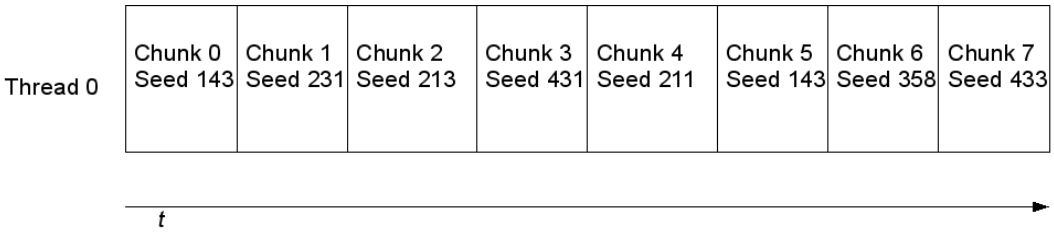


Abbildung 4.3: Serielle Bearbeitung von Chunks

In Abbildung 4.3 ist das Verhalten schematisch dargestellt, welches sich unter Nutzung des Feldes der Zufallszahlen ergibt. Auch wenn dies ein serieller Lauf ist, verwendet der serielle Thread für jeden Chunk eine eigene Zufallszahl. Dadurch werden pro Chunk immer jeweils die gleichen Pseudo-Zufallszahlen erzeugt. Der Nutzen dieser Vorgehensweise wird in Abbildung 4.4 deutlich. Auch hier wird wieder für jeden Chunk eine eigene Zufallszahl erzeugt.

Durch die Bindung der Zufallszahl an den zugehörigen Chunk ist nun jedoch sicher gestellt, dass auch hier wieder die gleichen Pseudo-Zufallszahlen erzeugt werden, diesmal jedoch auf verschiedenen Threads. Diese Tatsache kann aber vernachlässigt werden, da es nur auf die, notwendigerweise übereinstimmenden, Zufallsvariablen pro Chunk ankommt.

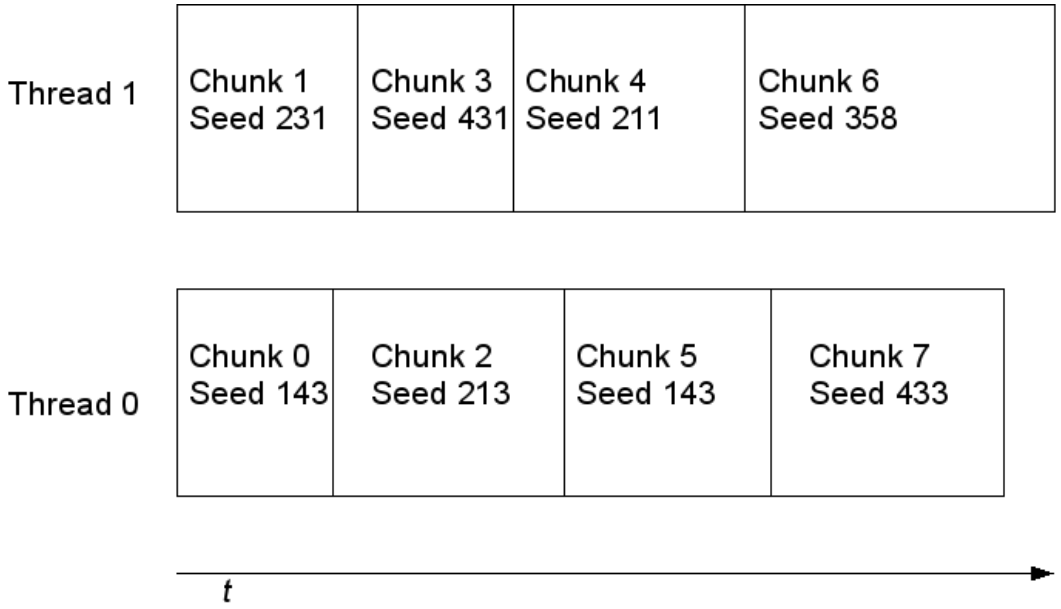


Abbildung 4.4: Parallele Bearbeitung von Chunks

Wie in Abbildung 4.2 ersichtlich ist, simuliert jeder Thread auf seinem eigenen Volumennetz und erzeugt separate Daten für die Oberfläche. Diese werden nach Abarbeitung der Teilchenverfolgung im globalen Datenbestand zusammengefasst und ergeben so wieder ein vollständiges Bild der Simulation. Die Ergebnisse des seriellen Laufs lassen sich exakt reproduzieren, falls die Parallelisierung korrekt implementiert ist.

4.2.1 Beispiel einer fehlgeschlagenen Validierung

Um die Arbeit des AVS zu illustrieren, soll das folgende Beispiel dienen. Listing 4.1 zeigt die Ausgabe von JuBE nach einer Validierung von Case 6. Sowohl der serielle Referenzlauf als auch der Probelauf wurde mit dem gleichen Random-Seed durchgeführt.

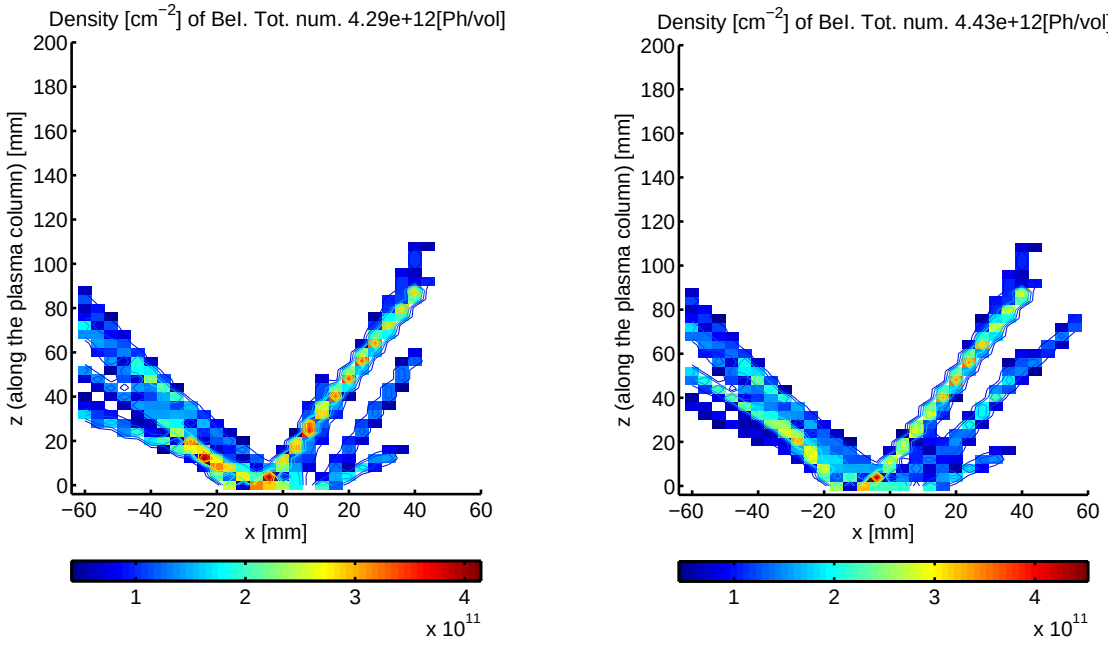
Listing 4.1: Fehlgeschlagener Test von Case 6

```
ERO_Intel-Nehalem-HPC-FF_CASE6_500_i000097
==========================================
 run time
  Subid                    nparts       check
  ------------------ ---------- ----------
     n1p1t8_t001_i01         500      FAILED
     n1p1t4_t001_i01         500      FAILED
     n1p1t2_t001_i01         500      FAILED
     n1p1t1_t001_i01         500      FAILED

 Timings
  Subid                     ptime      c6time
  ------------------ ---------- ----------
     n1p1t8_t001_i01       64.00    62824.16
     n1p1t4_t001_i01       92.00    91005.82
     n1p1t2_t001_i01      234.00   233898.30
     n1p1t1_t001_i01      656.00   655076.99
```

In Listing A.1 im Anhang ist die vollständige Ausgabe des ERO Testing Tools abgebildet. Die ersten drei fehlgeschlagenen Tests beziehen sich auf die Ionendichte im Volumennetz. Im Probelauf wurde weniger Beryllium im Volumen berechnet, als es bei den Referenzdaten der Fall ist. Zudem wurde bei zweifach geladenem Beryllium im Probelauf keine Dichtekonzentration festgestellt. Das letzte Indiz bedeutet einen Fehler bei der Berechnung der Ionisation. Wenn das neutrale Beryllium nicht zweimal ionisiert worden ist, befinden sich die entsprechenden Moleküle auch nicht im Volumennetz. Beim anschließenden Test des Oberflächennetzes wurde festgestellt, dass weniger Moleküle wieder zurück auf die Oberfläche gekommen sind (**NC** = **N**ew **C**ome). Dementsprechend ist auch der Fluss auf die Oberfläche geringer (**FLC** = **FL**uen**C**e). Der fehlgeschlagene Test auf physikalisch erodierte Teilchen in Case 6 (PER6 = **P**hysically **ER**oded Case **6**), welcher nur etwa halb so viele erodierte Teilchen als im Referenzlauf anzeigt, liefert ein erstes Indiz auf die Ursache. Wenn weniger Teilchen erodiert werden und in das Volumennetz starten, können folglich auch weniger den Weg zurück zur Oberfläche finden.
Um das Problem genauer eingrenzen zu können, wurde die Statistik des Laufs extrem reduziert. Es gibt nun lediglich 12 Oberflächenzellen, von denen jeweils nur ein Testteilchen startet, also erodiert wird. Da sich nun nur 12 Testteilchen durch das Volumen bewegen, kann ihr Weg leichter identifiziert werden. In Abbildung 4.5 wurde mit MERO visualisiert dargestellt, wie die 2D Verteilung der Teilchen des zugehörigen Laufs aussieht. Auf der linken Seite

sieht man den seriellen Referenzlauf, auf der rechten Seite den parallelen Probelauf, welcher mit 4 Threads durchgeführt wurde.



(a) Spektroskopie - Seriell

(b) Spektroskopie - 4 Threads

Abbildung 4.5: In y-Richtung integrierte Dichte des Be^0-Lichtes

Die beiden Verteilungen gleichen sich in den meisten Bereichen, allerdings sieht man auch gut, dass mindestens ein Testteilchen einen anderen Weg nimmt, als im seriellen Referenzfall. Der Weg überlagert sich mit einem der anderen Testteilchen, wie man am mittleren rechten Zweig der Verteilung erkennen kann. Zudem ist die Gesamtzahl der Photonen insgesamt etwas höher.
Da der Anschein erweckt wird, dass etwas mit den Testteilchen oder ihrer Verteilung über die Threads nicht stimmt, wurde die betroffenen parallele Region innerhalb von Case 6 genauer untersucht. Dabei wurde festgestellt, dass die Testteilchenvariablen nicht privat sind und die Schleifenvariable, mit der die betroffenen Teilchen angesprochen werden, ebenfalls nicht. In Abbildung 4.2 ist diese Schleifenvariable mit *i* benannt. Diese befindet sich in einer nicht parallelisierten Schleife. Die eigentliche Schleife über die Zellen des Oberflächennetzes (eine Ebene höher) ist mit OpenMP parallelisiert und ihre Schleifenvariable damit implizit privat. Dieser Umstand hat zweierlei Effekte:
Da die Schleifenvariable für die Testteilchen global gelesen und beschrieben werden kann, kann ein und dasselbe Teilchen mehrmals gestartet werden. Und zwar von verschiedenen Threads. Es kann ebenfalls vorkommen, dass ein Teilchen überhaupt nicht gestartet wird, da

sein Index nicht angesprochen wird.
Der zweite Effekt führt zu einem Speicherzugriffsfehler, wie er während der ersten Testläufe öfter aufgetreten ist. Wenn Thread A die Schleife betritt und beim vorletzten, also $N-1$ten Teilchen angekommen ist, wird abgefragt, ob $i < N$ ist. Dies ist offensichtlich der Fall. Betritt nun Thread B die gleiche Schleife, bevor Thread A i inkrementiert hat, liefert seine Abfrage $i < N$ ebenfalls ein positives Ergebnis. Beide Threads inkrementierten daraufhin i, so dass $i = N$ gilt. Somit zeigt i auf ein nicht existierendes Teilchen außerhalb des allozierten Speichers, da das Feld von $0..(N-1)$ indiziert ist. Dies führt zur besagten Speicherzugriffsverletzung.

4.2.2 Beispiel einer erfolgreichen Validierung

Nachdem die für die abweichenden Ergebnisse verantwortlichen Stellen im Code modifiziert wurden, liefert das AVS nun die in Listing 4.2 aufgeführten Ergebnisse. Die Validierung war somit erfolgreich und Case 6 kann nun unter Einsatz von OpenMP genutzt werden. Die Ergebnisse von parallelem und seriellem Lauf mit den gleichen Eingabebaten liefern exakte Übereinstimmung, was mit dem Programm **diff** überprüft wurde

Listing 4.2: Erfolgreicher Test von Case 6

```
ERO_Intel-Nehalem-HPC-FF_CASE6_500_i000095
==========================================
 run time
  Subid                   nparts       check
  ------------------ ---------- ----------
     n1p1t8_t001_i01         500          OK
     n1p1t4_t001_i01         500          OK
     n1p1t2_t001_i01         500          OK
     n1p1t1_t001_i01         500          OK

 Timings
  Subid                    ptime     c6time
  ------------------ ---------- ----------
     n1p1t8_t001_i01     102.00  100942.11
     n1p1t4_t001_i01     119.00  117610.49
     n1p1t2_t001_i01     140.00  138962.13
     n1p1t1_t001_i01     200.00  198346.20
```

4.2.3 Speedup der Parallelisierung

Aus Listing 4.2 wurden nun die Zeiten für Case 6 extrahiert (`c6time`) und ein Speedup-Plot erstellt, der die Verringerung der Ausführungszeit von Case 6 in Abhängigkeit von der Anzahl eingesetzter Threads angibt. Der Speedup ist ein Faktor, welcher die Beschleunigung der Ausführungsgeschwindigkeit eines Programms in Abhängigkeit zur Anzahl genutzter Threads angibt. Er ist definiert durch den Quotienten 4.1:

$$S_p = \frac{T_s}{T_p} \tag{4.1}$$

T_s: Ausführungszeit des seriellen Programms
T_p: Ausführungszeit des parallelen Programms mit p Prozessoren

Eine ideale Speedup-Kurve wäre eine Halbgerade 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.

$$\begin{aligned} T_p &= \frac{T_s}{p}(1-\alpha) + T_s\alpha \\ \Rightarrow S_p &= \frac{T_s}{\frac{T_s}{p}(1-\alpha) + T_s\alpha} = \frac{p}{(1-\alpha)+p\alpha} \leq \frac{p}{p\alpha} = \frac{1}{\alpha} \end{aligned} \tag{4.2}$$

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

Nach 4.2 ist somit der Kehrwert des seriellen Anteils die obere Grenze für den Speedup.
Wie man in Abbildung 4.6 erkennen kann, steigt die Kurve zunächst an und flacht dann ab. Der aktuelle Speedup beträgt bei Nutzung von 8 Threads ~ 2.4. Dies kann, wie oben beschrieben, an einem zu großen seriellen Anteil dieses Szenarios liegen, oder aber an einer nicht effektiv genug implementierten Parallelisierung. Durch weitere in Zukunft vorgesehene Optimierungen der Parallelisierung lässt sich wahrscheinlich eine bessere Skalierbarkeit erreichen.

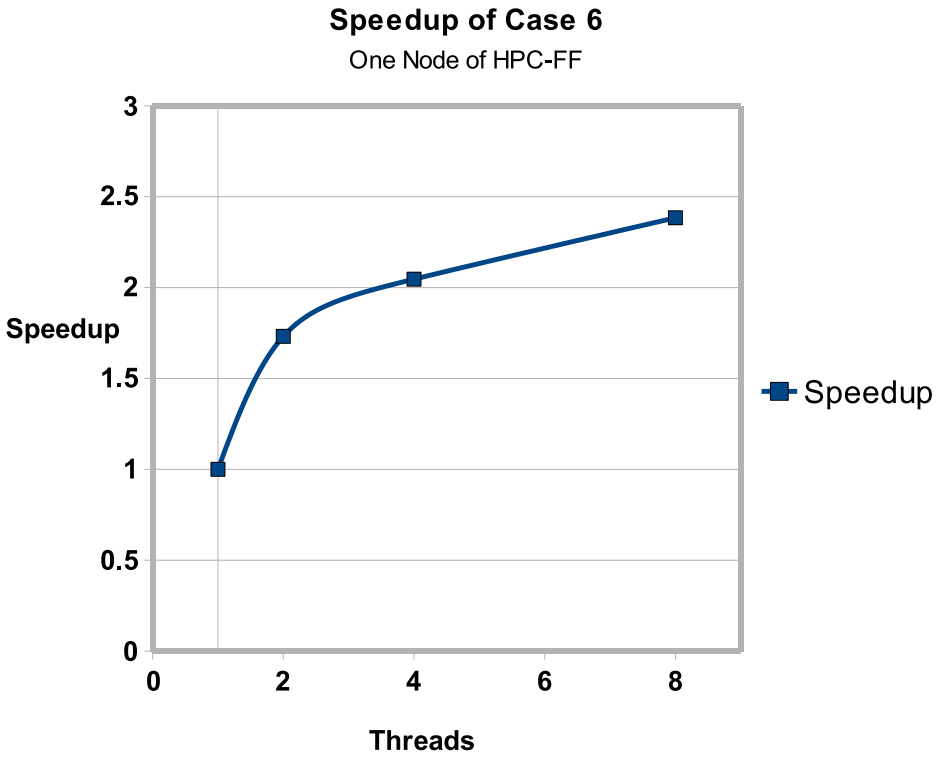


Abbildung 4.6: Speedup von Case 6

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

In Rahmen dieser Arbeit wurde ein automatisches Validierungssystem für den ERO-Code (Kap. 2.3) auf der Basis von JuBE (Kap. 3.1) und eines geeigneten Vergleichs-Tools ERO.ETT (Kap. 3.2) entwickelt. Dieses System kann aber auch zur Validierung anderer Modellierungssoftware, die in Rahmen der EU Task Force ITM betrieben wird, genutzt werden.
Das System basiert auf dem Vergleich von Ausgabedaten einer zu testenden Version des Codes mit „Referenz“-Daten. Es wurden mehrere Szenarien (Kap.3.4) vorkonfiguriert, deren Simulation zu den typischen Aufgaben von ERO zählt und die gut geeignet zum Testen der wichtigsten, in ERO enthaltenen, Module sind. Durch die hohe Flexibilität von JuBE ist es nun möglich, mit einem Aufruf des automatischen Validierungssystems gleichzeitig verschiedenste Szenarien zu validieren, ohne den manuellen Aufwand der Kompilierung, der Vorbereitung von Eingabedaten, der Ausführung und des Vergleichs der Ausgabedaten mit Referenzdaten bewerkstelligen zu müssen. Das automatische System meldet die gefundenen Unterschiede und der Benutzer kann entscheiden, ob diese akzeptabel oder sogar gewollt sind, weil sie durch Verbesserung des Codes entstanden sind. Im letzten Fall müssen die entsprechenden Referenzdaten ersetzt werden. Somit hilft das Validierungssystem auch dabei zu kontrollieren, ob die unternommenen Codeänderungen, welche die Ergebnisse eines Szenarios verbessern, nicht unerwünscht auch andere Teile des Codes beeinflussen.
Das Programm zum „intelligenten“ Vergleich von Ausgabedaten - „ERO.ETT“ - wurde implementiert, um die extensiven ERO 3D-Volumen- und 2D-Oberflächen-Daten auf charakteristische Werte, beispielsweise durch Integration oder durch Berechnung der mittleren Eindringtiefe, zu reduzieren. Der statistische Fehler solcher Werte ist offensichtlich viel kleiner als bei den Werten einer einzelnen Zelle. ERO.ETT kontrolliert automatisch, ob der Unterschied zwischen Test- und Referenz-Daten in Rahmen eines benutzerdefinierten Toleranz-Bereichs liegt. Zudem werden auch generelle Konsistenz-Tests durchgeführt.
Das Programm liefert einen Ergebnisbericht des Vergleiches im ASCII Format und ist in der Lage, automatisch E-Mails an den Benutzer zu senden.

Um die Arbeit des Validierungssystems zu illustrieren wird ein Testfall (Kap. 4.1) - das Debugging der Parallelisierung eines Moduls zur Berechnung von physikalischer Erosion - präsentiert. Es wird demonstriert, wie das automatische System dabei hilft Fehler im Code zu entdecken und zu lokalisieren. Das System kann auch für Effektivitätsanalysen („Speedup“) von parallelisierter Software genutzt werden.
Das System wurde auf die Supercomputer JUMP, JUROPA/HPC-FF und diverse Linux Workstations portiert, dort eingerichtet und getestet. Konfigurationsdateien und Templates für die Kompilierung, Vorbereitung von Eingabedaten und Ausführung wurden konzipiert und erstellt.
Insgesamt lässt sich festhalten, dass die vom ITM Projekt erhobenen Anforderungen an ein automatisches Validierungssystem erfüllt wurden. Durch die hohe Flexibilität und leichte Konfigurierbarkeit von JuBE lassen sich andere, innerhalb des ITM Projektes entwickelte, Modellierungs-Codes einfach in das System integrieren. Die entsprechende Programme (wie ERO.ETT für ERO) zum Vergleich der Ausgabedaten müssen jedoch entwickelt werden, weil sie offensichtlich einzigartig für jede Anwendung sind.

5.2 Ausblick

Das zunächst für ERO entwickelte Validierungssystem muss für weitere ITM-Codes angepasst werden. Dazu sollten entsprechende Tools zum Vergleich von Ausgabedaten, ähnlich zu ERO.ETT entwickelt und weitere Test-Szenarien in JuBE konfiguriert werden.
In einigen Aspekten muss ERO und das Validierungssystem entsprechend den ITM Vorschriften verallgemeinert werden. Beispielsweise muss der Datenaustausch zwischen den Codes in Form von CPOs [14] organisiert werden. Die Synchronisation zwischen Rechnungen und Datenübergabe zwischen den Codes muss mit KEPLER Workflows [12] gesteuert werden.
Die Parallelisierung von ERO muss weiter optimiert werden. Im behandelten „Case 6 “ bedeutet ein Speedup von ~ 2.4 bei 8 eingesetzten Threads eine Effizienz von lediglich 30%. Dazu bietet es sich an, diesen Fall mit dem Tool SCALASCA zu analysieren [21]. Diese Software gibt wertvolle Information über „Performance-Bottlenecks“ und Wartezeiten der beteiligten Threads.

Literaturverzeichnis

[1] A. Schnurpfeil S. Meier W. Frings L. Arnold. *JuBE: Jülich Benchmarking Environment*. Forschungszentrum Jülich. http://www.fz-juelich.de/jsc/jube.

[2] Unbekannter Author. *Bindungsenergie-Massenzahl*. 2010. http://commons.wikimedia.org/wiki/File:Bindungsenergie_Massenzahl.jpg#filelinks.

[3] Unbekannter Author. *Experimental devices: Tokamak*. 2010. http://leifi.physik.uni-muenchen.de/web_ph09_g8/umwelt_technik/09fusion/tokamak.htm.

[4] Unbekannter Author. *Experimental devices: Tokamak*. 2010. http://www.ipp.mpg.de/ippcms/eng/pr/exptypen/tokamak/index.html.

[5] Unbekannter Author. *Magnetisch eingeschlossene Fusionsplasmen*. 2010. http://www.dpg-physik.de/dpg/gliederung/fv/p/info/magnet.html.

[6] D. Borodin, A. Kirschner, A. Kreter, V. Philipps, A. Pospieszczyk, R. Ding, R. Doerner, D. Nishijima, and J. Yu. Modelling of Be transport in PSI experiments at PISCES-B. *Journal of Nuclear Materials*, 390-391:106, 2009.

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

[8] Ulrich Detert. *Experimental devices: Tokamak*. 2010. http://www.fz-juelich.de/jsc/juropa/configuration/.

[9] R. Ding, A. Kirschner, D. Borodin, S. Brezinsek, A. Pospieszczyk, O. Schmitz, V. Philipps, U. Samm, J. Chen, and J. Li. Simulation of light emission from hydrocarbon injection in TEXTOR using the ERO code. *Plasma Physics and Controlled Fusion*, 51:12, 2009.

[10] S. Droste, D. Borodin, A. Kirschner, A. Kreter, V. Philipps, and U. Samm. Impurity Transport Modelling in Edge Plasmas of Fusion Devices with the Monte Carlo Code ERO. *Contributions to Plasma Physics*, 7-9:628, 2006.

[11] Andreas Kirschner et al. Hydrocarbon transport in the MkIIa divertor of JET. *Plasma Physics and Controlled Fusion*, 45:309, 2003.

[12] B. Guillerminet et al. Integrated tokamak modelling: Infrastructure and Software Integration Project. *Fusion Engineering and Design*, 84:442, 2008.

[13] D. Reiter et al. The EIRENE and B2-EIRENE codes. *Fusion Science and Technology*, 47:172, February 2005.

[14] F. Imbeaux et al. Data Structure for the European Integrated Tokamak Modelling Task Force. 35th EPS Conference on Plasma Phys. Hersonissos, 2008.

[15] F.W. Perkins et al. ITER Physics Basis. *Nuclear Fusion*, 39, 1999.

[16] H.P. Summers et al. The Atomic Data and Analysis Structure. *Nuclear Fusion Research*, 78, 2005.

[17] A. Kirschner et.al. Simulation of the plasma-wall interaction in a tokamak with the Monte Carlo code ERO-TEXTOR. *Nuclear Fusion*, 40:989, 2000.

[18] Andreas Galonska. Parallelisierung und Optimierung eines Monte-Carlo-Transportmodells für Kernfusionsanwendungen. Technical report, Jülich Supercomputing Center. IB-2008-08.

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

[20] Forschungszentrum Jülich. *JuRoPA System Configuration*. `http://www.fz-juelich.de/jsc/juropa/configuration`.

[21] Forschungszentrum Jülich. *Scalasca - SCalable performance Analysis of LArge SCale Applications*. `http://www.fz-juelich.de/jsc/scalasca`.

[22] 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`.

[23] A. Kirschner, D. Borodin, S. Droste, V. Philipps, U. Samm, G. Federici, A. Kukushkin, and A. Loarte. Modelling of tritium retention and target lifetime of the ITER divertor using the ERO code. *Journal of Nuclear Materials*, 363-365:91, 2007.

[24] U. Kögler and J. Winter. ERO-TEXTOR - 3D-Monte-Carlo Code for Local Impurity-Modeling in the Scrape-Off-Layer of TEXTOR. Technical report, Institut für Plasmaphysik - Forschungszentrum Jülich, 1997. Jül-Report 3361.

[25] Grant McLean. *XML-Simple*. CPAN. `http://search.cpan.org/dist/XML-Simple/`.

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

[27] D. Naujoks, R. Behrisch, J.P. Coad, and L.deKock. Material transport by erosion and redeposition on surface probes in the scrape-off layer of JET. *Nuclear Fusion*, 33:581, 1993.

[28] 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`.

[29] U. Samm. TEXTOR: a pioneering device for new concepts in plasma-wall interaction, exhaust, and confinement. *Fusion Science and Technology*, 47:73, 2005.

[30] Horst Stöcker. *Taschenbuch der Physik*. Verlag Harri Deutsch, 2007.

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

[32] W3C. *Extensible Markup Language (XML)*. `http://www.w3.org/XML`.

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

Anhang A

Testreports

In diesem Anhang befinden sich 2 Testreports, welche die Arbeit des Validierungssystems illustrieren. Es wird jeweils ein Beispiel für einen fehlgeschlagenen und einen erfolgreichen Test gegeben.

Listing A.1: Ausgabe von ERO.ETT - Fehlgeschlagener Test von Case 6

```
**************************************
*          ERO TESTING TOOL           *
**************************************


===Directory 1===
Directory:/lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/
    CASE6500

Emission File
Name: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/CASE
    6500/BiasPot1p2_Emission_step0.m
Elements:
  be__ (+0)
  be__ (+1)
  be__ (+2)




Surface File
Name: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/CASE
    6500/BiasPot1p2_Surface_step0.m
Information:
  Cell Features = 17
  Elements = 1
  iSType = 6
  iTimeStepNo = 0
  dStepTime = 1
  dRealTime = 1
```

```
  dTotalTime = 1
  dXmax = 18
  dXmin = -18
  dYmax = 18
  dYmin = -18
  dDeltaX = 9
  dDeltaY = 9
  iNX = 1
  iNY = 12
Elements:
  be__




===Directory 2===
Directory:/lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/tmp
    /ERO_Intel-Nehalem-HPC-FF_CASE6_500_i000097/n1p1t4_t001_i01

Emission File
Name: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/tmp/ERO
    _Intel-Nehalem-HPC-FF_CASE6_500_i000097/n1p1t4_t001_i01/BiasPot1p
    2_Emission_step0.m
Elements:
  be__ (+0)
  be__ (+1)
  be__ (+2)




Surface File
Name: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/tmp/ERO
    _Intel-Nehalem-HPC-FF_CASE6_500_i000097/n1p1t4_t001_i01/BiasPot1p
    2_Surface_step0.m
Information:
  Cell Features = 17
  Elements = 1
  iSType = 6
  iTimeStepNo = 0
  dStepTime = 1
  dRealTime = 1
  dTotalTime = 1
  dXmax = 18
  dXmin = -18
  dYmax = 18
  dYmin = -18
  dDeltaX = 9
  dDeltaY = 9
  iNX = 1
```

```
  iNY = 12
Elements:
  be__




Relative Error: 1.000000e-01

ToDo-List:

  Function: CmpSum( AUTOSTRVAR = 'all'; AUTOSTRVAR = 'dens'; )

  Function: CmpSurf( AUTOSTRVAR = 'all'; )

  Function: CmpAvgPenDep( AUTOSTRVAR = 'be__'; AUTOFLTVAR = 0;
      AUTOSTRVAR = 'xy'; AUTOSTRVAR = 'dens'; AUTOFLTVAR = 0;
      AUTOFLTVAR = 0; AUTOFLTVAR = 20; AUTOFLTVAR = 100; AUTOFLTVAR =
      -90.1; AUTOFLTVAR = 10; )



**************************************
* FAILED CHECK!!!!!
* Checking <Compare Sum of be__ (+0)> failed!
* File: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/tmp/
    ERO_Intel-Nehalem-HPC-FF_CASE6_500_i000097/n1p1t4_t001_i01/BiasPot
    1p2_Emission_step0.m
* ValueName: dens
* ValueReference: 4.801962e+13
* ValueProbe:  1.674022e+13
**************************************
**************************************
* FAILED CHECK!!!!!
* Checking <Compare Sum of be__ (+1)> failed!
* File: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/tmp/
    ERO_Intel-Nehalem-HPC-FF_CASE6_500_i000097/n1p1t4_t001_i01/BiasPot
    1p2_Emission_step0.m
* ValueName: dens
* ValueReference: 2.405928e+12
* ValueProbe:  1.125354e+12
**************************************
**************************************
* FAILED CHECK!!!!!
* Checking <Compare Sum of be__ (+2)> failed!
* File: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/tmp/
    ERO_Intel-Nehalem-HPC-FF_CASE6_500_i000097/n1p1t4_t001_i01/BiasPot
    1p2_Emission_step0.m
* ValueName: dens
* ValueReference: 3.870750e+09
```

```
* ValueProbe:  0.000000e+00
**************************************
**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of NG     > successful!
**************************************

**************************************
* FAILED CHECK!!!!!
* Checking <Compare StatusSums> failed!
* File: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/CASE
    6500/BiasPot1p2_Surface_step0.m
* ValueName: be__ -> NC
* ValueReference: 1.133950e+14
* ValueProbe:  5.497480e+13
**************************************
**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of _T     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of _R     > successful!
**************************************

**************************************
* FAILED CHECK!!!!!
* Checking <Compare StatusSums> failed!
* File: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/CASE
    6500/BiasPot1p2_Surface_step0.m
* ValueName: be__ -> ME
* ValueReference: 1.139051e+03
* ValueProbe:  8.255660e+02
**************************************
**************************************
* FAILED CHECK!!!!!
* Checking <Compare StatusSums> failed!
* File: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/CASE
    6500/BiasPot1p2_Surface_step0.m
* ValueName: be__ -> MQ
* ValueReference: 1.200000e+01
* ValueProbe:  9.000000e+00
**************************************
**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
```

```
* Checking <Compare StatusSums of SZ     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of UC     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of PQ     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of PD     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of PD_CHEM> successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of BG     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of SH     > successful!
**************************************

**************************************
* FAILED CHECK!!!!!
* Checking <Compare StatusSums> failed!
* File: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/CASE
    6500/BiasPot1p2_Surface_step0.m
* ValueName: be__ -> FLC
* ValueReference: 1.144631e+14
* ValueProbe:  5.548720e+13
**************************************
**************************************
* SUCCESSFUL CHECK!!!!!
```

```
* Element: be__
* Checking <Compare StatusSums of CER    > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of PER    > successful!
**************************************

**************************************
* FAILED CHECK!!!!!
* Checking <Compare StatusSums> failed!
* File: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/CASE
    6500/BiasPot1p2_Surface_step0.m
* ValueName: be__ -> PER6
* ValueReference: 7.709860e+12
* ValueProbe:  3.626950e+12
**************************************
**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__ (0)
* Checking <Compare Penetration Depth of dens> successful!
**************************************

**************************************
*          Completed!!!              *
**************************************
--- CHECK FAILED ---
```

Listing A.2: Ausgabe von ERO.ETT - Erfolgreicher Test von Case 6

```
**************************************
*          ERO TESTING TOOL          *
**************************************


===Directory 1===
Directory:/lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/
    CASE6500

Emission File
Name: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/CASE
    6500/BiasPot1p2_Emission_step0.m
Elements:
  be__ (+0)
  be__ (+1)
  be__ (+2)
```

```
Surface File
Name: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/CASE
    6500/BiasPot1p2_Surface_step0.m
Information:
  Cell Features = 17
  Elements = 1
  iSType = 6
  iTimeStepNo = 0
  dStepTime = 1
  dRealTime = 1
  dTotalTime = 1
  dXmax = 18
  dXmin = -18
  dYmax = 18
  dYmin = -18
  dDeltaX = 9
  dDeltaY = 9
  iNX = 1
  iNY = 12
Elements:
  be__




===Directory 2===
Directory:/lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/tmp
    /ERO_Intel-Nehalem-HPC-FF_CASE6_500_i000095/n1p1t4_t001_i01

Emission File
Name: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/tmp/ERO
    _Intel-Nehalem-HPC-FF_CASE6_500_i000095/n1p1t4_t001_i01/BiasPot1p
    2_Emission_step0.m
Elements:
  be__ (+0)
  be__ (+1)
  be__ (+2)




Surface File
Name: /lustre/jhome8/fsnitmi/fsitmi02/ero/JuBE/applications/ERO/tmp/ERO
    _Intel-Nehalem-HPC-FF_CASE6_500_i000095/n1p1t4_t001_i01/BiasPot1p
    2_Surface_step0.m
Information:
  Cell Features = 17
  Elements = 1
```

```
  iSType = 6
  iTimeStepNo = 0
  dStepTime = 1
  dRealTime = 1
  dTotalTime = 1
  dXmax = 18
  dXmin = -18
  dYmax = 18
  dYmin = -18
  dDeltaX = 9
  dDeltaY = 9
  iNX = 1
  iNY = 12
Elements:
  be__



Relative Error: 1.000000e-02

ToDo-List:

  Function: CmpSum( AUTOSTRVAR = 'all'; AUTOSTRVAR = 'dens'; )

  Function: CmpSurf( AUTOSTRVAR = 'all'; )

  Function: CmpAvgPenDep( AUTOSTRVAR = 'be__'; AUTOFLTVAR = 0;
      AUTOSTRVAR = 'xy'; AUTOSTRVAR = 'dens'; AUTOFLTVAR = 0;
      AUTOFLTVAR = 0; AUTOFLTVAR = 20; AUTOFLTVAR = 100; AUTOFLTVAR =
      -90.1; AUTOFLTVAR = 10; )



**************************************
* SUCCESSFUL CHECK!!!!!
* Element: all
* Checking <Compare Sum of dens> successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: all
* Checking <Compare Sum of dens> successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: all
* Checking <Compare Sum of dens> successful!
```

```
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: all
* Checking <Compare common Surface Data> successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: all
* Checking <Compare Cell Properties> successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of NG     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of NC     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of _T     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of _R     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of ME     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of MQ     > successful!
**************************************
```

```
**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of SZ     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of UC     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of PQ     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of PD     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of PD_CHEM> successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of BG     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of SH     > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of FLC    > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
```

```
* Element: be__
* Checking <Compare StatusSums of CER    > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of PER    > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__
* Checking <Compare StatusSums of PER6   > successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: all
* Checking <Compare Surface> successful!
**************************************

**************************************
* SUCCESSFUL CHECK!!!!!
* Element: be__ (0)
* Checking <Compare Penetration Depth of dens> successful!
**************************************

**************************************
*          Completed!!!               *
**************************************
--- CHECK OK   ---
```

Danksagung

An erster Stelle danke ich Herrn Prof. Dr. M. Reißel, der mir die Möglichkeit zur Bearbeitung dieser interessanten Aufgabenstellung gab und die Arbeit betreute. Er war es auch, der mir während meines Studiums, die Techniken der Numerik zur Lösung komplexer wissenschaftlicher Probleme, näher brachte.

Bei Herrn Dr. D. Borodin möchte ich mich für die fachliche Betreuung bedanken. Durch ihn habe ich bei zahlreichen Gelegenheiten einen wunderbaren Überblick über den derzeitigen Stand der Fusionsforschung und die dahinter steckende Physik bekommen. Als Entwickler des ERO Codes gab er mir Orientierung im scheinbar undurchdringlichen Wald physikalischer Rechnungen, auf denen dieser Code basiert. Ich erhielt von ihm außerdem viele wertvolle Tipps zur formalen und inhaltlichen Gestaltung der Arbeit. Mit seiner Erfahrung und seinem Wissen stand er mir jederzeit helfend zur Seite.

Herrn Dr. Andreas Kirschner möchte ich ebenfalls für das physikalische Hintergrundwissen und die, ERO betreffenden, programmtechnischen Details, die er mir erläutert hat, bedanken. Sein Humor war immer eine willkommene Ablenkung für mich.

Ich danke Herrn Dr. Paul Gibbon, daß er mich in seinem Simulation-Lab Plasma-Physik aufgenommen und betreut und zudem die Finanzierung dieser Arbeit sicher gestellt hat. Durch ihn bin ich auch das erste Mal in Berührung mit der Plasmaphysik und der Fusionsforschung gekommen, da er den Kontakt zum IEF-4 hergestellt hat. Er war aus formaler und auch technischer Sicht ein sehr guter Ansprechpartner und hat immer Zeit gefunden, sich um meine Belange zu kümmern, was mir insbesondere auch bei der Anfertigung meiner vorhergegangenen Diplomarbeit von großer Hilfe war.

Herrn Dr. L. Arnold danke ich für die Unterstützung bei der Konfiguration von JuBE und für das Co-Referat bei der Vorstellung des automatischen Validierungssystems auf dem ITM Meeting im September 2009.

Herrn Wolfgang Frings und Frau Stefanie Meier möchte ich für die Entwicklung von JuBE danken, da ohne diese Basis der Rahmen einer Masterarbeit wahrscheinlich gesprengt worden wäre.

Herrn Prof. Gürich und Herrn Dr.Matthias Bolten danke ich für die Chance, mein Können im Jülicher Supercomputing Center unter Beweis zu stellen. Ohne sie wäre mein Einstieg in das Supercomputing ungleich schwieriger gewesen. Ihnen verdanke ich den Großteil meines Wissens über parallele Rechnerarchitekturen und die zugrunde liegende Programmiertechnik.

Herrn Günter Egerer danke ich besonders für seine Unterstützung bei der Programmierung in C/C++. Wenn ich Probleme bei der Programmierung oder Übersetzung das Programms hatte, stand er mir immer hilfreich zur Seite.

Dem JSC insgesamt danke ich für die Bereitstellung und den Betrieb der Supercomputer, die ich sowohl bei meiner Diplom- als auch bei meiner Masterarbeit einsetzen konnte und welche großartige Voraussetzungen für meine Simulationen boten.

Meiner gesamten Familie möchte ich für die großartige Unterstützung danken. Besonderer Dank geht hierbei an meine Großeltern, die mir durch ihre finanzielle Unterstützung die Weiterführung meines Studiums erst ermöglichten, sowie an meinen Vater und seine Frau, die mich immer motivieren konnten und mir über die Jahre ein sicheres Domizil boten.

Jül-4320
März 2010
ISSN 0944-2952