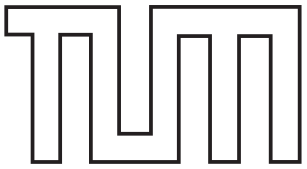
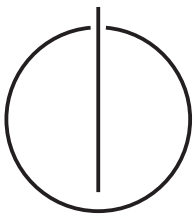


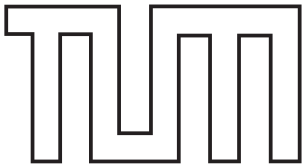
Fakultät für Informatik
der Technischen Universität München

Bachelorarbeit in Informatik

# Entwicklung eines skalierbaren Checkpoint-Restart Schemas mit parallelem File-I/O

Andreas Stephan





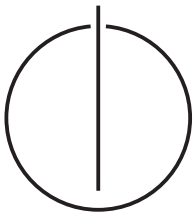
Fakultät für Informatik
der Technischen Universität München

Bachelor's Thesis in Informatik

# Entwicklung eines skalierbaren Checkpoint-Restart Schemas mit parallelem File-I/O

# Development of a scalable checkpoint-restart scheme with parallel file I/O

| | |
|---|---|
| Bearbeiter: | Andreas Stephan |
| Aufgabensteller: | Univ.-Prof. Dr. Hans-Joachim Bungartz |
| Betreuer: | Dipl.-Inf. Wolfgang Eckhardt |
| Abgabedatum: | 15. September 2014 |



Ich versichere, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.09.2014 .....

## Zusammenfassung

Die Molekulardynamik ist ein wichtiger Bereich in der Forschung. Sie bedient sich großer Rechenzentren oder Supercomputer, um ihre Simulationen durchzuführen. Bei solchen Simulationen entstehen riesige Datenmengen, wodurch eine effiziente Ein- und Ausgabe notwendig wird.

Diese Arbeit beschäftigt sich damit, unter Ausnutzung von parallelen Dateisystemen einen Checkpoint-Restart-Mechanismus für das Molekulardynamikprogramm MarDyn zu entwickeln, weil der bislang vorhandene Mechanismus für größere Simulationsläufe so lange dauert, dass er nicht benutzbar ist.

Zunächst haben wir eine Evaluierung der gängigen parallelen I/O-Bibliotheken vorgenommen. Danach haben wir eine Implementierung mit den Bibliotheken PnetCDF und MPI-IO realisiert. Die Grundidee dahinter ist, ein Dateiformat aufzubauen, das eine Zellstruktur ähnlich der Datenstruktur des Linked-Cells-Algorithmus verwendet.

Die Laufzeiten, die wir mit unserer Implementierung erreichen konnten, zeigen, dass es jetzt möglich ist, einen Checkpoint-Restart effizient auszuführen. Auch bei großen Tests mit 33 Millionen Molekülen und 16.000 Prozessen dauert das Schreiben eines Checkpoints nur wenige Sekunden.

# Inhaltsverzeichnis

# 1 Einleitung

Schon seit geraumer Zeit werden für wissenschaftliche Zwecke Computersimulationen entwickelt. Eine Richtung ist die Molekulardynamik. Sie beschäftigt sich mit der Simulation von Wechselwirkungen zwischen Molekülen. In der Forschung werden diese oft verwendet, um real nicht durchführbare oder zu teure Experimente durchzuführen.

Schon lange sind dies keine sequentiellen Programme mehr. Um immer mehr zu erfahren, ist es nur natürlich, dass die Simulationsläufe immer größer und genauer werden. Die Simulationen werden mittlerweile auf großen Rechenclustern oder auf Supercomputern eingesetzt.

Die Datenmengen, die bei solchem Simulationen entstehen, werden immer umfangreicher. Die Anforderungen an eine effiziente Ein-/Ausgabe(I/O) wachsen ständig. Die Ausnutzung von parallelen Dateisystemen erlaubt es, mehrere Datenleitungen zu benutzen. Dadurch kann man die Effizienz der Ein- und Ausgabe erheblich erhöhen.

Leider werden parallele Dateisysteme und damit möglicher paralleler I/O viel zu selten eingesetzt. So auch bei MarDyn, das Molekulardynamikprogramm, mit dem sich diese Arbeit beschäftigt. MarDyn hat einen eingebauten Checkpoint-Restart-Mechanismus, der es erlaubt, die Simulation nach einem Absturz oder nach einer Pause neu starten oder weiterlaufen zu lassen. Der Mechanismus benötigt leider schon ab einer Simulationsgröße von 16000 Prozessoren so lange, dass es nicht mehr praktikabel ist, ihn zu benutzen (Genaue Zeiten in Kapitel 5).

Die Arbeit greift diese Thematik auf. Wir versuchen für MarDyn einen Checkpoint-Restart zu implementieren, dessen Benutzung auch für größere Läufe sinnvoll ist.
Wir werden eine parallelen I/O-Bibliothek nutzen, um den I/O-Mechanismus zu erstellen. Um eine Ordnung in der Checkpoint-Datei zu erstellen, werden wir eine Datenstruktur aufbauen, um die Partikel zu speichern.

In Kapitel 2 werden wir die Grundlagen klären. Zuerst wird parallele Ein-/Ausgabe eingeführt. Im Anschluss gibt es eine kleine Einführung in die Molekulardynamik. Es folgt eine kurze Vorstellung des Simulationprogramms MarDyn und in dessen I/O-Systems.

In Kapitel 3 werden die Vorbereitungen für die Implementierung angegeben. Nachdem wir die bestehenden Vorgaben geklärt haben, werden wir das Dateiformat besprechen. Im Anschluss werden wir die Auswahl treffen, welche Bibliothek für parallelen I/O wir verwenden wollen. Um unsere Implementierung später besser mit der Bestehenden vergleichen zu können, stellen wir den momentan existierenden ASCII-I/O auf binär um.

In Kapitel 4 werden wir unsere Implementierung besprechen. Weil genug Zeit vorhanden war, konnten wir eine Umsetzung mit PnetCDF und MPI-IO erreichen. Im Anschluss

wird angegeben, wie die neuen I/O-Mechanismen zu benutzen sind.

In Kapitel 5 gibt es eine Evaluierung der Ergebnisse. Zunächst wird der MPI-IO-Mechanismus intensiver besprochen. Anschließend wird ein Vergleich zwischen der existierenden und den neuen Implementierungen durchgeführt.

| Anwendung | | | | |
|---|---|---|---|---|
| Abstrahierende Bibliothek | HDF5 | | NetCDF/PnetCDF | |
| Hardwarenahe Bibliothek | MPI-IO | | | |
| Paralleles Dateisystem | Lustre | GPFS | PanFS | PVFS |
| I/O Hardware | | | | |

Tabelle 1: Die Schichten des parallelen I/O

# 2 Grundlagen

## 2.1 Einführung in parallelen I/O

Wir geben nunmehr eine Einführung in parallele Ein-/Ausgabe(I/O) [CCLC07]. In Abbildung 1 sind die Komponenten des parallelen I/O erkennbar.
Nötig ist ein paralleles Dateisystem. Ohne ein solches wäre kein paralleler I/O möglich. Typischerweise befinden sich diese auf großen Rechencluster oder Supercomputer. Einige bekannte Vertreter solcher Dateisysteme sind Lustre [Lus], GPFS [SH02], PanFS [WUA+08] und das PVFS [CLRT00].

Die zwei darüberliegenden Schichten der Tabelle 1 zeigen Bibliotheken, mit denen man den parallelen I/O nutzbar machen kann. Diese werden in Kapitel 3.3 bei der Evaluierung, welche Bibliothek wir benutzen wollen, beschrieben.

Bei sequentiellen I/O gibt es typischerweise verschiedene Modi. Beispiele sind das Anhängen von Daten an das Dateiende, oder das Leeren der Datei vor dem Schreiben. Dies ist meistens sehr einfach und es reicht den passenden Befehl im Programm zu übergeben.

Verschiedene Modi gibt es bei parallelem I/O ebenso. Sie gestalten sich aber etwas komplexer.
Das einfachste Verfahren ist ein unabhängiger Dateizugriff. Das bedeutet, dass jeder Prozess seine eigene Datei zum Lesen oder Schreiben hat. Dieser Modus ist recht simpel zu benutzen. Für viele Anwendungen ist er allerdings unpraktisch, weil man die Daten im Nachhinein zusammenführen oder weiterverarbeiten muss.

Es ist öfter der Fall, dass mehrere Prozesse eine I/O-Operation auf der selben Datei ausführen. Bei Lesezugriffen ist das kein Problem. Beim Schreiben muss man allerdings aufpassen, weil sich die Daten der verschiedenen Prozesse überschreiben können. Man muss also den Speicherraum der Datei auf die Prozesse aufteilen. Dies erhöht die Komplexität und erfordert Kommunikation.

Viele parallele I/O-Bibliotheken bieten sogenannte kollektive Operationen an. Auch MPI verwendet diesen Begriff, um eine gemeinsame Operation mehrerer Prozessoren zu deklarieren. Bei I/O-Operationen bedeutet kollektiv, dass die in einer Gruppe beteiligten Prozesse einen gemeinsamen Dateizugriff durchführen. Jeder Prozess in der Gruppe muss

die selbe Operation aufrufen, wodurch die Bibliothek intern gegebenenfalls Optimierungen vornehmen kann.

## 2.2 Einführung in die Molekulardynamik

Im folgenden wird eine kurze Einführung in die Molekulardynamik(MD) gegeben. Sie beschäftigt sich damit, Wechselwirkungen zwischen Molekülen in einem bestimmten Molekülmodell zu simulieren. Mehr Informationen finden sich im Buch von Griebel [GKZ07]. In der Regel basiert eine MD-Simulation auf den Newtonschen Bewegungsgleichungen. Durch diese ist es möglich, die zeitliche Veränderung zu beschreiben. Dazu werden die Partikel mit den Eigenschaften Position, Geschwindigkeit, Masse und Ladung beschrieben.

Benutzt werden MD-Simulationen in vielen Bereichen: In der Chemie testet man auf die Eigenschaften von Stoffen. Die Materialwissenschaften benutzen sie, um neue Stoffe zu entwickeln. Festere Strukturen wie zum Beispiel Zellen werden in der Biologie untersucht.

Häufig werden kurzreichweitige Potentiale angewendet, um die Wechselwirkungen zwischen den Molekülen zu berechnen. Der Vorteil von kurzreichweitigen Potentialen ist, dass die Wechselwirkungen zwischen zwei Molekülen bei größerem Abstand sich stark verkleinern. Einer der bekanntesten Vertreter von kurzreichweitigen Potentialen ist das Lennard-Jones Potential. Es ergibt sich folgende Kraftformel für die Moleküle $i, j$:

$$F_{ij} = 24\epsilon \frac{1}{r_{ij}^2} \left[ \left( \frac{\sigma}{r_{ij}} \right)^6 - 2 \left( \frac{\sigma}{r_{ij})} \right)^{12} \right] r_{ji}$$

Um die Laufzeit zu optimieren, nimmt man in Kauf, eine Wechselwirkung $F_{ij}$ ab einem gewissen Abstand nicht mehr zu berechnen. Dieser Abstand wird als Cutoffradius bezeichnet. Der Vorteil von kurzreichweitigen Potentialen ist, dass das Ergebnis schon ab einem relativ kleinen Cutoffradius nicht verfälscht wird.

Um die neue Position und die neue Geschwindigkeit zu berechnen, gibt es sogenannte Zeitintegrationsschemata. Ein bekannter Vertreter ist das Leapfrog-Schema. Nachstehende Formeln werden für Position $x$ und Geschwindigkeit $v$ mithilfe der Beschleunigung $a$ verwendet.

$$x_{i+1} = x_i + v_{i+\frac{1}{2}} \delta t$$
$$v_{i+\frac{3}{2}} = v_{i+\frac{1}{2}} + a \delta t$$

Das Weglassen der Berechnungen zwischen zwei Partikeln, wenn sie weiter als der Cutoffradius voneinander entfernt sind, ist die Motivation für den Linked-Cells-Algorithmus. Er teilt die Simulationdomain in ein gleichmäßiges Gitter, wobei die Kantenlängen dem Cutoffradius entsprechen. Die einzelnen Gebiete in dem Gitter werden Zellen genannt. Ein Beispielausschnitt einer Domain ist in Abbildung 1. Wie schon gesagt, wird die Kraft $F_{ij}$ nur berechnet, wenn der Abstand zwischen den Partikeln $i$ und $j$ kleiner als der

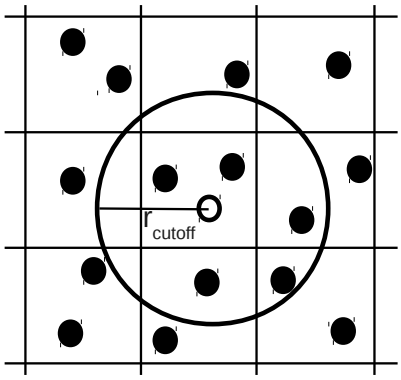



Abbildung 1: Ein Teil der Domain eingeteilt in die Zellen des Linked-Cells Algorithmus

Cutoffradius ist. In der Abbildung muss das mittlere Partikel nur die Wechselwirkungen mit den Partikeln innerhalb des Kreises berechnen.
Die naive Variante iteriert über alle Partikel und testet, welche anderen Partikel in deren Cutoffradius liegen. Wenn $n$ die Anzahl an Partikeln ist, würde man so eine Laufzeit von $\mathcal{O}(n^2)$ haben. Da die Kantenlängen dem Cutoffradius entsprechen, muss beim Linked-Cells-Algorithmus nur in den benachbarten Zellen geprüft werden, ob Partikel im Cutoffradius des aktuellen Partikels liegen. Somit erreicht der Linked-Cells Algorithmus eine verbesserte Laufzeit von $\mathcal{O}(n)$.

## 2.3 MarDyn

MarDyn[Buc10] ist eine MD-Simulation, die erstellt wurde, um Nanofluide zu untersuchen. Sie ist sehr modular aufgebaut und kann verschiedene Partikelbehälter verwenden. MarDyn unterstützt mehrzentrige Molekülmodelle. Die Molekülarten können unterschiedliche Molekülzentren besitzen. Eine Molekülart wird in der Simulation als Komponente bezeichnet. Die Position der einzelnen Molekülzentren wird relativ zum Zentrum der Komponente gespeichert. Die Positionen innerhalb der Komponente sind fest.

Ein Beispiel der Vorgehensweise zur Parallelisierung in MarDyn ist in Abbildung 2 zu sehen. Jedem Prozess wird ein Teil der Domain zugewiesen. Im Beispiel wird die Domain einfach geviertelt. In MarDyn hat jeder Partikelbehälter eine Zellstruktur. Für die Parallelisierung gibt es verschiedene Arten von Zellen. Es gibt Zellen, die in dem Simulationsbereich liegen, der dem Prozess zugewiesen ist. Weil man die direkt benachbarten Zellen für die Kraftberechnung benötigt, werden Zellen von den anliegenden Prozessbereichen kopiert. Diese Zellen werden als Halozellen bezeichnet. In Abbildung 2 werden Halozellen grau dargestellt. Die zu kopierenden Randzellen sind hellblau und die normalen Zellen dunkelblau. Der rechte Teil zeigt, nach welcher Logik die Zellen ausgetauscht werden.

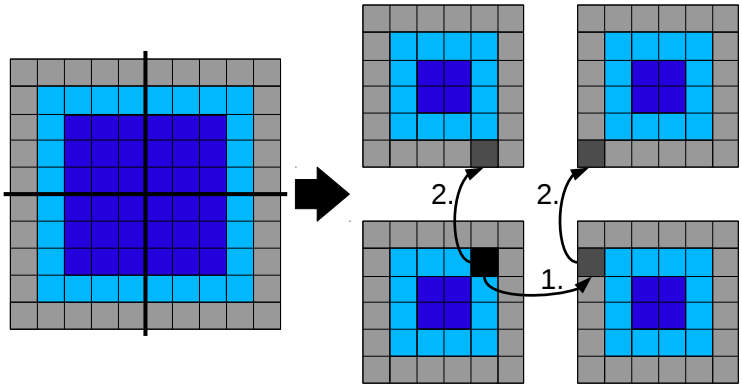



Abbildung 2: Komplette Domain links. Rechts Austauch einer Zelle in den Halobereich der benachbarten Prozesse

Nun wird auf die Teile von MarDyn eingegangen, die in dieser Arbeit benutzt oder verändert werden.

MarDyn hat verschiedene DomainDekompositionen implementiert. Deren Aufgabe ist es, den einzelnen Prozessoren einen Teilbereich der Domain zuzuweisen. Als Domain bezeichnen wir in dieser Arbeit den Simulationsraum. Die Domaindekompositionen sind außerdem für die Kommunikation und das Austauschen der Moleküle zwischen den Prozessoren zuständig.
Die abstrakte Klasse, die als Schema für eine solche Domaindekomposition implementiert wurde, heißt *DomainDecompBase*. Von ihr erbt die Klasse *DomainDecompDummy*, die für eine sequentielle Kompilierung zuständig ist, weshalb hier nichts passiert. Eine andere erbende Klasse ist die *DomainDecomposition*. Sie teilt die Domain, die bis jetzt immer ein Quader ist, in gleich große Teile, sodass jeder Prozess einen identisch großen Simulationsbereich erhält. Die *KDDomainDecomposition* teilt die Domain in kleine Quader der selben Größe auf. Dann führt sie eine Lastverteilung durch, indem sie jedem Prozess eine unterschiedliche Anzahl an kleinen Quadern zuweist und versucht, dass auf jeden Prozess ungefähr gleich viele Moleküle liegen.

Um die Moleküle zu speichern, gibt es in MarDyn zwei verschiedene Molekülcontainer. Der eine benutzt den im Kapitel 2 angesprochenen Linked-Cells Algorithmus und der andere passt diesen ein wenig an, indem er Zellen mit vielen Partikeln zu kleineren Zellen verfeinert.

## 2.4 Der bestehende I/O

Wir beschreiben nun das bisherige I/O Format. Das ist insofern wichtig, weil wir den bisherigen Konfigurationsmechanismus beibehalten wollen. Der Grund hierfür ist, dass der Code rückwärtskompatibel bleiben soll.

```
1 MDProjectConfig
2
3 #
4 timestepLength 0.01
5 cutoffRadius       5.
6 phaseSpaceFile OldStyle lj40000_t300.inp
7 parallelization DomainDecomposition
8 # datastructure followed by the parameters for the datastructure
9 # for LinkedCells, the cellsInCutoffRadius has to be provided
10 datastructure LinkedCells 1
11 #
12 output ResultWriter 1 mardynresult
13 output CheckpointWriter 100 default
14 output XyzWriter 100 default
15 output PovWriter 100 default
16 output VISWriter 100 default
17 output BinaryCheckpointWriter 1 binary
18 output ParallelCheckpointWirter 1 parallel
```

Abbildung 3: Ein Beispiel der Konfigurationsdatei

In MarDyn geschieht die Initialisierung über zwei verschiedene Dateien. Die Konfigurationsdatei beschreibt globale Einstellungen der Simulation. Ein Beispiel ist in Abbildung 3 zu sehen. Außerdem gibt es eine Moleküldatei. In ihr werden die Moleküldaten und die zugehörigen Einstellungen gespeichert. Ein Ausschnitt einer solchen Datei wird in Abbildung 4 gezeigt. Im Folgenden werden wir die zwei Abbildungen genauer erklären.

### 2.4.1 Konfigurationsdatei

In Zeile 1 der Konfigurationsdatei muss *MDProjectConfig* stehen. Sie enthält keine Informationen, muss aber enthalten sein. Ohne diese Zeile wird das Programm abgebrochen. Zeile 4 beschreibt die Zeitschrittlänge der Simulation. In Zeile 5 wird der CutoffRadius angegeben. Dieser wurde in Abschnitt 2 eingeführt.
Zeile 6 beschreibt den Namen der Moleküldatei. Dazu dient der Befehl *phaseSpaceFile*. Hier wird das *OldStyle*-Format benutzt, um die Daten einzulesen. In der nächsten Zeile wird die Parallelisierungsart festgelegt und geklärt, welche Domaindekomposition benutzt werden soll. Zeile 10 zeigt die Datenstruktur für die Speicherung der Moleküle auf. Von Zeile 12 bis Zeile 18 werden jeweils die Outputmodule hinzugefügt. Diese werden am Ende dieser Ausarbeitung beschrieben, da wir die parallelen Outputmodule erst noch einführen müssen.

### 2.4.2 Moleküldatei

Die erste Zeile der Moleküldatei wird wieder genutzt, um die Validität zu überprüfen. Sie setzt sich aus drei Teilen zusammen. Zu Beginn muss eines dieser sechs Wörter stehen:

```
 1 mardyn trunk 20090731
 2 currentTime     0.0
 3 #rho    0.0438273
 4 Temperature     0.7
 5 Length  97.     97.     97.
 6 NumberOfComponents      1
 7 1 0 0 0 0
 8  0. 0. 0.       10000.  1. 1. 5.0 0.0
 9  0. 0. 0.
10 1.0e+10
11 NumberOfMolecules       40000
12 MoleculeFormat  ICRV
13 1       1       75.7616 86.4547 68.2849 0.000524377 0.00150218 0.00214105
14 2       1       11.0364 46.5174 34.9379 0.00430925 0.0072667 0.00248634
15 3       1       71.8861 17.591 12.0316  0.00461185 0.00405601 0.00582299
16 4       1       22.401 44.1166 34.6829  0.00933269 -0.000957709 0.00695854
17 5       1       2.50684 77.6731 3.64561 -0.00213014 0.00755743 -0.00286593
18 6       1       11.1195 87.0263 91.7764 0.00129772 -0.00339112 -0.00257073
19 7       1       73.6955 93.751 65.8296  0.000765359 0.00815842 -0.00447441
20 8       1       9.05625 63.3589 94.142  -0.00838106 -0.00417468 -0.00211616
```

Abbildung 4: Ein Beispiel der Mölekül datei

"mardyn", "MOLDY", "ls1r1", "mrdyn", "MDProject", "MarDyn", "MARDYN".
Das zweite Wort muss "trunk" sein. Zuletzt kommt eine Versionsnummer, die größer als 20080701 sein sollte. Zeile 2 gibt an, bei welchem Zeitpunkt der Checkpoint-Restart neu gestartet werden soll. Bei einem Checkpoint-Restart gibt dies den Zeitpunkt an, an dem man aufgehört hat zu simulieren und jetzt neu startet.
In Zeile 4 steht die Temperatur. Diese ist für den Thermostat wichtig. Genauere Informationen finden sich bei Zeile 5 gibt die Domaingröße an. Die erste Zahl ist der x-Wert, die zweite der y-Wert und die dritte ist der z-Wert.
Danach werden die Molekülarten, genannt Komponenten, beschrieben. In Zeile 6 wird zuerst einmal die Anzahl an Komponenten angegeben. Für jede Komponente gibt es eine Anzahl an Lennard-Jones-Zentren, Ladungszentren, Dipolen, Quadrupolen und Tersoff-Zentren. In der Abbildung wird in Zeile 7 festgelegt, dass es ein Lennard-Jones-Zentrum, 0 Ladungen, 0 Dipole, 0 Quadrupole und 0 Tersoff-Zentren gibt.
Beispielhaft werden jetzt die Eigenschaften eines Lennard-Jones-Zentrum angeben, die in Zeile 8 zu sehen ist. Die ersten 3 Werte geben die x-,y-,z-Position relativ zum Molekülzentrum an. Die nächsten Werte sind die Masse, der Epsilonwert, der Sigmawert, der Molekülspezische Cutoffradius, der auf 0 gesetzt werden kann, und die Shift-Variable. In Zeile 9 stehen Dummy-Werte für die Komponente. Diese sind momentan nicht in Benutzung, müssen aber dennoch angegeben werden. Am Ende der Komponentenbeschreibung steht in Zeile 10 ein Parameter für die Reaction-Field Methode [BW73], der in de Regel als $\epsilon_{RF}$ bezeichnet wird. Genauere Informationen zur Nutzung der verschiedenen Komponenten finden sich bei [Buc10].
In Zeile 11 steht die Anzahl an Molekülen. Mit dieser Zahl kann man kontrollieren, wie viele Moleküle eingelesen werden sollen. In Zeile 12 steht das Molekülformat. Hier gibt es drei verschiedene: IRV, ICRV und ICRVQD. Das erste beschreibt die Molekül-ID(I), die Position(R) und die Geschwindigkeit(V). Beim nächsten kommt noch die Zuordnung zur

Komponente(C) hinzu. Bei Benutzung von ICRVQD werden auserdem die Drehorientierung(Q) und der Drehimpuls(D) angegeben.
Von Zeile 13 bis Zeile 20 sind noch 8 Moleküle im Format ICRV gezeigt. Die Reihenfolge der Parameter ist die selbe, wie in der kurz vorher vorgestellten Bedeutung der Molekülformate.

# 3 Vorbereitungen für die Implementierung

In diesem Kapitel werden wir die Vorbereitungen für die Implementierung darlegen. Zuerst werden einige Vorgaben besprochen und danach wird das Dateiformat festgelegt. Mit dem Wissen über das Format wird dann die Bibliothek ausgewählt, mit der wir die Implementierung durchführen wollen. Um einen besseren Vergleich der Ergebnisse zu bekommen, stellen wir den bisher auf ASCII-Zeichen basierenden I/O-Mechanismus auf binär um.

## 3.1 Vorgaben an den Mechanismus

Jetzt kommen ein paar Vorgaben an die Implementierung. Diese sind schon vor Beginn der Arbeit festgelegt worden.
Das bisher bestehende System aus einer Konfiguartionsdatei in Kombination mit einer Moleküldatei soll nicht verändern. Der Hauptgrund ist die dadurch sichergestellte Rückwärtskompatibilität. Durch die Trennung dieser beiden Datensätze vereinfachen wir außerdem die Erstellung des neuen I/O-Mechanismus.
Sämtliche Moleküldaten sollen sich in einer Datei befinden. Es wäre auch möglich gewesen die Daten verschiedener Prozesse in unterschiedliche Dateien zu schreiben. Dann wäre es aber schwierig rückwärtskompatibel zu bleiben.

Wir wollen nun die mögliche Auswahl an Bibliotheken einschränken. Wie wir in Abbildung 1 gesehen haben gibt es MPI-IO und die darauf aufbauenden Bibliotheken NetCDF und HDF5. Es gibt auch noch Alternativen wie SionLIB [FWP09], das am Jülich Supercomputing Centre entwickelt wurde. Wir haben uns aber entschieden nur Bibliotheken zu benutzen, die über MPI-IO auf das parallele Dateisystem zugreifen, weil diese die etablierten sind.

## 3.2 Besprechung des Dateiformats

Erst werden Punkte aufgelistet, die im Allgemeinen wichtig sind, um einen effizienten parallelen I/O zu implementieren. Im Anschluss werden wir anhand dieser Kriterien unser Dateiformat festlegen.

Es ist klar, dass wir mit unserem parallelen Dateisystem eine möglichst hohe Bandbreite erreichen wollen. Hierfür ist wichtig, dass wir eine hohe Parallelität, indem jeder Ausgabebuffer möglichst gefüllt wird. So kann das Dateisystem auch selbstständig Optimierungen vornehmen.
Beim Schreiben gibt es mehrere Möglichkeiten, um dies zu erreichen. Eine ist, dass man ein Loadbalancing vornimmt und versucht an jedem Prozess ungefähr den gleichen Datenanteil zu haben. Auch kann man die Daten zweier oder mehrerer Prozesse zusammenführen und diese zusammen in die Datei schreiben.

Beim Lesen der Datei ist das Hauptproblem, dass jedem Prozess ein gewisser Bereich der

Domain zugeordnet ist. Man hat also die Problematik, dass man eine logische Zuordnung von Daten in der Datei zu jeweiligem Prozess finden muss. Optimal wäre es hier natürlich, wenn die Zugehörigkeit der Daten zu den Prozessen disjunkt wäre, weil das Dateisystem dann vollständig parallel lesen könnte. Hinzu kommt die Schwierigkeit, dass sich die Anzahl an Prozessoren bei einem Checkpoint-Restart verändern kann.

Darauf aufbauend werden wir nun unser Dateiformat diskutieren.

Man könnte wie schon bei der bestehenden Implementierung eine ganz normale Liste an Partikeln benutzen, um die Datei zu speichern. Das Lesen der Datei würde aber problematisch, weil wir keine Zuordnung zwischen Molekül und Prozess finden können. Es würde zusätzliche Kommunikation zwischen den Prozessen erforderlich werden, welche die Komplexität erhöhen würde und auch Auswirkungen auf die Laufzeit haben sollte.

Wir haben uns gegen diese Möglichkeit entschieden. Wir suchen daher nach einer Struktur, mit der man eine Zuordnung zwischen Partikeln und Prozess finden kann.

Zuerst kam uns ein trivialer Ansatz in den Sinn: Wir weisen jedem Prozess einen Bereich in der Datei zu, der für ihn reserviert ist und wo er seine Partikel positionieren kann. Das Schreiben wäre sehr leicht zu implementieren und hätte bestimmt auch eine gute Performanz. Das parallele Dateisystem sollte Optimierungen vornehmen können, da physikalisch benachbarte Prozessoren ihre Daten nebeneinander in die Datei schreiben sollten. Wenn man mit der selben Anzahl an Prozessen die Simulation neu startet, könnte man sehr leicht eine Zuordnung finden. Problematisch wird es, wenn ein Checkpoint-Restart mit einer anderen Anzahl an Prozessoren durchgeführt wird. Dann würde es schwierig werden, eine Zuordnung zu finden, weil die Domain anders aufgeteilt wäre. Lösen könnte man das nur, indem man den Bereich jedes Prozesses speichert. Weil aber nicht jedem Prozess ein gleich großer Bereich zugeteilt ist, würde dies aufwendige Rechnungen nach sich ziehen.

Eine bessere Idee ist, eine Datenstruktur aufzubauen, wie sie der Linked-Cells-Algorithmus verwendet. Dieser wurde in Kapitel 2 eingeführt. Die Simulationsdomain ist in MarDyn immer ein Quader. Die Domain wird in gleich große Subdomains aufgeteilt. Wir nennen die Subdomains wie beim Linked-Cells-Algorithmus Zellen.
Da die Domaindekompositionen in MarDyn immer Quader als Grundeinheiten nutzen, passt diese Datenstruktur zu MarDyn. Mit Hilfe des Cutoffradius, also mit der Seitenlänge einer Subdomain, kann von einer Position im Simulationsraum auf die zugehörige Zelle des Dateiformats geschlossen werden. So ist es bei einem Neustart mit einer anderen Anzahl an Prozessoren oder einer anderer Domaindekomposition trotzdem recht einfach eine richtige Zuordnung zu finden.

Letztendlich haben wir uns für den Linked-Cells Ansatz entschieden, weil beim Einlesen die größten Freiheiten gegeben sind. Auch sollte bei einem Neustart mit gleicher Konfiguration das Einlesen genauso effizient wie die anderen Varianten sein.

```
netcdf simple_xy {
dimensions:
x = 3 ;
y = 5 ;
variables:
int vals(x, y) ;
data:
vals =
0, 1, 2, 3, 4, 5,
6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16;
}
```

(a)

```
HDF5 "dset.h5" {
      GROUP "/" {
        GROUP "MyGroup" {
          DATASET "dset" {
             DATATYPE { H5T_STD_I32BE }
             DATASPACE { SIMPLE ( 3, 5 ) / ( 3, 5 ) }
             DATA {
                 1, 2, 3, 4, 5,
                 6, 7, 8, 9, 10,
                 11, 12, 13, 14, 15
             }
          }
       }
    }
}
```

(b)

Abbildung 5: Speichern eines $3 \times 5$ Arrays. Links PnetCDF, rechts HDF5

## 3.3 Vorstellung der Bibliotheken

Wie schon besprochen werden wir uns auf MPI-IO und darauf aubauende Bibliotheken konzentrieren. Hierzu gehören zwei parallele Versionen von NetCDF [RD90] und HDF5 [HDF].

**MPI-IO** Zuerst gehen wir auf MPI-IO ein, weil sie die hardwarenäheste Bibliothek ist und die anderen auf darauf aufbauen. Seit MPI-2 ist MPI-IO ein fester Bestandteil des MPI-Standards und muss nicht extra in das Programm eingebunden werden. In MPI-IO wird immer direkt mit dem Filepointer gearbeitet. Man muss die aktuelle Position in der Datei, an der gelesen oder geschrieben werden soll, setzen. Um die Parallelität auszunutzen, kann man auf einige Funktionalitäten zugreifen, die einem bei darauf aufbauenden Bibliotheken oft abgenommen werden. Zum Beispiel kann selbst überlegt werden, ob jeder Prozess für sich arbeiten soll, oder ob ein Lese/Schreib-Vorgang kollektiv, also in gemeinsamer Abstimmung, ablaufen soll.

**PnetCDF** Die nächste Bibliothek, auf die eingegangen wird, ist PnetCDF [LLC+03]. Entwickelt wurde sie in Zusammenarbeit der Northwestern University und des Argonne National Laboratory(ANL). Sie verwendet das Format von NetCDF3. Ein Beispiel des Formats ist in Abbildung 5a angegeben.
Das Format speichert mehrdimensionale Arrays. Ein solches Array wird in der Datei als Variable bezeichnet. PnetCDF benutzt einen Headerbereich, um eine Menge an Dimensionen und eine Menge an Variablen zu speichern. Mit einer Dimension können auch mehrere Variablen beschrieben werden. Die Positionen, an welchen die Variablen positioniert werden sollen, berechnet PnetCDF selbstständig. Im Programm muss das Deklarieren des Headers und der eigentliche I/O logisch getrennt sein. Da wir nur mit einer Datei arbeiten wollen, muss in PnetCDF jeder Lese-/Schreibvorgang kollektiv sein. Das heißt, jeder Prozess muss bei jedem I/O-Aufruf beteiligt sein.

**HDF5** Kommen wir zu HDF5 [HDF], der fünften Version des Hierarchical Data Formats. Ein Beispiel ist in Abbildung 5b. Der Grundtyp ist wie in PnetCDF ein mehrdimensionales Array, welches als Dataset bezeichnet wird. HDF5 lässt außerdem, wie der Name schon sagt, ein hierarchisches Datenformat zu. Das heißt, man kann Gruppen bilden, die wiederum aus Gruppen oder Datasets bestehen. Die Struktur einer HDF5-Datei hat somit gewisse Ähnlichkeiten zu der aus Ordnern bestehenden Struktur eines Dateisystems.

**NetCDF4** NetCDF4 ist eine komplett überarbeitete Version von NetCDF. Das Hauptziel war es, Kompatibilität zu HDF5 zu schaffen. NetCDF4 ermöglicht es, Dateien sowohl im HDF5-Format als auch im NetCDF3-Format zu öffnen. Hier wurde viel in der sequentiellen Performanceoptimierung getan, weshalb einige Nutzer auf NetCDF4 umgestiegen sind. Die parallele Komponente ruft aber lediglich die parallelen Funktionen von HDF5 auf.

## 3.4 Auswahl der Bibliothek

Prinzipiell gibt es zwei Kriterien für unsere Entscheidung. Das erste Kriterium ist die Performanz. Das Zweite ist die Komplexität der Implementierung in Bezug auf unser Datenformat. Wir werden nun zuerst darauf eingehen, wie die Umsetzung in der jeweiligen Bibliothek ausschauen würde. NetCDF4 werden wir nicht weiter betrachten, da sämtliche parallelen Funktionalitäten mit HDF5 nativ benutzt werden können.

Die Umsetzung mit Hdf5 wäre wohl die leichteste. Es müsste für jede Zelle ein eigenes Dataset, das heißt ein mehrdimensionales Array, erzeugt werden. Im globalen Bereich der HDF5-Datei könnte man dann globale Werte setzen, um eine Zuordnung zwischen Position der Zelle in der Domain und in der Datei zu finden.

Die Bibliothek PnetCDF verwendet mehrdimensionale Arrays, genannt Variablen, als Speichertyp. Wir würden jede Zelle als eine Variable abspeichern. Es ist zu erwarten, dass die Laufzeit besser als die von HDF5 ist, weil HDF5 in der Abstraktionshierarchie ein wenig höher angesiedelt ist.

Bei MPI-IO müssten wir uns ein Format selber erstellen. Wir würden auch mit dieser Bibliothek die Zellen als Array in die Datei schreiben. Der Unterschied würde darin bestehen, dass die Positionen der Zellen in der Datei selbst zu berechnen wären. Wenn diese Berechnungen effizient gemacht werden, sollte MPI-IO die beste Laufzeit bieten.

Die Entscheidung fiel letztlich zu Gunsten von PnetCDF aus. Es bietet die nötige Abstraktion für unser Zellformat. Wir gehen davon aus, dass es zu einer besseren Laufzeit als HDF5 führt, weil HDF5 mit seiner Hierarchiestruktur zusätzlichen Overhead benötigt. [MCA+06] bestätigt, dass die Laufzeit mindestens genauso gut sein sollte.
Gegen MPI-IO haben wir uns entschieden, weil uns die Komplexität zu Beginn als zu hoch erschien und wir lieber eine höher angesiedelte Bibliothek verwenden wollten.

Trotzdem haben wir noch eine Implementierung in MPI-IO gemacht, weil ausreichend Zeit zur Verfügung stand.

## 3.5 Umstellung des ASCII-I/O auf binären I/O

Da sowohl PnetCDF als auch MPI-IO binäre Formate sind, haben wir uns entschlossen, die aktuelle Implementierung auf ein binäres Format umzustellen. Ein Nebenergebniss soll ein effizienterer sequentieller I/O-Mechanismus sein, dessen Ausgabedateien weniger Speicherplatz verbrauchen.

Ab jetzt werden wir die Einstellungen zu Beginn der Moleküldatei als Molekülheader bezeichnen.
Wir speichern nunmehr den Molekülheader und die Moleküldaten in zwei verschiedene Dateien. Das ist erforderlich, weil es nicht möglich ist, in einer Datei sowohl ASCII-Daten als auch Binärdaten zu haben. Diese Aufteilung wird uns später nützlich sein, weil die parallelen I/O-Bibliotheken ebenfalls ein binäres Format haben.
Um dies umzusetzen, haben wir eine Anpassung in den Klassen *Domain, DomainDecompBase* und *Molecule* vorgenommen. In der Klasse *Domain* gab es die Methode *writeCheckpoint()*, die einen Checkpoint im Oldstyle-Format herausgeschrieben hat. Sie haben wir in die Methoden *writeCheckpointHeader()* und *writeCheckpoint()* aufgeteilt, wobei die erst genannte Methode für das Schreiben des Molekülheaders und die Zweite für das Schreiben der Moleküldaten zuständig ist. So ist es möglich, dynamisch zu steuern, ob man Molekülheader und Moleküldaten in eine oder in zwei verschiedene Dateien schreiben will. Das wird beim parallelem I/O-Mechanismus vorteilhaft sein.
Die Methode *writeCheckpoint()* leitet nun an die Methode *writeMoleculesToFile()* aus der Klasse *DomainDecompBase* weiter, ob wir eine Binärdatei oder eine ASCII-Datei beschreiben. Diese iteriert über alle Moleküle in der Simulation und ruft für jedes Molekül entweder die Methode *write()* für das *OldStyle*-Format oder die Methode *writeBinary()* für das Binär-Format aus der Klasse *Molecule* auf.

# 4 Implementierung

Wie am Ende des vorherigen Kapitels erwähnt, haben wir eine Implementierung mit PnetCDF und eine Implementierung mit MPI-IO vorgenommen, welche wir in diesem Kapitel beschreiben. Anschließend weisen wir auf Anknüpfungspunkte für weitere Arbeiten hin. Zuletzt stellen wir dar, wie die I/O-Module zu benutzen sind.

## 4.1 Dateiformat

Als erstes werden wir das Dateiformat genauer spezifizieren. In Abschnitt haben wir uns dafür entschieden, ein Zellformat aufzubauen, bei dem wir die Domain in gleiche große Subdomains aufteilen. Da die Domain ein Quader ist, sind die Subdomains kleinere Quader gleicher Größe. Die Subdomains werden ab sofort als Zellen bezeichnen. Jede Zelle enthält eine Liste an Partikeln, weil eine Reihenfolge der Partikel innerhalb einer Zelle nicht mehr notwendig ist.
Folgender Abschnitt wird sich damit beschäftigen, die Seitenlängen der Zellen zu berechnen. In Analogie zu dem Linked-Cells-Algorithmus, der in Kapitel 2 eingeführt wurde, werden wir die Variable, die die Seitenlänge der Zellen speichert, Cutoffradius nennen.

## 4.2 Berechnung des Cutoffradius

Das entscheidende Kriterium für den Cutoffradius ist, dass sich jede Subdomain auf einem Prozess befindet. Würden sich die Moleküle einer Subdomain auf verschiedenen Prozessen befinden, könnte man diese nicht ohne weiteres in die Datei schreiben. Das würde wieder Kommunikation zwischen den Prozessen erfordern. Die Prozesse müssten entweder Moleküle austauschen oder vereinbaren, welcher Prozess an welche Position des Zellbereichs in der Datei schreiben soll. Ansonsten wäre nicht sicherzustellen, dass sich die Daten nicht gegenseitig überschreiben.

Um Komplexität zu sparen, ist es das Ziel, dass sich jede Zelle nur auf einem Prozess befindet und nicht auf mehreren verteilt ist. Zusätzlich versuchen wir die Zellen so groß wie möglich zu machen, damit die I/O-Kanäle des Dateisystems so gut wie möglich ausgelastet werden.

Um dies zu erreichen, müssen wir uns die verschiedenen Domaindekompositionen anschauen, da diese für die Aufteilung der Domain auf die Prozesse zuständig sind.
In Kapitel 2 haben wir diese schon kennengelernt. Umgesetzt wird das Ganze, indem wir die Klasse *DomainDecompBase* um die abstrakte Methode *getIOCutoffRadius(int dimension)* erweitern. Diese gibt uns den Cutoffradius für die jeweilige Dimension zurück.

Der *DomainDecompDummy*, der die Dekomposition für eine sequentielle Kompilierung ist, ist für uns nicht interessant, weil in einer solchen Kompilierung keine parallele I/O-Bibliothek benutzt werden kann. *getIOCutoffRadius()* gibt den Dummywert 0 zurück.

Als nächstes beschäftigen wir uns mit der normalen *DomainDecomposition*. Sie weist jedem Prozess einen gleich großen Teilbereich der Domain zu. Wir können somit den gesamten Bereich eines Prozesses als eine Zelle definieren.
Schon bei der Initialisierung der Simulation hat die *DomainDecomposition* die Domain in ein Gitter an Prozessen aufgeteilt. Es ist deshalb bekannt, in wie viele Prozesse die Domain in jeder Dimension aufgeteilt ist.
In der Methode *getIOCutoffRadius(int dim)* teilen wir die globale Länge der Domain durch die Anzahl an Prozessen in der angegebenen Dimension *dim* und geben diesen Wert zurück.

Außerdem gibt es die *KDDomainDecomposition*. Sie führt eine Lastverteilung durch und versucht, jedem Prozess ungefähr die selbe Anzahl an Molekülen zuzuteilen. Um dies zu bewerkstelligen, teilt die *KDDomainDecomposition* die Domain in kleine Quader auf, die dann auf die Prozesse verteilt werden.
Wir benutzen die Seitenlängen der Quader der *KDDomainDecomposition* als die Größe unserer Zellen. Es ist nicht möglich einen größeren Cutoffradius zu finden, weil es passieren kann, dass ein Prozess nur einen kleinen Quader besitzt. Die Methode *getIOCutoffRadius(int dim)* gibt hier einfach bloß die Membervariable *_cellSize[dim]* zurück. *_cellSize* speichert die Seitenlänge eines kleinen Quaders der *KDDomainDecomposition*.

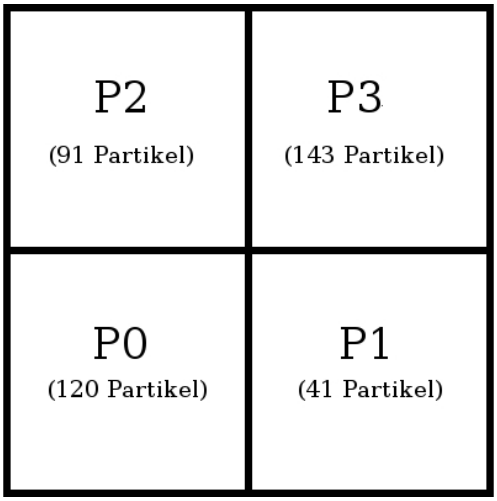
## 4.3 PnetCDF

Wir wollen nun mit der Bibliothek PnetCDF unser Zellformat abspeichern. Wie in Abschnitt 3.3 erwähnt, besteht eine PnetCDF-Datei aus einer Menge an mehrdimensionalen Arrays, die Variablen genannt werden. Um dies zu realisieren, gibt es in PnetCDF einen Headerbereich, der aus Dimensionen und Variablen besteht. Die Variablen werden durch einen Datentyp, z.b. *int*, und einer oder mehreren Dimensionen charakterisiert. Aus diesen Informationen berechnet PnetCDF sodann intern die Position der Variable in der Datei. In Abschnitt 4.1 haben wir das Dateiformat festgelegt. Es ist eine Menge an Zellen, die jeweils aus einer Partikelliste bestehen. In der PnetCDF-Datei wird also in jeder Variable genau eine Zelle gespeichert.

Wir verwenden PnetCDF sowohl zum Schreiben als auch zum Einlesen im kollektiven Modus, der in Kapitel 2.1 eingeführt wurde.

### 4.3.1 Schreiben der Datei

Eine PnetCDF-Datei besteht aus 2 Teilen, dem Headerbereich und dem Datenbereich. Da wir die Datei im kollektiven Modus öffnen, muss jeder Prozess den selben Header haben. Dadurch kann PnetCDF organisieren, welcher Prozess an welche Position schreibt. Deshalb hat PnetCDF zwei Phasen eingeführt. Die erste Phase ist die Define-Phase, während der man die Headerdaten setzt und PnetCDF am Ende der Phase prüft, ob der Header auf allen Prozessen konsistent ist. In der zweiten Phase, der Schreib-Phase, kommt der effektive Schreibvorgang.




```
netcdf pnetcdf-0.restart {
// file format: CDF-5 (big variables)
dimensions:
        d0 = 1815 ;  // 1815 = 121 * 15
        d1 = 615 ;   // 615 = 41 * 15
        d2 = 1365 ;  // 1365 = 91 * 15
        d3 = 2205 ;  // 2205 = 143 * 15
variables:
        double v0(d0) ;
        double v1(d1) ;
        double v2(d2) ;
        double v3(d3) ;
// global attributes:
                :cutoffRadiusX = 48.5 ;
                :cutoffRadiusY = 48.5 ;
                :cutoffRadiusZ = 97. ;
                :numMolecules = 400ULL ;
}
```

Abbildung 6: Links die Domain, bei der jeder Prozess genau eine Zelle enthält(normale *DomainDecomposition).* Rechts der zugehörige PnetCDF-Header

Zuerst kommen wir zur Define-Phase, das heißt dem Setzen des Headers. Im Abschnitt 4.1 haben wir festgelegt, dass jede Zelle/Variable eine Liste von Partikeln ist. Um eine Liste fester Länge zu speichern, brauchen wir eine Dimension pro Variable. Wir erstellen ein Array *numParticlesPerCell*, das für jede Variable die Anzahl an Partikeln speichert. Damit das Array korrekt gefüllt werden kann, brauchen wir zwei Schritte. Zuerst iteriert jeder Prozess über seine Partikel und findet heraus, wie viele Partikel er pro Zelle besitzt. Jetzt ist *numParticlesPerCell* lokal gesetzt. Um die globalen Werte zu erhalten, müssen wir die Arrays der verschiedenen Prozesse noch zusammenführen. Das geschieht über ein *MPI-Allreduce*. Diese Funktion benutzen wir, um alle *numParticlesPerCell*-Arrays der verschiedenen Prozesse komponentenweise zu addieren und das Ergebnisarray an alle Prozesse zurück zu senden. Danach ist die Anzahl an Partikeln pro Zelle auf allen Prozessen bekannt.

Wir haben nunmehr die Daten, um den PnetCDF-Header zu füllen. Bevor wir die Variablen schreiben, sind noch einige Attribute zu setzen, damit die Daten sinnvoll eingelesen werden können. Dies sind zum einen der Cutoffradius und zum anderen die globale Anzahl an Molekülen. Gespeichert werden diese als sogenannte globale Attribute der PnetCDF-Datei. Die Variable $i$ wird im Header als '*vi*' und die dazu gehörende Dimension als '*di*' bezeichnet.

PnetCDF erlaubt es nicht, für eine Variable mehr als einen Grunddatentypen zu verwenden. Wir können also nicht den bereits implementierten MPI-Datentypen *ParticleData*, der zum Austausch von Molekülen dient, benutzen, um die Daten zu speichern. Wir müssen deshalb auf einen Grunddatentypen festlegen. In unserem Fall ist es der Typ *double*, weil mit diesem auch Integerwerte gespeichert werden können. Eine Zelle mit $l$ Partikeln hat die Dimension $l \cdot 15$, weil ein Partikel genau 15 Membervariablen enthält.

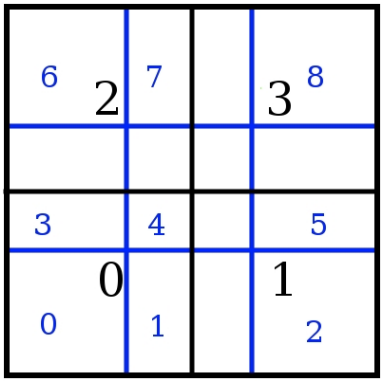



Abbildung 7: Domain mit 4 Prozessen(schwarz) und 9 Zellen(blau) aus der einzulesenden Datei. Prozess 0 muss jetzt zum Beispiel die Zellen 0, 1, 3, 4 lesen.

Wir rufen *ncmpi_enddef()* auf, wodurch der Header geschrieben und auf Konsistenz geprüft wird. Ein Beispiel für die Bildung eines Headers ist in Abbildung 6 zu sehen.

Es wird nun für jede Zelle des Prozesses mit $l$ Partikeln ein *double*-Array der Länge $15 \cdot l$ angelegt. Es wird über alle Partikel des Prozesses iteriert und die Partikel in die Arrays kopiert.
Danach rufen wir für jede Zelle die PnetCDF-Funktion *ncmpi_put_var_double_all()* auf. Diese schreibt eine komplette Variable in die Datei. Der Zusatz *_all* gibt an, dass es sich um eine kollektive Methode handelt. Es ist im übrigen nicht möglich, eine nichtkollektive Methode im kollektiven Modus auszuführen.

### 4.3.2 Einlesen der Daten

Die Molekülkonfiguration lesen wir über das alte Headerformat(Abbildung 4) ein. Das Lesen der PnetCDF-Datei besteht aus zwei Phasen. In der Ersten wird der Header eingelesen, woraufhin der Cutoffradius und die Anzahl an Partikeln pro Zelle bekannt ist.

Um die Zellen in der zweiten Phase zu lesen, müssen wir in Erfahrung bringen, welchem Prozess welche Zelle zuzuordnen ist. Ein Beispiel der Problematik ist in Abbildung 7 dargestellt. Dazu brauchen wir die Bounding Box, welche die Bezeichnung für die Subdomain des Prozesses ist. Mit dem Cutoffradius können wir berechnen, in welchen Zellen sich die Eckpunkte der Bounding Box befinden. Da wir den Index jeder Ecke aufgeteilt in seine Dimensionen kennen, können wir ganz einfach über die Zellen iterieren, die in der Bounding Box liegen oder sie schneiden.

Bevor wir jetzt die Variablen einlesen, führen wir ein *AllReduce* durch, um herauszufinden, was die maximale Anzahl an Variablen ist, die ein Prozess einliest. Dies ist notwendig, weil unsere Lesemethode kollektiv ist. Das heißt, bei jedem Durchgang, wo ein Prozess *read* aufruft, müssen alle anderen Prozesse ebenfalls *read* aufrufen. Wenn ein

Prozess weniger Zellen lesen muss als andere, führt dieser Dummy-Reads aus.

Zuletzt wird ein *Allreduce* ausgeführt, um die Anzahl an Molekülen pro Komponente zu setzen. Bisher war jedem Prozess nur die lokale Anzahl an Molekülen pro Komponente bekannt. Mit diesen Daten kann die Initialisierung abgeschlossen werden.

## 4.4 MPI-IO

Dieser Abschnitt widmet sich der Implementierung in MPI-IO. Der Hauptunterschied zu PnetCDF besteht darin, dass wir die Positionen der Daten in der Datei selber bestimmen müssen. Hierzu wird ein Header gebaut, der dem PnetCDF-Header sehr ähnlich ist.

### 4.4.1 Schreiben der Datei

Um das Format mit MPI-IO zu klären, besprechen wir zunächst, welche Daten der Header der Datei benötigt. Wie bei PnetCDF müssen wir die Anzahl an Molekülen und den Cutoffradius mitspeichern. Außerdem brauchen wir ausreichend Informationen, um die Positionen der einzelnen Zellen in der Datei festzulegen. Für die Positionsberechnung ist die Speichergröße des Headers und die Anzahl an Partikel, die vor einer Zelle in der Datei stehen, erforderlich. Im Header muss also zusätzlich die Anzahl an Partikeln pro Zelle gespeichert werden. In der PnetCDF-Datei ist diese Rolle den Dimensionen zugekommen.

Wir benutzen wie schon beim PnetCDF-Schreibvorgang ein *AllReduce*, um die Anzahl an Partikeln pro Zelle herauszufinden.
Bei MPI-IO ist es jetzt möglich, den Datentyp *ParticleData* zum Schreiben zu verwenden. Mit diesem werden in MarDyn die Partikel zwischen den Prozessen ausgetauscht. Das umständliche Bauen der *double*-Arrays bleibt uns deshalb mit MPI-IO erspart.

Der Schreibvorgang mit MPI-IO ist in folgende Schritte aufgeteilt:

1. Der Header wird von dem Prozess mit Rang 0 geschrieben.
2. Der selbe Prozess broadcastet die Headerlänge an die anderen Prozesse.
3. Alle Prozesse schreiben ihre Zellen. Die Position einer Zelle $i$ berechnet sich aus der Summe der Headerlänge und der Summe an Partikeln der Zellen 0 bis $i-1$. Bekannt sind diese Daten durch den Broadcast und das *Allreduce*.

### 4.4.2 Lesen der Datei

Beim Lesevorgang ist die größte Veränderung im Vergleich zu PnetCDF der Header. Es stellte sich die Frage, ob wir den Header durch eine kollektive Methode einlesen oder wir den Prozess mit Rang 0 den Header lesen und anschließend broadcasten lassen. Bei der kollektiven Variante wäre es möglich, dass MPI-IO selbstständig Optimierungen durchführt und dadurch die Laufzeit verbessert.

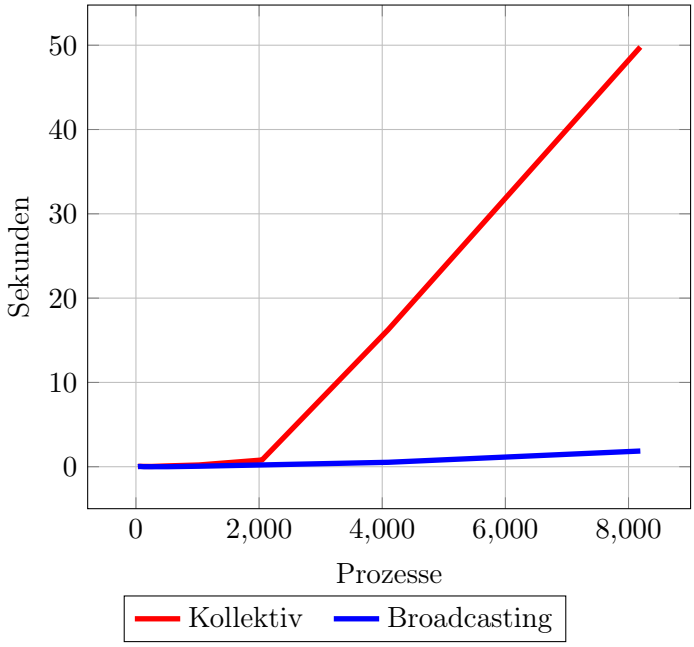



Abbildung 8: benötigte Zeit, um den Header zu lesen

Um Kenntnis über die bessere Variante zu erhalten, haben wir beide Versionen implementiert und auf Geschwindigkeit getestet(siehe Abbildung 8). Wir haben dies auf dem SuperMuc in Garching, auf dem das parallele Dateisystem GPFS läuft, getestet. Da wir für unseren Test die normale Domaindekomposition benutzt haben, gibt es bei einem Lauf mit $i$ Prozessen $i$ Zellen. Für den Header ergibt sich eine Größe von $40 + 4 \cdot i$ Bytes. Wir haben Messungen für alle Zweierpotenzen zwischen 16 und 8192 Prozessen vorgenommen.
Wie man deutlich erkennen kann, wächst die Laufzeit des Broadcastings nur linear zur Anzahl an Prozessen. Bei Betrachtung dieser Laufzeiten ist klar, dass wir uns für das Broadcasten des Headers entschieden haben.

Um festzustellen, welche Zelle durch welchen Prozess gelesen werden muss, verwenden wir die selbe Logik wie in PnetCDF. Um die Position einer Zelle $i$ zu berechnen, addieren wir die Längen der Zellen 0 bis $i - 1$ und die Headerlänge.

## 4.5 Anknüpfungspunkte für weitere Arbeiten

Sollte eine neue Domaindekomposition entwickelt werden, muss auch eine passende Version der Methode *getIOCutoffRadius* implementiert werden. Damit der parallele I/O-Mechanismus funktioniert, ist sicher zu stellen, dass eine Zelle sich nie auf verschiedenen Prozessen befindet, sondern immer nur auf einem.
Für die Effizienz wäre es optimal, wenn jeder Prozess genau eine Zelle enthält. Der

```
output CheckpointWriter 1 oldstyle
output BinaryCheckpointWriter 1 binary
output PnetCDFCheckpointWriter 1 pnetcdf
output MPI-IOCheckpointWriter 1 mpi-io
```

Abbildung 9: Mögliche Ausgabeeinstellungen

```
phaseSpaceFile OldStyle oldstyle-0.restart.xdr
phaseSpaceFile Binary binary-0.restart.header.xdr binary-0.restart.xdr
phaseSpaceFile PnetCDF pnetcdf-0.restart pnetcdf-0.restart.nc
phaseSpaceFile MPI-IO mpi-io-0.restart mpi-io.restart.mpi
```

Abbildung 10: Mögliche Eingabeeinstellungen

Cutoffradius sollte also so groß wie möglich gewählt werden.

Wie sich nach der Implementierung während des Benchmarkings herausgestellt hat, gibt es einen Fehler bei der PnetCDF-Implementierung. Ab 1024 Prozessen funktioniert der Mechanismus nicht mehr. Laut Fehlermeldung liegt das daran, dass NetCDF nicht mehr als 1000 Dimensionen zulässt. Wir sind uns allerdings nicht ganz sicher, ob das der Fehler ist, weil laut PnetCDF-Dokumentation [Pne] diese Beschränkung nur für NetCDF und nicht für PnetCDF gilt.

## 4.6 Benutzung der I/O-Module

Ergänzend zu Abschnitt 2.4.1 werden wir nun erklären, wie die verschiedenen I/O-Module zu benutzen sind. Man kann ein Einleseverfahren wählen und zudem beliebig viele Ausgabeverfahren.

Abbildung 9 zeigt die vier Plugins, mit denen man die Daten heraus schreiben kann. Mit dem Wort "*output*" wird der Konfigurationsdatei angedeutet, dass ein Ausgabeplugin definiert wird. Das nächste Wort wählt das Plugin aus. *CheckpointWriter* steht für das *Oldstyle*-Verfahren. Binärausgaben werden mit dem *BinaryCheckpointWriter* gemacht. Mit dem *PnetCDFCheckpointWriter* oder dem *MPI-IOCheckpointWriter* wird entweder das PnetCDF- oder das MPI-IO-Verfahren benutzt. Die darauf folgende Zahl gibt an, nach wie vielen Zeitschritten eine Ausgabe gemacht werden soll. Das Schlusswort definiert den Namen der Datei.

Nun kommen wir zur Definition der Einleseplugins(siehe Abbildung 10). Eine Konfigurationsdatei ist nur korrekt, wenn ein Einleseplugin vorhanden ist. Am Anfang steht das Wort *phaseSpaceFile*. Das ist sinnvoll, weil der PhaseSpace in der Regel den zu simulierenden Raum beschreibt. Als nächstes kommt der Pluginname, wie zum Beispiel *OldStyle*. Bei *OldStyle* gibt man den Namen einer Eingabedatei an. Für die anderen drei

Varianten muss zuerst der Name der Datei mit dem Molekülheader und dann der Name der Datei mit den Moleküldaten angegeben werden.

Damit die parallelen I/O-Mechanismen funktionieren, muss das Programm parallel kompiliert werden, weil beide Mechanismen Komponenten von MPI benötigen.
Außerdem muss man für PnetCDF die Umgebungsvariablen *$PNETCDF_INC* für den Includepfad und *$PNETCDF_LIB* für den Bibliothekspfad setzen.
Hinweis: Auf dem MAC-Cluster und dem SuperMUC, auf denen unsere Tests gelaufen sind, sind die Variablen nach dem Laden des PnetCDF-Moduls im übrigen schon gesetzt.

# 5 Ergebnisse

Wir haben zunächst die MPI-IO Implementierung genauer analysiert. Im Anschluss daran haben wir einen Laufzeitvergleich zwischen allen Implementierungen durchgeführt. Dieser wurde zuerst auf einem sequentiellem und auf einem parallelem Dateisystem gemacht.

Getestet haben wir auf dem MAC-Cluster, einem Rechencluster der TU München mit einem sequentiellem Dateisystem und auf dem SuperMUC in Garching, auf dem das General Parallel Filesystem(GPFS) installiert ist. Theoretisch kann das komplette System 200 GB/s leisten.

## 5.1 Analyse der MPI-IO Implementierung

Für die Analyse haben wir uns die Datenübertragungsrate und die Hauptverbraucher innerhalb der Lese-/Schreibvorgänge angeschaut.

**Setup** Wir haben zur Analyse eine Moleküldatei der Größe $4,9$GB benutzt. Die Moleküle sind regelmäßig angeordnet. Sie bilden einen Quader, der den größten Teil der Domain ausfüllt. Für das hier verwendete Beispiel, das 33 Millionen Moleküle besitzt, wären bei Produktivläufen ungefähr 8000 bis 33000 Prozesse realistisch. Die Ergebnisse sind auf dem SuperMUC entstanden.

In Abbildung 11 sind die Bandbreiten für das Lesen und das Schreiben der Zellen ersichtlich. Wir haben die Zeiten über zwei Läufe gemittelt.
Es ist schwer, einen Vergleich zu anderen Benchmarks anzustellen, weil diese in der Regel mindestens 4GB pro Prozess lesen/schreiben. Wir haben allerdings für alle Prozesse nur 4,9GB zu lesen/schreiben, denn für MarDyn wollten wir eine realistische Dateigröße testen. In der Abbildung 11 gibt es sowohl beim Lesen als auch beim Schreiben einen Knick. Das Problem ist, dass ab einer gewissen Prozessanzahl, hier 256 oder 1024 Prozesse, der Datendurchsatz nicht mehr steigt. Weil es aber bei einer größeren Prozessanzahl mehr Zellen gibt, muss auch der Filepointer öfter verschoben werden mit der Konsequenz, dass die Datenrate ab diesem Zeitpunkt sinkt. Hätten wir mit einer größeren Datei gearbeitet, wäre die Datenrate weiter gestiegen und der Knick wäre erst später aufgetreten.

Über den Lesevorgang im Speziellen ist nicht viel zu sagen. Dessen Laufzeit wird durch das Lesen der Zellen bestimmt. Das Einlesen und Broadcasten des Headers dauert nicht nennenswert lange. Wir wollen deswegen nicht näher auf den Lesevorgang eingehen.

Nun analysieren wir noch die einzelnen Komponenten des Schreibvorgangs (Abbildung 12). Wie man erkennen kann, wird der Schreibvorgang durch den Unteraufruf *assertDisjunctivity()*, der die Korrektheit der Daten überprüft, und das eigentliche Schreiben der Zellen dominiert.
Das *Allreduce* wurde nicht genauer betrachtet, weil es anders als erwartet, nicht wirklich ins Gewicht fällt.

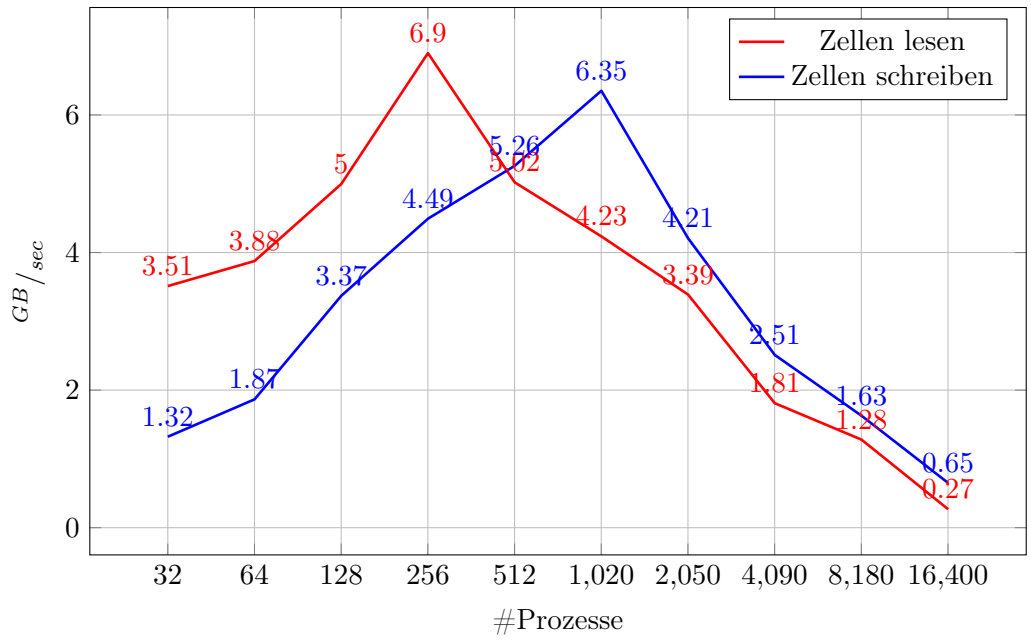



Abbildung 11: Lesen/Schreiben einer 4,9GB Datei, die gleichmäßig auf die Prozesse aufgeteilt ist, mit MPI-IO

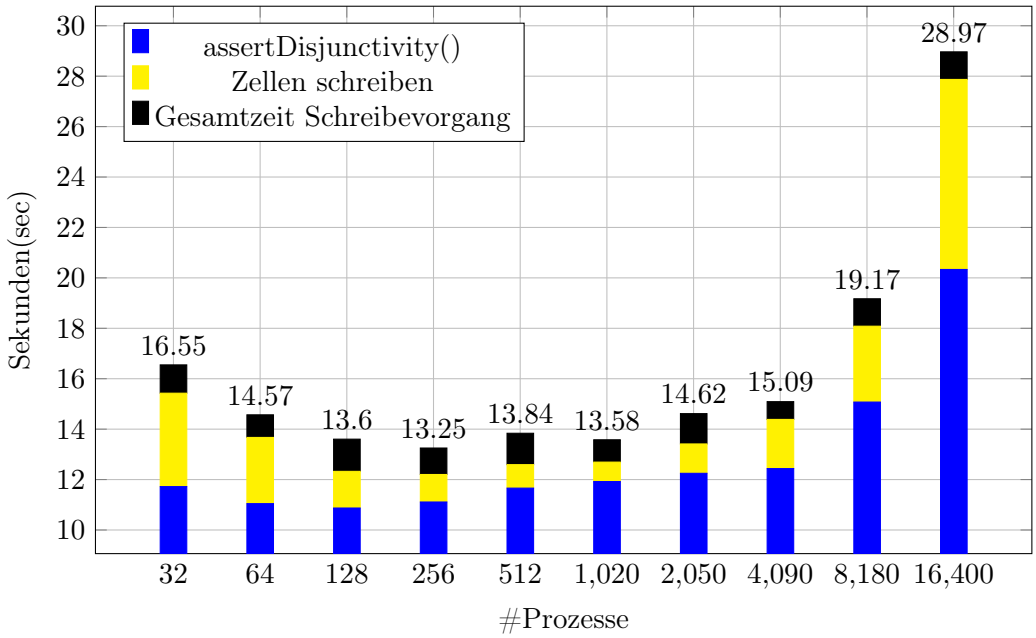



Abbildung 12: Schreiben von 33 Millionen Molekülen. Die Gesamtzeit, die der Schreibevorgang benötigt, mit den 2 größten Zeitverbrauchern

| #Moleküle | 0,33 Millionen | 3 Millionen | 33 Millionen |
|---|---|---|---|
| Oldstyle | 148 MB | 1,35 GB | 10,1 GB |
| Binär | 71 MB | 0,53 GB | 4,5 GB |
| PnetCDF | 43 MB | 0,53 GB | 4,1 GB |
| MPI-IO | 43 MB | 0,53 GB | 4 GB |

Tabelle 2: Größe der Moleküldateien für den jeweiligen I/O-Mechanismus, gemessen auf dem MAC-Cluster

## 5.2 Vergleich der Implementierungen

Für den Vergleich haben 3 verschiedene Molekülgrößen benutzt:

- 0,33 Millionen Moleküle: Realistische Anzahl für ungefähr 50-350 Prozesse
- 3 Millionen Moleküle: Normal werden ungefähr 750-3000 Prozesse dafür benutzt
- 33 Millionen Moleküle: Normal mit 8000-33000 Prozessoren berechnet

Um realistische Ergebnisse zu erhalten, ließen wir jede Konfiguration dreimal laufen und mittelten daraufhin die Werte.

Bevor wir den Laufzeitvergleich anstellen, schauen wir uns den Speicherverbrauch an, die durch die einzelnen Implementierungen entstehen (siehe Tabelle 2).
Schön zu sehen ist, dass die binäre Implementierung und die beiden parallelen Implementierungen, deren Datensätze ebenfalls binär sind, über 50% des Speicherplatzes reduzieren. Auf anderen Systemen fallen die Speichergrößen und deren Verhältnis geringfügig anders aus. Das hängt vermutlich mit den verschiedenen Versionen der Bibliotheken zusammen.

**Sequentiell** Die Ergebnisse, die auf dem MAC-Cluster entstanden, werden in Abbildungen 13, 14 und 15 dargestellt.
Es ist zu ersehen, dass mit der binären Umsetzung des Oldstyle-Formats bereits eine starke Laufzeitverbesserung erreicht wird. Das Einlesen unseres Zellformates mit MPI-IO und PnetCDF bringt jedoch nochmals eine zusätzliche Laufzeitverkürzung, denn jeder Prozess liest nur die Partikel, die ihm zugeordnet sind und nicht alle Partikel. Gerade bei den Tests mit 33 Millionen Molekülen haben wir uns extrem viel Laufzeit gespart. MPI-IO benötigt zum Einlesen nur 0,01% im Vergleich zum bestehenden OldStyle-Mechanismus. Auch das Schreiben mit MPI-IO und PnetCDF zeigt ebenfalls ein gutes Laufzeitverhalten, welches durch das Weglassen des *assertDisjunctivity()*-Aufrufs eine weitere Laufzeitverkürzung um mehr als 10 Sekunden erzielen würde.
In Abbildung 15 sieht man, dass das Lesen mit MPI-IO um 50% schneller geht, als mit PnetCDF. Vermutlich benutzt PnetCDF eine kollektive Operation, um die Daten auf die verschiedenen Prozesse aufzuteilen. In Abbildung 8 haben wir gesehen, dass unser Broadcastingansatz hier einen Laufzeitvorteil bringt, welcher auf einem sequentiellem Dateisystem noch gravierender sein sollte.

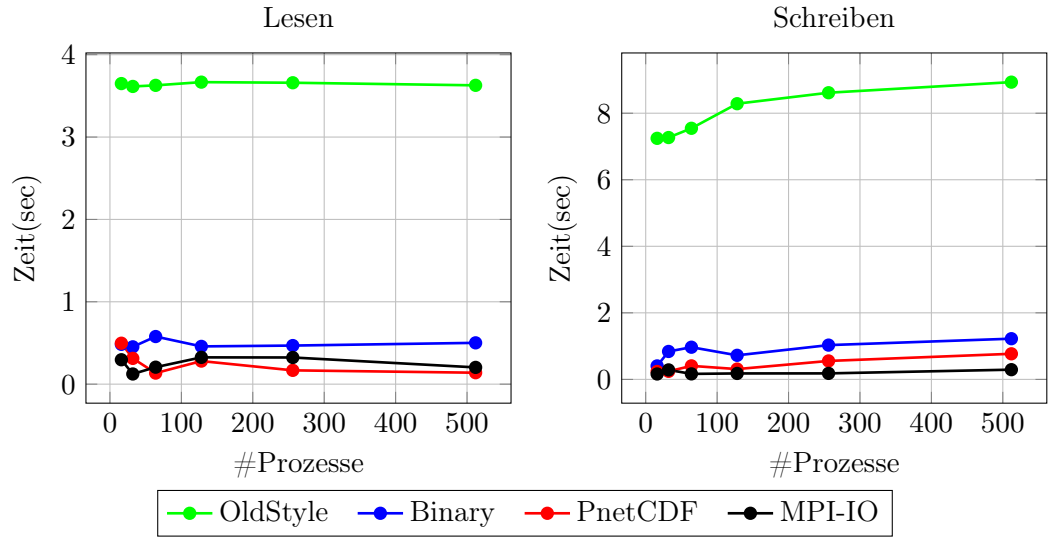



Abbildung 13: Tests mit 0,33 Millionen Molekülen auf dem MAC-Cluster

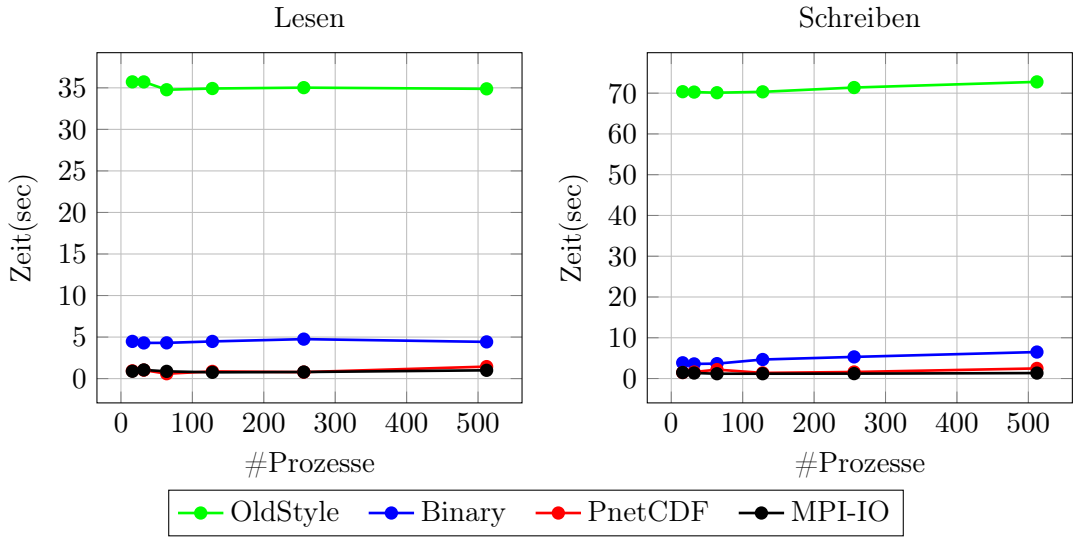



Abbildung 14: Tests mit 3 Millionen Molekülen auf dem MAC-Cluster

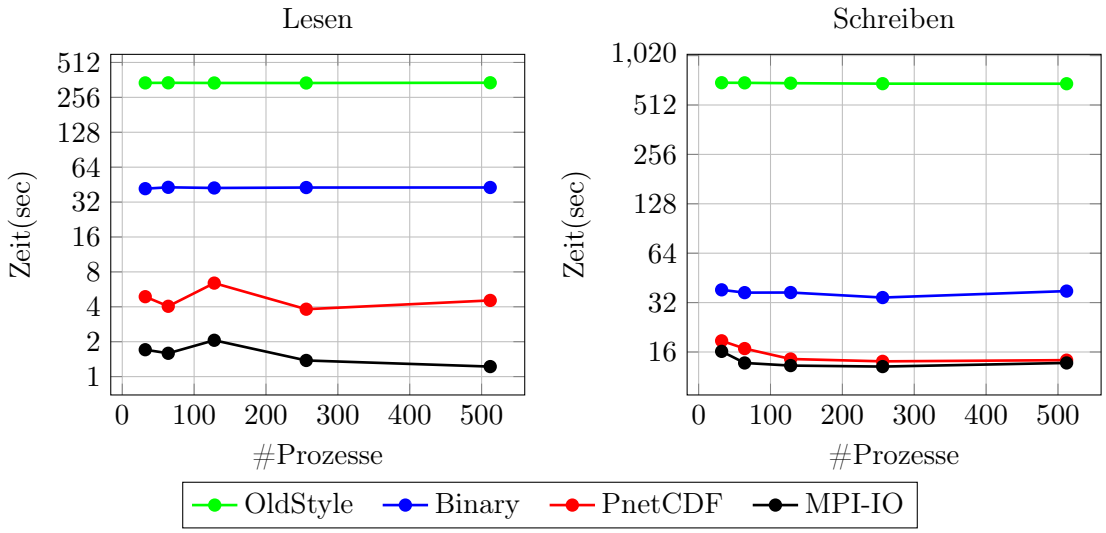



Abbildung 15: Tests mit 33 Millionen Molekülen auf dem MAC-Cluster.

**Parallel** Die parallelen Tests auf dem SuperMUC werden in den Abbildungen 16, 17 und 18 gezeigt. Das Beispiel mit 33 Millionen Molekülen haben wir auf bis zu 16.000 Prozessoren laufen lassen. Wie in Kapitel 4 beschrieben, funktioniert PnetCDF ab 1024 Prozessen nicht mehr, weswegen wir Tests nur bis 512 Prozessen getestet haben.
Wie erwartet, liefen die beiden parallelen I/O-Mechanismen auf einem parallelem Dateisystem schneller als auf einem sequentiellem. Der Unterschied in unseren realistischen Beispielen ist aber nicht so hoch, obwohl dieser sich spätestens ab 256 Prozessen stark bemerkbar machen sollte. Der Grund ist, dass zum Einen die Gesamtdauer der Lese-/Schreiboperationen zu gering ist und zum Anderen die Datenmenge pro Prozess bei den benutzten Dateigrößen zu gering ist.

Unser ursprüngliches Ziel, einen I/O-Mechanismus zu schaffen, mit dem ein Checkpoint-Restart sinnvoll ist, haben wir erreicht. Selbst bei großen Läufen mit 16000 Prozessoren fällt es nicht besonders ins Gewicht, wenn man jede halbe Stunde einen Checkpoint schreiben lässt.

Würde der Unteraufruf *assertDisjunctivity()* für einen Produktionslauf weglassen werden, könnte der Schreibvorgang sogar für große Simulationen weniger als 8 Sekunden dauern. Das Einlesen ist durch die Zellstruktur wesentlich verbessert worden, weil jeder Prozess nur noch die Partikel lesen muss, die ihm wirklich zugeordnet sind.

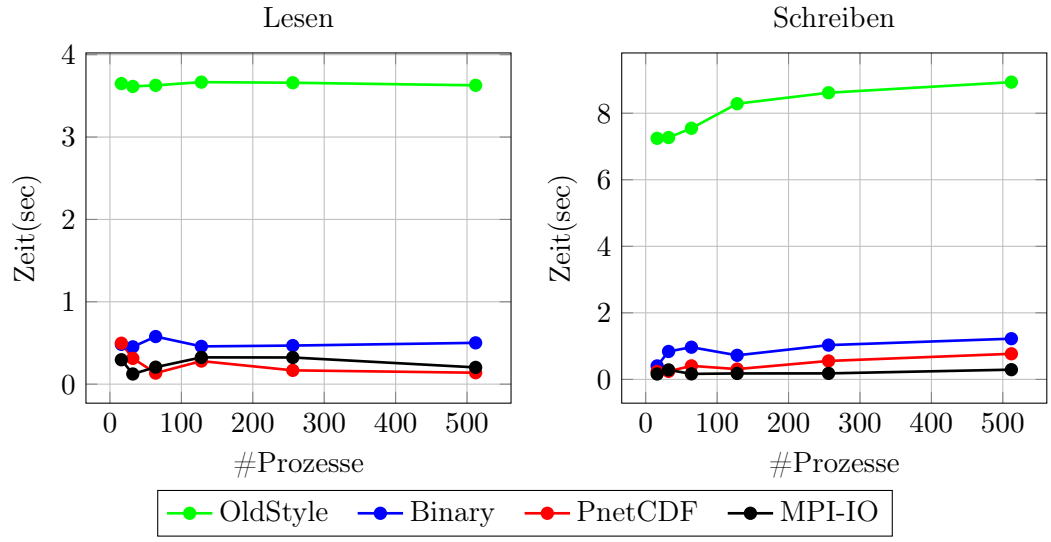



Abbildung 16: Tests mit 0,33 Millionen Molekülen auf dem SuperMUC

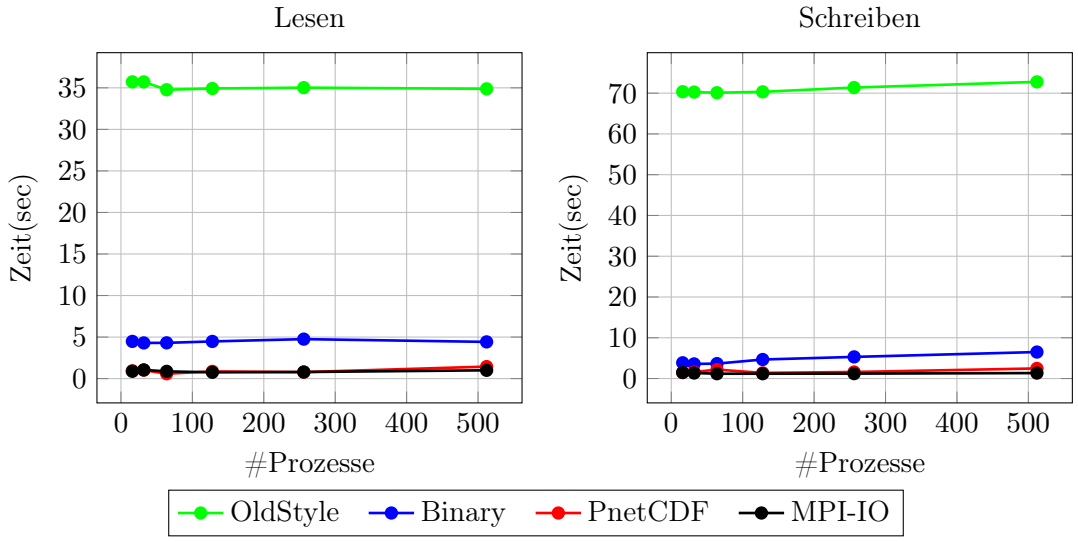



Abbildung 17: Tests mit 3 Millionen Molekülen auf dem SuperMUC

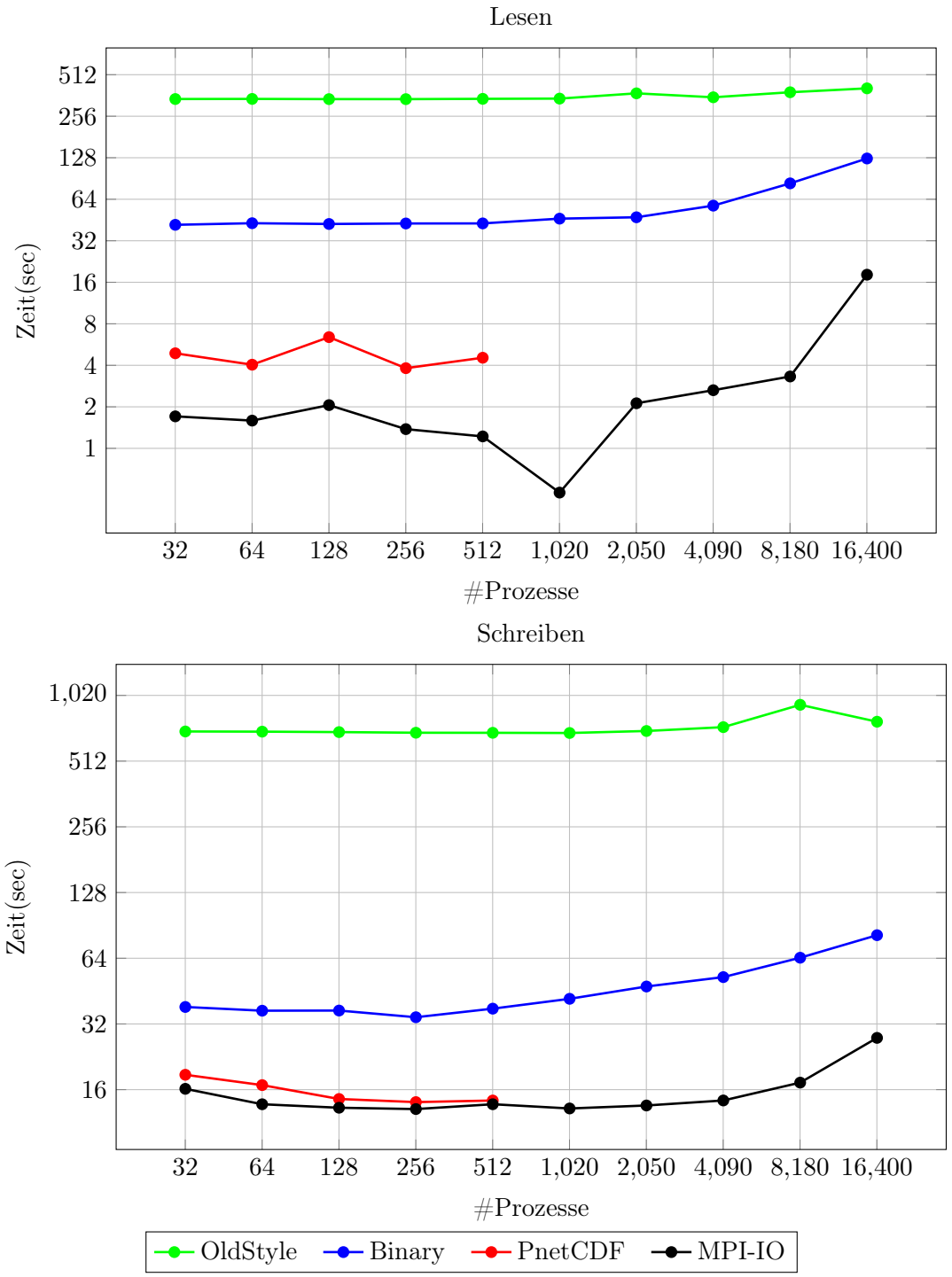



Abbildung 18: Tests mit 33 Millionen Molekülen auf dem SuperMUC

# 6 Zusammenfassung

Diese Arbeit hat sich mit parallelem I/O beschäftigt. Wir haben das Molekulardynamikprogramms MarDyn erweitert und darauf geachtet, keine Veränderungen vorzunehmen, die bestehende Funktionalitäten beeinträchtigen könnten.

Zunächst haben wir eine Evaluierung der gängigen parallelen I/O-Bibliotheken vorgenommen. Aufgrund derer haben wir uns für PnetCDF entschieden.

Aus zeitlichen Gründen konnten wir eine Implementierung mit den beiden Bibliotheken PnetCDF und MPI-IO durchführen. Die Grundidee dahinter ist, eine Zellstruktur ähnlich der Datenstruktur des Linked-Cells-Algorithmus aufzubauen und abzuspeichern.
Damit ein besserer Vergleich zum bestehenden Mechanismus, der auf ASCII-Zeichen basiert, angestellt werden kann, haben wir diesen auf ein binäres Format umgestellt.

Das gesetzte Ziel, einen Checkpoint-Restart zu entwickeln, der so wenig Laufzeit verbraucht, dass es auch für Produktivläufe sinnvoll ist ihn einzusetzen, haben wir somit erreicht. Sollte man sich entscheiden, jede halbe Stunde einen Checkpoint zu schreiben, kostet der I/O weniger als 1% der Gesamtlaufzeit. Durch das verwendete Zellformat ist die Zeit des Einlesens auf ein Minimum verkürzt worden und braucht nur wenige Sekunden.

## Literatur


[Buc10] Martin Buchholz. *Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen*. PhD thesis, Institut für Informatik, Technische Universität München, 2010.

[BW73] J. A. Barker and R. O. Watts. Monte Carlo studies of the dielectric properties of water-like models. *Molecular Physics*, 26:789–792, September 1973.

[CCLC07] Avery Ching, Kenin Coloma, Jianwei Li, and Alok Choudhary. High-performance techniques for parallel I/O. In *Handbook of Parallel Computing: Models, Algorithms, and Applications*. CRC Press, December 2007.

[CLRT00] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4*, ALS'00, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[FWP09] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable massively parallel i/o to task-local files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 17:1–17:11, New York, NY, USA, 2009. ACM.

[GKZ07] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[HDF] Hdf5 homepage. `http://www.hdfgroup.org/HDF5/`.

[LLC+03] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[Lus] Lustre dateisystem. `http://www.lustre.org/`.

[MCA+06] Yang MuQun, Chilan Christian, Cheng Albert, Koziol Quincey, Folk Mike, and Arber Leon. Parallel i/o performance study and optimizations with hdf5, a scientific data package. Technical report, The HDF Group, 2006.

[Pne] Pnetcdf c dokumentation. `http://cucis.ece.northwestern.edu/projects/PNETCDF/doc/pnetcdf-c/index.html/`.

[RD90] Russ Rew and Glenn Davis. Data management: Netcdf: An interface for scientific data access. *IEEE Comput. Graph. Appl.*, 10(4):76–82, July 1990.

[SH02] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pages 231–244, 2002.

[WUA+08] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.