FRIEDRICH-ALEXANDER-UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Checkpointing-Mechanismen für ein massiv paralleles Software-Framework

Christian Schmitt

Masterarbeit

Checkpointing-Mechanismen für ein massiv paralleles Software-Framework

Christian Schmitt

Masterarbeit

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dipl.-Inf. F. Schornbaum
Bearbeitungszeitraum:	11.06.2012 – 30.10.2012

Ich versichere, dass ich diese Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 30.10.2012 .....

Abstract

Checkpointing Mechanisms for a Massively Parallel Software Framework

As scientific requirements to size and exactness of simulations grow, the need for calculating capacity rises. This results in faster, larger and thus more complex super computers, which, in turn, result in higher failure rates of the cluster's components. Hence there is development of techniques to minimize the consequences of program execution abortions.

This work describes the widely used technique of *Checkpoint/Restart (C/R)* in its theoretical foundations and successive optimizations. In addition, an architecture for implementation as part of a massively parallel software framework will be proposed by suggesting possible data formats and a module structure. An implementation following this design will be presented and discussed in characteristic details. This solution will be evaluated with regard to performance, correctness and usability to the software framework's users and developers.

Kurzfassung

Checkpointing-Mechanismen für ein massiv paralleles Software-Framework

Durch die stetig steigenden Anforderungen der Forschung an die Genauigkeit und Größe von Simulationen wächst der Bedarf an Rechenkapazitäten. Dies resultiert in immer schnelleren, größeren und dadurch komplexeren Supercomputern, was in der Folge zu erhöhten Ausfallraten einzelner Cluster-Bestandteile führt. Es werden daher Technologien entwickelt, um die Folgen von Programmabbrüchen zu minimieren.

Diese Arbeit beschreibt die sehr weit verbreitete Technologie des *Checkpoint/Restart (C/R)* in ihren theoretischen Grundlagen und den darauf aufbauenden Optimierungen. Zudem wird eine Architektur für die Implementierung als Teil eines massiv parallelen Software-Frameworks vorgeschlagen, indem mögliche Datenformate besprochen und eine Modul-Gliederung entworfen werden. Die diesen Vorgaben folgende Implementierung wird vorgestellt und in charakteristischen Details besprochen. Die erarbeitete Lösung wird in Hinblick auf die Performance, Korrektheit und die Nutzbarkeit für Benutzer und Entwickler des Frameworks evaluiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

1 Einleitung

1.1 Motivation

Spitzenforschung ist heute mehr denn je auf Höchstleistungsrechner angewiesen: Simulationen ersetzen zunehmend teure, oft auch gefährliche physikalische Experimente. Erst durch numerische Simulationen mittels leistungsfähigen Rechen-Clustern können komplexe theoretische Modelle aufgestellt und validiert werden. Die konsequente Nutzung dieser Möglichkeiten führt dazu, dass bestehende technische Prozesse und Systeme verbessert werden können. Letztendlich ermöglicht dieses Vorgehen sowohl neue wissenschaftliche und technische Errungenschaften, als auch die Entwicklung innovativer und wettbewerbsfähiger Produkte.

Mit *waLBerla (widely applicable Lattice Boltzmann solver from Erlangen)* entsteht am *Lehrstuhl für Systemsimulation* der *Friedrich-Alexander-Universität Erlangen-Nürnberg* ein massiv paralleles und hochflexibles Software-Framework zur Erstellung von umfangreichen Simulations-Anwendungen. Der Fokus lag ursprünglich auf der Entwicklung von Strömungssimulationen mittels der Lattice-Boltzmann-Methode. Mittlerweile ist der Einsatz von *waLBerla* aber auch bei anderen, auf strukturierten Gittern basierenden Anwendungen möglich, indem es eine Vielfalt von Komponenten, wie beispielsweise effiziente Mehrgitterlöser für partielle Differentialgleichungen, bietet. Dadurch existieren inzwischen auch beispielsweise Anwendungen für die Simulation freier Oberflächen oder für Flüssigkeitssimulationen, die mit Festkörper-Dynamik-Simulationen gekoppelt sind.

Dem aus derartigen wissenschaftlichen Anforderungen folgenden Bedarf an immer leistungsfähigere Systeme wird mit der Installation immer größerer Cluster-Systeme Rechnung getragen. Diese Entwicklung lässt sich mit der Top500-Liste veranschaulichen.

Während in der Ausgabe von Juni 2005 alle erfassten Systeme auf die Zahl von 580.336 Prozessoren kommen, sind es drei Jahre später im Juni 2008 bereits 2.414.219 Prozessoren; nach weiteren drei Jahren schon 7.779.924 Kerne [Top12]. Die Entwicklung ist graphisch in Abbildung 1.1 dargestellt. Durch die steigende Anzahl der verbauten Prozessoren und damit auch von Rechnern (Rechenknoten) in solchen Verbunden steigt das Risiko eines Ausfalls des Gesamtverbundes, was oft gleichbedeutend mit dem Abbruch der aktuell laufenden Simulation ist und mit dem Verlust der bisher gewonnen Daten einhergeht. Bei einer angenommenen mittleren Fehlerrate von einem Fehler pro 25 Jahre und Rechenkern tritt für einen aus 500.000 Kernen bestehenden Cluster bereits nach durchschnittlich 26 Minuten ein Prozessorfehler

auf. Das momentan schnellste System der Welt *Sequoia* ist aus über 1,5 Millionen Rechenkernen aufgebaut [Top12].

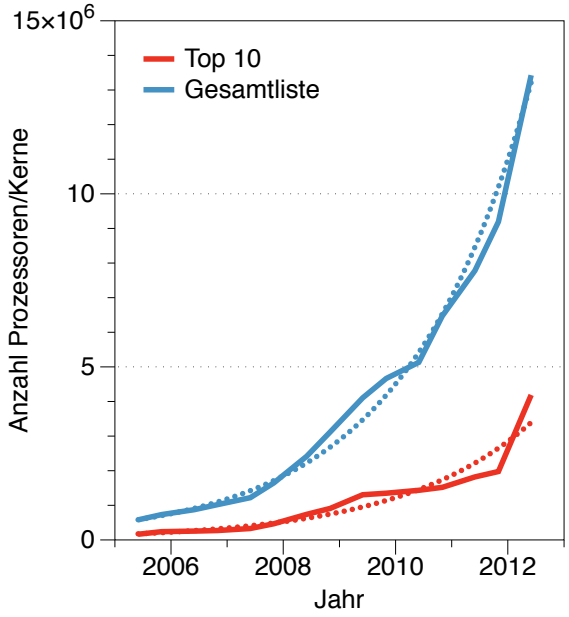

Abb. 1.1: Entwicklung Prozessorzahl der Top500-Liste [Top12].

Im Laufe der Zeit wurden verschiedene Techniken entwickelt, die einem solchen Ausfallszenario vorbeugen (*forward error recovery*) oder die Auswirkungen von Ausfällen minimieren (*backward error recovery*) können. In letzterer Kategorie ist das sogenannte *Checkpointing* eine der am häufigsten genutzten Techniken. Hierbei werden zu definierten Zeitpunkten die relevanten Daten der Anwendung auf persistenten Speicher geschrieben, um im Falle eines Neustarts (beispielsweise nach Programmabbruch) von diesem Punkt weiterarbeiten zu können.

Eine derartige Funktionalität ermöglicht es auch, lang andauernde Simulationen in kürzere Läufe aufzuteilen. Häufig ist es nicht möglich, Cluster über mehrere Wochen komplett für einen Job exklusiv zu reservieren, so dass die Simulation in zeitlich kürzere Berechnungsläufe aufgeteilt werden muss.

1.2 Aufgabenstellung und Zielsetzung

Das *waLBerla*-Framework soll fehlertoleranter werden, indem es um eine Checkpointing-Funktionalität erweitert wird. Diese soll verhindern, dass errechnete Ergebnisse langer Simulationsläufe bei einem Abbruch verloren gehen. Daneben soll auf Basis des entwickelten Verfahrens auch ein Restart-Mechanismus geschaffen werden, der es erlaubt, eine einzelne Simulation in mehrere Läufe aufzuteilen.

Die Implementierung soll, dem Framework-Gedanken hinter *waLBerla* folgend, generisch sein, so dass beliebige bereits existierende und auch durch Neuentwicklungen hinzu kommende Datenstrukturen gespeichert und geladen werden können. Zu diesen Datenstrukturen zählen beispielsweise existierende Kern-Strukturen wie Gebietsaufteilungen. Darüber hinaus soll die Einbindung in die Vielzahl der auf dem Framework aufbauenden Anwendungen möglichst einfach sein, unabhängig von deren Komplexität. Zur Entwickler-Unterstützung soll es eine Möglichkeit geben, die korrekte Funktion zu validieren, d. h. zu testen, dass durch einen Restart keine Daten verfälscht werden. Außerdem soll ein Restart auch auf anderer Hardware bzw. mit

einer anderen Anzahl von Prozessen möglich sein. Die erstellte Lösung soll schlussendlich im Hinblick auf die angesprochenen Ziele, sowie insbesondere auf die erreichte Performance beim Erstellen von Checkpoints, evaluiert werden. Bei Bedarf sollen Optimierungsansätze vorgeschlagen und ggf. umgesetzt werden.

1.3 Strukturierung

In **Kapitel 2** werden für das Verständnis benötigte Begriffe definiert und Grundlagen zu Checkpointing und Optimierungsstrategien erläutert.

Kapitel 3 erklärt in Grundzügen den Aufbau von waLBerla und stellt mögliche Dateiformate vor, in denen Checkpoints gespeichert werden können. Anschließend wird die Architektur des entstandenen Checkpointing-Moduls erläutert und es werden Hinweise zur Implementierung gegeben.

In **Kapitel 4** wird das realisierte Modul in Hinblick auf verschiedene Ziele getestet und bewertet.

Abschließend gibt **Kapitel 5** eine Zusammenfassung über die in dieser Arbeit erreichten Resultate und schlägt mögliche weiterführende Arbeiten vor.

2 Grundlagen

Im Folgenden wird das Verfahren des Checkpointings näher beleuchtet und es werden theoretische Grundlagen zur Basis dieser Arbeit erläutert. Anschließend werden Ansätze zur Optimierung der bestehenden Verfahren aufgezeigt und thematisch verwandte Arbeiten vorgestellt.

2.1 Checkpointing

Checkpointing, auch *Checkpoint/Restart (C/R)* genannt, bezeichnet die Vorgehensweise, den Zustand eines Prozesses auf stabilem (persistentem) Speicher zu sichern. Dadurch kann der Prozess zu einem späteren Zeitpunkt exakt vom gespeicherten Punkt wieder gestartet und die Arbeit an dieser Stelle fortgesetzt werden [SPD+05].

Das Checkpointing existiert in verschiedenen Ausprägungsformen und ist eine weit verbreitete Technik, die beispielsweise auch Prozessmigration zur Lastbalancierung zwischen Rechnern erlaubt. Der von Betriebssystemen bekannte *Ruhezustand* (auch *Hibernation* genannt) kann ebenfalls als Checkpoint angesehen werden. Virtualisierungsprogramme wie *VMWare* erlauben das „Einfrieren“ des laufenden Gast-Betriebssystems und erstellen dabei einen Checkpoint [SPD+05]. Darüber hinaus verfügen sie häufig über Mechanismen, um das in der virtuellen Maschine installierte System auf ältere Stände zurückzuführen.

Als *serielles Checkpointing* bezeichnet man das Vorgehen, dass die Anwendungsausführung unterbrochen wird bis der Checkpoint komplett geschrieben wurde. Unter *forked Checkpointing* versteht man, dass der eigentliche Prozess weiter rechnet, während im Hintergrund der Checkpoint von einem Kind-Prozess geschrieben wird [Vai97, Hur10]. Neben diesen existieren weitere, optimierte Verfahren (vgl. Unterabschnitt 2.1.5).

Als *Checkpoint/Restart Service (CRS)* bezeichnet man einen Dienst, der einen Checkpoint eines einzelnen Prozesses erzeugen und laden kann. Mittels paralleler Laufzeitumgebungen und entsprechenden Kommunikationskanälen können CRS auch mit Anwendungen mit mehreren Prozessen umgehen. Sie werden in drei Stufen unterschieden, die in Unterabschnitt 2.1.2 erläutert werden [Hur10].

Ein CRS kann der Anwendung gestatten, den Zeitpunkt der Checkpoint-Erstellung selbst zu wählen oder die Erstellung in bestimmten Phasen zu verhindern. Man spricht dann von *synchronem Checkpointing*. Bei *asynchronem Checkpointing* wird der Checkpoint von außerhalb der Anwendung, beispielsweise über Signale, initiiert. Die Anwendung kann diese Zeitpunkte nicht beeinflussen [BMO06].

Ein Checkpoint, der erstellt, aber nicht für einen Restart genutzt wurde, wird *unnötig* genannt [Hur10]. *Checkpoint-Intervall* bezeichnet die Zeitdauer zwischen der Fertigstellung zweier aufeinanderfolgender Checkpoints [Vai97]. Die Häufigkeit innerhalb der Programmausführungszeit, mit der Checkpoints erstellt werden, nennt man *Frequenz* [Hur10]. Es ist darauf zu achten, dass sie so niedrig gewählt wird, dass sich die Checkpoint-Intervalle nicht überschneiden und somit nicht mehrere Sicherungen zur selben Zeit erstellt werden.

Die *Mean Time Between Failures (MTBF)* bezeichnet die durchschnittliche Zeitspanne, die zwischen zwei aufeinanderfolgenden Fehlerereignissen vergeht.

2.1.1 Einordnung

C/R ist ein Mechanismus der *rückwärts gerichteten Wiederherstellung* (*backward error recovery*), d. h. nach einem Fehler wird ein korrekter Anwendungs- bzw. Datenzustand aus einem vorher gesicherten Zustand wiederhergestellt.

Daneben existieren noch zwei weitere Kategorien der Fehlerbehandlung: Die *vorwärts gerichtete Wiederherstellung* (*forward error recovery*) und die *Fehlervorbeugung* (auch *Fehlerprävention*, *fault prevention*).

Bei der vorwärts gerichteten Wiederherstellung wird nicht davon ausgegangen, dass im Fehlerfall ein gespeicherter korrekter Zustand verfügbar ist. Es wird stattdessen versucht, den aktuellen, fehlerhaften Zustand in einen korrekten Zustand zu überführen. Diese Vorgehensweise ist, im Gegensatz zur rückwärts gerichteten Wiederherstellung, hoch speziell und nur auf eine Anwendung bzw. einen Algorithmus anwendbar und erfordert dabei sehr genaue Kenntnisse über den Ablauf und die Funktionsweise der Anwendung. Dafür ist sie aber sehr effizient: Rechenzeit wird erst benötigt, wenn ein Fehler auftritt. Während der Programmausführung werden keine Daten unnötig gesichert und damit auch kein Speicherplatz verbraucht [Mil85].

Bei der Fehlerprävention sollen Fehler vor ihrem Auftreten erkannt und behoben werden (*proaktive Fehlerbehandlung*). Dies geschieht über zusätzliche Prüfungen hinsichtlich der Korrektheit des aktuellen Zustands und sich abzeichnender Fehler auf Anwendungs-, System- und Hardware-Ebene. Soweit möglich werden Gegenmaßnahmen ergriffen, beispielsweise indem Prozesse von Rechenknoten, die in Kürze auszufallen drohen, auf andere Knoten migriert werden [LES10].

2.1.2 Funktionsweisen

Abhängig von der Anwendungstransparenz, der Flexibilität, der Portabilität und des Orts der Implementierung lassen sich CRS in drei Ebenen einteilen:

Anwendungsebene

Ein CRS dieser Stufe interagiert direkt mit der Anwendung. Diese muss daher entsprechend angepasst werden, um die zu sichernden Daten identifizier- und zugreifbar zu machen. Diese CRS benötigen also Wissen über die Anwendung sowie ihre Funktionsweise und Zustände, d. h. sie sind nicht oder kaum anwendungstransparent. Andererseits sind sie dadurch aber in der Lage, die Checkpoints mit dem geringsten Speicherbedarf und daher potentiell auch dem geringsten zeitlichen Overhead zu erzeugen. Die Konsistenz der Checkpoints muss durch die Anwendung sicher gestellt werden. Der Ansatz ist sehr flexibel, da die Anwendung selbst geeignete Zeitpunkte zur Erstellung eines Checkpoints (beispielsweise wenn die zu sichernde Datenmenge besonders gering ist) wählen kann. Prinzipiell entstehen bei diesem Ansatz auch die Checkpoints mit der höchsten Portabilität, da zum Erstellen und Einlesen keine besonderen Systemfunktionen benötigt werden. Es muss aber ggf. auf Eigenheiten der Hardware-Architektur, wie Endianness oder das Format der Fließkommazahlen, geachtet werden.

Jede Anwendung kann mit einem solchen CRS versehen werden. Je nach Aufbau kann sich die Implementierung aber sehr zeitintensiv gestalten, so dass verschiedene Möglichkeiten entwickelt wurden, dies zu automatisieren. Es existieren inzwischen eine Vielzahl von übersetzerbasierten Ansätzen (*Porch* [Str98], *CATCH* [LF90], C^3 [BMPS03], *CPPC* [RMG+10] u. a.). Diese ersetzen Markierungen im Quellcode der Anwendung durch (Ent-)Packroutinen, die beim Erstellen des Checkpoints bzw. beim Restart genutzt werden [Hur10].

Benutzerebene

CRS der Benutzerebene laufen im User-Space und virtualisieren die Systemaufrufe der Anwendung. Dadurch hat der CRS Zugriff auf den kompletten genutzten Speicher und kann transparent für die Anwendung den jeweiligen Zustand sichern und laden. Diese Virtualisierung macht den CRS unabhängig vom genutzten Betriebssystem-Kernel und sorgt für Portabilität, solange sich durch die Hardware bzw. Systemarchitektur ergebende Einschränkungen - wie beispielsweise der Aufbau der Datensegmente - beachtet werden. Der Ansatz führt zu größeren Checkpoints als bei CRS auf Anwendungsebene, da ohne Anwendungswissen nicht zwischen wichtigen und unwichtigen Daten unterschieden werden kann. Zusätzlich erzeugt die Virtualisierung bei jedem Systemaufruf Overhead. Im Gegenzug sind dafür wenig bis keine Änderungen der Anwendung nötig.

Die meisten CRS dieser Ebene werden über dynamische Bibliotheken zur Anwendung geladen, um neben den Systemaufrufen auch die `main`-Funktion abzufangen. Über diese wird der CRS initialisiert und ggf. der Prozess aus einem Checkpoint wiederhergestellt. Häufig haben CRS-Bibliotheken dieser Ebene Probleme mit Interprozesskommunikation, Prozesshierarchien, Shared-Memory-Parallelisierungen, Signalen

und Timern, da deren Aufrufe schwierig zu virtualisieren sind. Meist kann aber direkt auf die verwendeten Bibliotheken des Prozesses (insbesondere bei solchen zur Interprozesskommunikation) zugegriffen werden [Hur10].

Verbreitete Implementierungen sind beispielsweise *Libckpt* [PBKL95] und *ckpt* [Zan05] für UNIX-basierte Systeme. Die speziell auf parallele Programmierung ausgerichteten Programmierumgebungen *Brazos* und *SCore* bieten Unterstützung bei den Umsetzungen für Windows [ESB99] respektive UNIX [TSH+00, Rom02]. Das Batch-System *Condor*[1] kann über einen integrierten CRS Jobs zwischen Knoten innerhalb eines Netzwerks migrieren [LTBL97].

Systemebene

Die CRS der Systemebene arbeiten im Gegensatz zu den bereits genannten Klassen im Kernel-Space. Sie haben daher direkten Zugang zu den Prozessen und ihren Threads und kommen daher fast ohne virtualisierte Systemaufrufe aus, wodurch sich der Overhead verringert. Entsprechend ist dieses Verfahren das am wenigsten portable, da passende, modifizierte Betriebssystem-Kernel oder Kernel-Module eingesetzt werden müssen. Die erzeugten Checkpoints sind eng an deren Versionen gebunden. Wie bei den CRS auf Benutzerebene kann meist auf die zur Interprozesskommunikation verwendeten Bibliotheken zugegriffen werden.

Zur Gruppe von CRS auf Systemebene gehören u. a. *Linux-CR* [LH10] und *BLCR* [HD06]. Auch die Virtualisierung von Betriebssystemen, beispielsweise über *Xen* [NMES07] oder *VMWare*, kann dazu gezählt werden [Hur10].

Zusammenfassung

Jeder der beschriebenen Ansätze hat Vor- und Nachteile, durch die sich der Einsatz in bestimmten Szenarien oder (Simulations-)Programmen jeweils empfiehlt oder verbietet. In der Praxis kommen auch Mischformen der erläuterten Techniken vor. Häufig benötigen Benutzerebenen-CRS im Gegensatz zu den Anforderungen der Anwendungsebene nur geringfügige Anpassungen der Anwendung [Rom02].

2.1.3 Performance-Auswirkungen

Die Auswirkungen des Checkpointings auf die Gesamtperformance eines Programms können mit zwei Metriken beschrieben werden: Als *Overhead* bezeichnet man die durch C/R verursachte Erhöhung der Gesamtlaufzeit eines Programms. Unter *Latenz* versteht man die zum Erstellen eines Checkpoints benötigte Zeitspanne. Beim Fall des seriellen Checkpointings sind Overhead und Latenz identisch. Anders hingegen beim forked Checkpointing: Der Overhead ist hier typischerweise sehr gering, während die Latenz oftmals sogar höher als beim seriellen Checkpointing liegt [Vai97].

[1] http://research.cs.wisc.edu/condor/

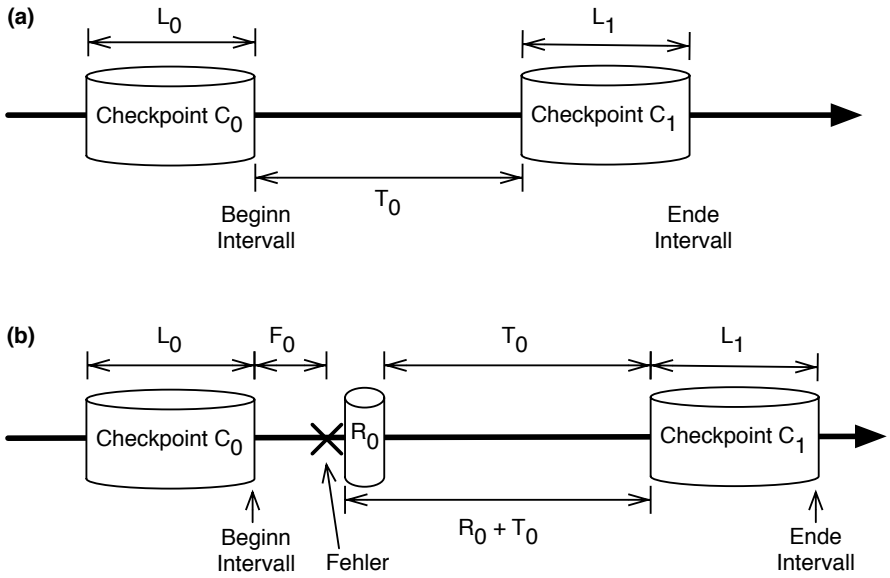

Abb. 2.1: Ausführungszeiten bei sequenziellem Checkpointing. Jeweils ein Intervall ohne und mit Neustart.

Jede Erstellung eines Checkpoints C_i verlängert, wie in Abbildung 2.1a leicht erkennbar, die Ausführungsdauer einer Anwendung um L_i. Die durch das Checkpointing eingebrachte Gesamt-Latenz $L = \sum_i L_i$ hat keine Auswirkungen auf die (eigentliche, fehlerfreie) Programmausführungszeit $T = \sum_i T_i$, sondern nur auf die Gesamtausführungszeit t_{Ges}.

Kommt es zu einem Neustart der Anwendung, dann muss neben der Rekonstruktionsdauer R auch die Programmausführungszeit zwischen dem letzten Checkpoint (bzw. Anwendungsstart) und dem Fehler berücksichtigt werden. Die in dieser Zeit erzielten Ergebnisse gehen verloren und müssen erneut erarbeitet werden, wodurch sich die Gesamtausführungszeit entsprechend erhöht. Dieser Fall ist in Abbildung 2.1b dargestellt. Das Intervall zwischen C_0 und C_1 dauert dem zufolge nicht mehr T_0 Zeiteinheiten. Unter der Voraussetzung, dass sofort nach Fehlerauftritt der Restart erfolgt, verlängert es sich auf $F_0 + R_0 + T_0$ Zeiteinheiten.

Für die Gesamtausführungszeit t_{Ges} und den Checkpointing-Overhead O ergibt sich bei seriellem Checkpointing:

$$t_{\text{Ges}} = \sum_{i=0}^{n_C} T_i + O \tag{2.1}$$

$$O = \sum_{i=0}^{n_C-1} L_i + \sum_{i=0}^{n_R-1} (R_i + F_i) \tag{2.2}$$

mit

n_C = Anzahl der erstellten Checkpoints
n_R = Anzahl der geladenen Checkpoints
T_i = „Sinnvolle“ Ausführungszeit zwischen Checkpoint i und $i+1$
L_i = Latenz des Checkpoint i
R_i = Zeitdauer für Restart i
F_i = Zeitdauer der wiederholten Berechnungen nach Restart i

Dieses Ergebnis gilt unter der Voraussetzung, dass für L die (niedrigere) Dauer des Kopiervorgangs eingesetzt werden muss, entsprechend für forked Checkpointing.

2.1.4 Optimale Checkpoint-Intervalle

Könnte man den durch das Checkpointing eingebrachten Overhead vernachlässigen, dann wäre der ideale Zeitpunkt für einen Checkpoint nach jeder Instruktion. Je nach verwendeter Strategie und Größe der Daten ist der Overhead aber enorm, so dass die Frequenz des Checkpointings gut gewählt werden muss. Unter diesen Umständen wäre der ideale Zeitpunkt zur Checkpoint-Erstellung jeweils direkt vor dem Auftreten eines Fehlers. Dies würde perfekte Fehlervorhersagen erfordern; eine unrealistische Annahme. Moderne Ansätze kombinieren daher die traditionelle statische Checkpointing-Frequenz mit probabilistischen Ansätzen: Durch die feste Frequenz sollen Ungenauigkeiten der Fehlervorhersagen ausgeglichen werden, während der Einfluss der Vorhersagemodelle die Erstellung von unnötigen Checkpoints, bedingt durch die feste Frequenz, minimieren soll.

Die meisten Intervall-Modelle gehen von einer Poisson-Verteilung des Fehlerauftretens aus. Neuere Forschungen schlagen Weibull- bzw. Gamma-Verteilungen als Basis vor [BCR+11, HRC09]. Im Allgemeinen ist die Wahl der Intervalllänge unabhängig von der Latenz, sondern hängt stark vom Overhead ab [Hur10].

[You74] schlägt, ausgehend von einer Poisson-Verteilung der Fehlerhäufigkeit, für das optimale Checkpoint-Intervall τ folgende erste Approximation vor:

$$\tau = \sqrt{2LM} \text{ mit } M\text{: MTBF} \tag{2.3}$$

[Dal03] erweitert diese Näherung um einen Term R für den Restart-Overhead:

$$\tau = \sqrt{2L(M+R)} \qquad \text{für } L \ll M \tag{2.4}$$

Der Ansatz gilt unter der Bedingung, dass die Checkpoint-Latenz deutlich kleiner als die MTBF ist und ist daher nicht für sehr große Cluster geeignet. Für diesen Fall wird folgende Näherung vorgeschlagen:

$$\tau = \sqrt{2L(M+R)} - L \tag{2.5}$$

Beiden Abschätzungen liegt die Annahme zugrunde, dass sich die Dauer der wiederholten Berechnungen durch $\frac{1}{2}\left(\sum T_i + \sum L_i\right) n_R$ gut abschätzen lässt. Dies gilt jedoch nicht bei $M \leq T_i + L_i$. Im Allgemeinen unzureichend ist auch die Annahme, dass nur während der Berechnungszeit, nicht aber während des Schreibens des Checkpoints oder des Restarts ein Fehler auftreten kann. Schließlich wird als optimales Modell für die Gesamtzeit in Abhängigkeit der optimalen Intervalllänge vorgeschlagen:

$$\begin{aligned} T_{\text{Ges}}(\tau) &= \frac{T_{\text{s}} - L + \frac{LT_s}{\tau}}{1 - \frac{1}{M}\left\{\left[\phi_1(\tau)(\tau+L)+R\right]P(\tau) + \phi_2(\tau)(R+\tau+L)(1-P(\tau))\right\}} \\ &\text{mit} \\ \phi_1(\tau) &= \frac{M}{\tau+L} + \frac{1}{1-e^{\frac{\tau+L}{M}}} \\ \phi_2(\tau) &= \frac{M}{R+\tau+L} + \frac{1}{1-e^{R+\frac{\tau+L}{M}}} \\ P(\tau) &= e^{-\frac{R+\tau+L}{M}} \end{aligned} \tag{2.6}$$

T_{s} bezeichnet die rein zur Lösung des Problems aufzubringende Zeit ohne Checkpoint-Latenz und wiederholte Berechnung.

Eine derartige Gleichung löst man üblicherweise numerisch, indem man die Nullstellen der Ableitung iterativ, beispielsweise mit Hilfe des Newton-Verfahrens, ermittelt. Da die analytische Ermittlung der Ableitung in diesem Fall aber nicht leicht ist, bietet es sich an, eine einfache Bisektionsmethode zu verwenden. Ausgehend von den beiden Auswertungen bei $t_{\text{low}} = \epsilon$ und $t_{\text{high}} = M + R + L$ berechnet man $t_{\text{avg}} = \frac{1}{2}(t_{\text{low}} + t_{\text{high}})$. Danach berechnet man die Ableitungen an jedem dieser drei Punkte mittels Finiter Differenzen und prüft die Ergebnisse auf Vorzeichenwechsel. Das Minimum von $T_{\text{Ges}}(\tau)$ befindet sich in dem Intervall $[\tau_{\text{low}}, \tau_{\text{avg}}]$ bzw. $[\tau_{\text{avg}}, \tau_{\text{high}}]$, deren Grenzpunkte entgegengesetzte Vorzeichen haben. Durch wiederholte Bisektion mit Subsegmenten der Länge $[\epsilon, M]$ erreicht man den Punkt, dass $\tau_{\text{high}} - \tau_{\text{low}} < \epsilon$ wird. Damit ist eine hinreichende genaue Näherung an τ gegeben, die die Gesamtausführungszeit minimiert [Dal03].

2.1.5 Weitere Optimierungsansätze

Um den Overhead zu reduzieren sollte nicht nur die Wahl des optimalen Checkpoint-Intervalls betrachtet werden, sondern auch der Schreibvorgang der Checkpoints an sich.

Eng mit forked Checkpointing verwandt ist *mirrored Checkpointing*, auch *continuous Checkpointing* genannt. Hierbei wird eine Kopie des Speicherabbilds der Anwendung ständig im Speicher gehalten, welche dann im Hintergrund auf stabilen Speicher geschrieben wird. Besonders vorteilhaft ist diese Technik im Einsatz bei Echtzeitsystemen, da der durch das Checkpointing eingebrachte Overhead reduziert wird und es so leichter fällt, harte Zeitvorgaben zu erfüllen.

Bei einem *pre-copy-Ansatz* entfällt der Aufwand, die Speicherkopie des Prozesses dauerhaft aktuell zu halten. Vor der Erstellung des Checkpoints wird einmalig ein Speicherabbild erstellt, um es auf stabilen Speicher zu schreiben. Danach kann es wieder gelöscht werden. Diese Technik eignet sich besonders, wenn die Anwendung den Speicher des Systems zu einem so großen Teil belegt, dass durch die Kopie *Thrashing-Effekte*[2] auftreten würden.

Der zeitliche Hauptfaktor bei der Erstellung des Checkpoints ist das Schreiben der Daten auf stabilen Speicher. Es ist daher naheliegend, hier anzusetzen und die Menge der zu schreibenden Daten zu reduzieren. Der einfachste Ansatz ist es, nur die wirklich notwendigen Daten zu sichern. Dies benötigt aber Wissen über die Anwendung und ist darum nicht immer umsetzbar. Eine weitere Möglichkeit ist die *Komprimierung* der zu schreibenden Daten. Als positiver Nebeneffekt benötigen die Checkpoints weniger Speicherplatz, im Gegenzug steigt aber die Prozessorbelastung.

Meist ändern sich zwischen zwei aufeinanderfolgenden Checkpoints nicht alle Daten, so dass nur die Differenzen zwischen den Speicherabbildern gesichert werden können. Man spricht dann von *inkrementellem Checkpointing*. Beim Restart werden dann mehrere Checkpoints benötigt, um den aktuellen Zustand wiederherstellen zu können. Der gesparte I/O-Aufwand muss aber mit erhöhter Rechenzeit erkauft werden, um die Unterschiede zwischen den Speicherabbildern zu erkennen [Hur10]. Neue Forschungen schlagen hierfür beispielsweise GPU-basierte Hash-Verfahren vor [FRB+11].

2.1.6 Zusammenfassung

Es wurden wichtige Begriffe definiert, die im Verlauf dieser Arbeit verwendet werden, und das C/R in den Begriff der Fehlertoleranz eingeordnet. Anschließend wurden verschiedene Funktionsweisen und Möglichkeiten zur Implementierung des Checkpointings vorgestellt und diskutiert. Abschließend wurden die Auswirkungen auf die Performance einer Anwendung sowie Optimierungen, insbesondere die optimale Wahl der Zeitpunkte für die Erstellung der Checkpoints beschrieben.

[2] Beim *Thrashing* ist das System permanent damit beschäftigt, Seiten aus dem Hauptspeicher auszulagern und nachzuladen, so dass der Prozessor die meiste Zeit im Wartezustand verharren muss. Die erzielte Rechenleistung sinkt dadurch auf einen Bruchteil des theoretisch möglichen Werts.

2.2 Verwandte Arbeiten

In Unterabschnitt 2.1.2 wurden bereits einige existierende CRS und C/R-Bibliotheken vorgestellt.

In [Sha11] werden die C/R-Fähigkeiten von *OpenMPI* anhand der *NAS Parallel Benchmarks (NPB)* und eines prototypischen numerischen Strömungslösers auf Basis der Lattice-Boltzmann-Methode (LBM) untersucht. Des Weiteren werden die Auswirkungen des verwendeten Dateisystems, die Platzierung der MPI-Prozesse und die Größe des Checkpoints im Hinblick auf den Overhead analysiert. Es wird eine verlustbehaftete Komprimierung für das Checkpointing der LBM vorgeschlagen: Statt alle eigentlich benötigten 19 Werte der Verteilungsfunktion in jeder Zelle zu sichern sollen nur jeweils vier Werte gesichert werden. Beim Restart sollen die fehlenden Daten aus diesen errechnet werden.

[BGG+09] beschreibt ein virtuelles Dateisystem, welches das Datenlayout einer Anwendung auf ein optimiertes Layout abbildet. Durch diese Indirektion soll die Checkpointing-Latenz auf einen Bruchteil der sonst benötigten Zeitspanne sinken. Erreicht wird dies durch das Umschichten von kleinen und einzelnen Blöcken hin zu großen, zusammenhängenden Blöcken, die prozessweise angeordnet sind.

Ebenfalls den Weg über ein virtuelles Dateisystem schlägt [XCZ08] vor, um das Speichern von Dateizuständen zu verbessern. Bestehenden CRS auf Anwendungs- und Benutzerebene soll es dadurch ermöglicht werden, ohne die Portabilitätseinschränkungen der CRS auf Systemebene transparent für den Benutzer Dateioperationen zu sichern. Umgesetzt wird dies über inkrementelles Checkpointing und damit ähnlich dem von Datenbanken bekannten Transaktions-Protokoll: Alle Dateimodifikationen sollen als Differenzen aufgezeichnet werden, um bei einem Restart den exakten Zustand rekonstruieren zu können.

Abseits des traditionellen HPC-Umfelds betrachtet [VSCS+09] den Einsatz von C/R in Massively Multiplayer Online Games (MMOGs), wie beispielsweise *World of Warcraft* oder *Eve Online*, und vergleicht verschiedene Techniken. In diesem Anwendungsgebiet ist es wichtig, die Checkpointing-Latenz so gering wie möglich zu halten, um einen erträglichen Spielbetrieb zu ermöglichen. Durch die hohe Änderungszahl bestimmter Daten (bspw. Spielerpositionen oder Lebenspunkte der Charaktere), sollen aber u. U. gewisse Ungenauigkeiten beim Restart erlaubt werden können.

2.3 Zusammenfassung

Es wurde die theoretische Grundlage zu dieser Arbeit erklärt, um die in den nachfolgenden Kapiteln vorgestellte Architektur und Implementierung der C/R-Funktionalität in waLBerla nachvollziehen zu können. Die aufgeführten verwandten Arbeiten geben einen Überblick über andere Architekturen und Implementierungen, aber auch Ansätze zur Verbesserung bestehender Lösungen. Es wurde auch der Einsatz

dieser Techniken in einem Anwendungsgebiet abseits des High Performance Computing (HPC) diskutiert.

3 Architektur und Implementierung

Im folgenden Kapitel wird der grobe Aufbau des waLBerla-Frameworks und für die Implementierung in Frage kommende Datenformate vorgestellt. Danach wird die Organisation des Checkpointing-Moduls erläutert und auf Aspekte der Umsetzung eingegangen. Nach der Erläuterung der Integration des Checkpointings in waLBerla-Anwendungen wird die Umsetzung einiger Unterkomponenten besprochen.

3.1 Struktur von waLBerla

Das Software-Framework waLBerla ist klar nach Funktionalitäten gegliedert. Der Kernbereich, der sog. *Core*, enthält die von jeder *Anwendung* benötigten Komponenten. Das sind beispielsweise das Einlesen der Konfigurationsdatei oder die Überprüfung der spezifizierten physikalischen Simulations-Parameter auf Plausibilität. Daneben existieren eine Vielzahl verschiedener *Module*, deren Funktionalität sich von der speziellen Behandlung von Randbedingungen über die Einbindung von Grafikprozessoren via *CUDA* oder *OpenCL* bis hin zu Ausgabemöglichkeiten in VTK-Dateien oder via graphischer Benutzeroberfläche erstreckt.

Eine Anwendung ist die Verbindung des Kerns und mehrerer Module zu einem lauffähigen Programm. Die benötigten Komponenten werden über bereitgestellte Funktionen dem Core mitgeteilt. Diese Komponenten umfassen Arbeitsschritte, die zugehörigen Algorithmen, Daten und benötigte Datenstrukturen. Der Core löst die Abhängigkeiten der jeweiligen Einzelteile auf und veranlasst, entsprechend der benötigten Reihenfolge, ihre Initialisierung.

Analog werden die in jedem Zeitschritt der umgesetzten Simulation auszuführenden Arbeitsschritte definiert: Die zu verwendenden Algorithmen und Operationen werden mit ihrer gewünschten Reihenfolge dem Core mitgeteilt. Dazu gehört neben den auszuführenden Rechenschritten auch beispielsweise die Kommunikation bei der Verwendung mehrerer Prozesse via Message Passing Interface (MPI) oder die Ausgabe der errechneten Daten [Don11, Fei12].

Module verfügen in der Regel über einen *Manager*, der die Initialisierung der notwendigen Datenstrukturen übernimmt und über den auch der Zugriff erfolgt. Diese Manager sind als *globale Datenobjekte* (*GDO*) anwendungsweit verfügbar. Oft kann auch direkt auf die Daten des Moduls über diesen Mechanismus zugegriffen werden. Auch die Daten des Core sind so erreichbar: Ein Beispiel hierfür ist die jeder Simulation zugrunde liegende Blockstruktur, die unter anderem die errechneten Resultate,

aber auch die Geometrie des Gebiets und andere Daten speichert. Pro Datenart wird hierfür ein Feld verwendet, welches über eine einzigartige ID angesprochen wird.

Es ist leicht zu erkennen, dass beim Erstellen eines Checkpoints die GDO gesichert werden müssen, um den kompletten Zustand der Anwendung zu erhalten.

3.2 Mögliche Datenformate

Es existieren viele Technologien, um Daten aus dem Arbeitsspeicher persistent auf die Festplatte zu schreiben. Im Rahmen dieser Arbeit wurden drei populäre Bibliotheken untersucht und hinsichtlich ihrer Eignung für Checkpointing in waLBerla bewertet.

3.2.1 boost::serialization

Bei *boost::serialization*[1] handelt es sich um einen Teil der sehr mächtigen und weitverbreiteten C++-Bibliothek *Boost*[2]. Diese ist eine Sammlung portabler Unterbibliotheken, die unterschiedlichste Aufgaben erfüllen, von der vereinfachten Implementierung von Entwurfsmustern über Metaprogrammierung und Speicherverwaltung bis hin zu Graphen-Algorithmen[DAR12]. Die *Boost License* (siehe Anhang B.1), unter der der Serialisierungs-Part steht, ist im Stil der MIT-Lizenz gehalten und gilt damit als frei und mit der GNU General Public License (GPL)[3] vereinbar[FSF12]. Sie erlaubt die Verwendung sowohl in kommerzieller, closed-source, als auch in quelloffener Software. In waLBerla wird Boost bereits ausgiebig genutzt. Somit liegt es nahe, boost::serialization zur Persistierung der Zwischenergebnisse zu verwenden.

Die Serialisierung mittels boost::serialization ist auf das Speichern und Laden von Klasseninstanzen ausgelegt. Die Implementierung kann auf zwei Arten erfolgen: In der Klasse selbst (*intrusive*) oder außerhalb (*non-intrusive*). Im ersten Fall wird die Klasse um eine Methode `serialize()` (alternativ ist eine Aufteilung in `save()` und `load()` möglich) erweitert, die dadurch Zugriff auf alle Member-Variablen hat. Im anderen Fall müssen die relevanten Member-Variablen von außen zugänglich (`public`) sein. Die Serialisierung erfolgt in ein *boost::archive* genanntes Archiv, welches binär, in Textform oder als XML in eine Datei geschrieben wird. boost::serialization und boost::archive bilden auch die Basis für *boost::mpi*, bei dem solche serialisierten Archive zwischen Prozessen versendet werden.

boost::serialization unterstützt Polymorphie, indem es die Serialisierung von Objekten über einen Zeiger des Basistyps zulässt. Auch Arrays und STL-Container stellen keinerlei Probleme dar. Zudem können die serialisierten Objekte mit einer Versionsnummer versehen werden, um bei Änderungen in der Datenstruktur der

[1] http://www.boost.org/libs/serialization/
[2] http://www.boost.org/
[3] http://www.gnu.org/licenses/gpl.html

Klasse auch ältere Stände laden zu können. Listing A.1 zeigt ein Beispielprogramm, das einige der erwähnten Features verwendet.

Für die vorliegende Arbeit wurde die Verwendung von boost::serialization verworfen. Die Serialisierung auf Klasseninstanzenebene macht eine sinnvolle Integration in waLBerla sehr schwierig. Eine Anwendung kann damit nur schwer steuern, welche Daten wichtig sind und gesichert werden müssen und welche Daten wieder errechnet werden können, so dass auf deren Speicherung verzichtet werden kann. Da boost::mpi keine Unterstützung für MPI-I/O [DAR12, Tro08] bietet, müssten die Daten aller Instanzen in prozess-separate Dateien geschrieben werden. Dies führt bei großen Clustern mit mehreren Hundert Rechenknoten zu Performance-Einbußen und macht die Verwaltung der so erstellten Checkpoints unnötig komplex. Weiterhin gestaltet sich die Rekonstruktion des Checkpoints mit einer anderen Anzahl von Prozessen deutlich schwieriger, da die Daten beim Laden anders auf die Prozesse verteilt werden müssen, als dies beim Schreiben der Fall war. Jeder Prozess muss dann möglicherweise aus verschiedenen Dateien lesen, um alle benötigten Informationen zu bekommen. Der alternative Ansatz, einen oder mehrere separate Schreibprozesse zu bestimmen, scheidet ebenfalls aus Geschwindigkeitsgründen aus, da sehr hoher Kommunikations- und Synchronisierungsaufwand zwischen den Prozessen erforderlich wäre.

3.2.2 HDF5

Die Bezeichnung *Hierarchical Data Format (HDF)* steht für ein Dateiformat zur Speicherung und Verwaltung großer Mengen meist numerischer Daten. Es wurde ursprünglich vom US-amerikanischen *National Center for Supercomputing Applications (NCSA)* an der *University of Illinois* entwickelt. Inzwischen wird die Standardisierung und Weiterentwicklung von der gemeinnützigen *HDF Group* vorangetrieben. Diese stellt auch Application Programming Interfaces (APIs) für die in der Wissenschaft am meisten verwendeten Programmiersprachen C, C++, Fortran 77, Fortran 90 und Java bereit. Außerdem existiert eine offizielle, bislang unvollständige Schnittstelle für .NET[4]. Drittanbieter bieten weitere Bindings für beispielsweise Python[5,6], Perl[7] oder MATLAB[8] an. Die *HDF5 License* (siehe Anhang B.2) ist eine freie, im MIT-Stil gehaltene Lizenz und damit kompatibel zur GPL.

Für I/O-Operationen nutzt die HDF5-Bibliothek die Schnittstelle MPI-I/O, die aktuelle MPI-Implementierungen zur Verfügung stellen. Sie kann somit transparent ohne Wissen der exakten Cluster-Konfiguration und -Hardware arbeiten.

[4] http://hdf5.net/
[5] http://alfven.org/wp/hdf5-for-python/
[6] http://code.google.com/p/h5py/
[7] http://search.cpan.org/dist/PDL-IO-HDF5/
[8] http://www.mathworks.com/help/techdoc/ref/hdf5.html

Im Nachfolgenden wird der aktuelle Standard *HDF5* vorgestellt. Jede HDF5-Datei ist hierarchisch aufgebaut. Die einzelnen Elemente werden als *Information Sets* bezeichnet. Ein solches Set ist ein Container für Nutzdaten, wie einzelne Variablen oder Arrays, und speichert auch die zugehörigen Metadaten, wie Datentyp und Dimensionen. Daneben kann ein Information Set wiederum ein weiteres Information Set enthalten. An oberster Stelle dieses hierarchischen Aufbaus steht ein einzelnes Wurzelelement.

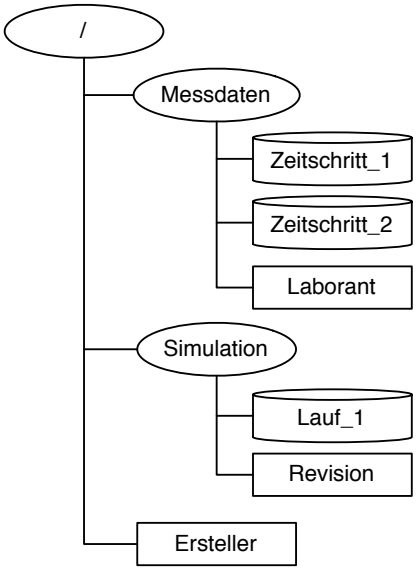

Abb. 3.1: Logische Darstellung einer HDF5-Datei mit Gruppen, Datasets und Attributen. Der erzeugende Programmcode ist in Listing A.2, der technische Aufbau in Listing A.3 aufgeführt.

Groups

Ein Information Set zur Stukturierung der Datei, das aber keine Nutzdaten enthält, wird in HDF5 als *Group* bezeichnet und lässt sich mit einem Verzeichnis auf Dateisystemebene vergleichen. Es hat (mit Ausnahme des Wurzelelements) stets ein übergeordnetes Element und eine beliebige Anzahl Nachfolger, d. h. es ergibt sich ein gerichteter azyklischer Graph [FHK+11]. In Abbildung 3.1 sind unterhalb des Wurzelelements die beiden Gruppen „Messdaten“ und „Simulation“ zu sehen.

Dataspaces

Als *Dataspace* wird ein durch Dimensionalität und Ausdehnung definiertes Information Set bezeichnet, das den Rahmen für (Nutz-)Daten einer HDF5-Datei vorgibt. Es existieren drei verschiedene Arten von Dataspaces:

- *Skalare* Datenräume besitzen eine Dimensionalität von 0, d. h. sie nehmen nur einzelne Elemente auf. Auch wenn diese von komplexen, zusammengesetzten Datentypen sein können, werden skalare Datenräume in der Regel für Attribute, bestehend aus einem einzigen Wert, verwendet.

- *Einfache* Datenräume werden zur Speicherung ein- oder mehrdimensionaler Arrays verwendet. Sie besitzen eine Mindest- und eine Maximalgröße (die aber auch als unendlich groß definiert werden kann). Die maximale Dimensionalität liegt implementierungsbedingt bei 32.

- *Null*-Datenräume können keine Daten speichern. Dieser Datenraumtyp existiert lediglich aus Kompatibilitätsgründen [HDF12a].

Datasets

Daten werden, vergleichbar einem üblichen Dateisystem, nicht direkt in den Gruppen oder Dataspaces gespeichert, sondern in *Datasets*. In einem Dateisystem entsprechen den Datasets die Dateien. Diese Art von Information Set repräsentiert Array-Variablen mit beliebigen Abmessungen und Dimensionen. Diese Größen werden vom Dataspace vorgegeben, der das Dataset enthält. Abbildung 3.1 beinhaltet die beiden Datasets „Zeitschritt_1“ und „Zeitschritt_2“ unter der Gruppe „Messdaten“ und „Lauf_1“ unter „Simulation“.

Den Datasets können verschiedene Speicher-Layouts (zusammenhängend, stückweise oder kompakt für kleine Datasets) zugrunde gelegt werden. Durch diese kann bei bestimmten Operationen eine höhere Performance erreicht und ggf. zusätzliche Funktionalitäten wie paralleles I/O oder Datenkompression nutzbar gemacht werden [FHK+11].

Attributes

Um einzelne Werte zu speichern sind die *Attributes* vorgesehen. Diese entsprechen *Key/Value Pairs* und werden in der Regel eingesetzt, um die in Datasets gespeicherten Informationen genauer zu beschreiben. Das Beispiel von Abbildung 3.1 definiert die beiden Attribute „Laborant“ und „Revision“, um die jeweilige Gruppe näher zu beschreiben, während sich das Attribut „Ersteller“ sich auf die gesamte Datei bezieht [FHK+11].

Datentypen

HDF5 unterstützt neben den standardmäßigen Datentypen wie `char`, `int`, `double` auch die Erstellung eigener, zusammengesetzter Datentypen. Um Portabilität zu gewährleisten werden die Eigenheiten verschiedenener Prozessorarchitekturen (u. a. x86, Cray, MIPS, Alpha) in Bezug auf Endianness und Adressierbarkeit berücksichtigt. Bei Bedarf werden die Daten transparent für den Benutzer konvertiert. Zur Definition jedes Datasets und jedes Attributs ist stets ein Datentyp notwendig [FHK+11]. Im Beispiel aus Abbildung 3.1 könnte man beispielsweise für die Datasets „Zeitschritt_1“, „Zeitschritt_2“ und „Lauf_1“ den Typ `H5T_NATIVE_DOUBLE` verwenden, auf Intel-Systemen ein Alias für den Typ `H5T_IEEE_F64LE`. Die Attribute „Laborant“ und „Ersteller“ sind Zeichenketten und würden als `H5T_C_STRING` mit variabler Länge gespeichert werden. Für „Revision“ als `unsigned int` könnte man `H5T_NATIVE_UINT` (Intel: `H5T_STD_U64LE`) wählen. Die Daten werden beim Lesen auf anderen Plattformen in den jeweils passenden Datentyp konvertiert.

Zusammenfassung

HDF5 eignet sich sehr gut als Bibliothek zum persistenten Speichern der Checkpoints. Da es sich um eine Standardbibliothek des wissenschaftlichen Rechnens handelt, ist die Verbreitung auf Clustern entsprechend groß. Auch die Nutzung von MPI-I/O ist ein großes Plus. Man darf davon ausgehen, dass es auf jedem Cluster vorhanden und nutzbar konfiguriert ist, unabhängig vom Cluster-Aufbau und der verwendeten Hardware. Die erstellten Checkpoints werden in einer Datei gespeichert und sind so portabel, dass sowohl mit einer anderen Prozessoranzahl, als auch sogar auf einer komplett anderen Hardware ein Restart möglich ist.

Zusätzlich ist es möglich, die erstellten Checkpoint-Dateien in anderen Programmen zu verwenden, da eine Vielzahl von Programmen HDF5-Dateien einlesen und weiterverarbeiten kann.

3.2.3 netCDF

Network Common Data Form (netCDF)[9] bezeichnet ein selbstbeschreibendes und portables Dateiformat für wissenschaftliche Anwendungen, sowie die zugehörige Bibliothek, die zur Erstellung und Manipulation von netCDF-Dateien dient. Es ist ein Standardformat des wissenschaftlichen Rechnens und wird vor allem in der Klimatologie und in Geoinformationssystemen verwendet. Ursprünglich basierte das Dateiformat auf dem Common Data Format (CDF) der *National Aeronautics and Space Administration (NASA)* und wird von *Unidata*[10] entwickelt, welche wiederum

[9] http://www.unidata.ucar.edu/software/netcdf/
[10] http://www.unidata.ucar.edu/

hauptsächlich von der US-amerikanischen *National Science Foundation*[11] finanziert wird. Die *Unidata netCDF License* (siehe Anhang B.3) ist eine Lizenz im MIT-Stil. Sie ist damit als frei einzustufen und mit der GPL kompatibel [FSF12]. Die Bibliothek selbst darf beliebig verändert werden und darauf aufbauende oder davon abgeleitete Software darf in Quell- oder binärer Form beliebig verbreitet werden, solange der Lizenztext enthalten ist.

Es existieren mittlerweile vier verschiedene Versionen des Dateiformats:

- *klassisch*: Dieses Format entstammt der ersten Version der netCDF-Referenz-Implementierung von Unidata und ist seit 1989 bis heute das Standardformat für neu erstellte Dateien. Dieses Format wird oftmals auch als *netCDF3* bezeichnet [RD90].

- *64-bit offset*: Dieses 2004 eingeführte Dateiformat ist nahezu identisch zur klassischen Variante, hebt aber die Maximalgröße für Datensätze von 2GB auf 4GB an [RDE+12].

- *netCDF4*: 2008 eingeführt, erlaubt dieses Format die Komprimierung von Teilen der Datei (pro Variable bzw. Datensatz), die Verwendung von komplexen Datentypen und mehreren unlimitierten Dimensionen. Darüber hinaus unterstützt es paralleles I/O. Technisch basiert netCDF4 auf HDF5 [HR08].

- *netCDF4 klassisch*: Zusammen mit netCDF4 wurde dieses Format eingeführt, das die Performance-Verbesserungen des neuen Formats ohne die Komplexität der dafür nötigen, neuen Programmierschnittstellen oder des veränderten Datenmodells nutzbar machen soll [RDE+12]. Es bietet sich somit besonders für die Migration bestehender Anwendungen an.

Die Bibliothek ist in der Programmiersprache C geschrieben. Es existieren Schnittstellen für C, C++, Fortran 77 und Fortran 90. Daneben existiert eine offizielle, ebenfalls von Unidata stammende Implementierung in Java[12]. Weiterhin gibt es Portierungen und Wrapper von Drittanbietern u. a. für .NET[13] und MATLAB[14].

netCDF wurde auf verbreiteten Betriebsystemen (konkret AIX, HPUX, IRIX/ IRIX64, Linux, Mac OS X, Solaris und Windows) erfolgreich getestet [Rew12]. Ein Programm, das den in Abbildung 3.1 dargestellten Aufbau umsetzt, ist in Listing A.4 abgebildet. Der resultierende technische Aufbau der entstandenen Datei ist in Listing A.5 und Listing A.6 zu sehen.

Da nur netCDF4 aufgrund der genannten Eigenschaften für die vorliegende Arbeit von Relevanz ist und dieses auf HDF5 aufbaut, lassen sich die Erklärungen der

[11] http://www.nsf.gov/
[12] http://www.unidata.ucar.edu/software/netcdf-java/
[13] http://netcdf.codeplex.com/
[14] http://mexcdf.sourceforge.net/

Konzepte und des Datenmodells aus Unterabschnitt 3.2.2 entsprechend übertragen. Analog gilt die Schlussfolgerung für die Nutzbarkeit in waLBerla.

3.2.4 Zusammenfassung

Aus den drei untersuchten Bibliotheken scheidet boost::serialization aufgrund der technischen Einschränkungen aus. Da netCDF lediglich in Version 4 in Frage kommen würde und dabei auf HDF5 basiert, wurde entschieden, HDF5 als Format der Wahl für Checkpointing in waLBerla zu verwenden.

Dabei wird die C-Bibliothek direkt genutzt und nicht die C++-Wraper verwendet. Zum einen müssen bei der Installation von HDF5 die C++-Komponenten extra aktiviert werden, so dass diese möglicherweise nicht auf jedem System umittelbar verfügbar sind. Zum anderen sind diese qualitativ noch verbesserungswürdig und lassen einige Features der C-Bibliothek vermissen [RKBS10].

3.3 Aufbau des Checkpointing-Moduls

Die Struktur des Checkpointing-Moduls folgt dem üblichen Aufbau eines waLBerla-Moduls: Ein *Manager* dient als Schnittstelle zwischen Anwendung und Modul. Er übernimmt die Initialisierung und delegiert die Aufgaben an die jeweiligen spezialisierten Unterkomponenten. Im Folgenden sollen der *CheckpointingManager* und die beteiligten Komponenten vorgestellt und ihre Implementierung erklärt werden.

3.3.1 Manager

Der *Manager* ist die Zentrale des Checkpointing-Moduls und dient der Anwendung als Schnittstelle zur Funktionalität des Moduls. Er bindet eine *Strategy* (vgl. Unterabschnitt 3.3.2) ein, die festlegt, ob ein bestehender Checkpoint geladen werden kann oder ob ein neuer Checkpoint erstellt werden soll. Über weitere eingebundene Untermodule, genannt *Workers* (vgl. Unterabschnitt 3.3.3), wird festgelegt, welche Daten geschrieben und gelesen werden sollen. Die zur Verfügung stehenden Strategien und Workers werden von der Anwendung übergeben. Aus diesen wählt er gemäß der Definition in der Konfigurationsdatei die konkret zu verwendenden Objekte aus.

Eine Übersicht möglicher Konfigurationsparameter, die im Umfang dieser Arbeit entstanden sind, findet sich in Abschnitt C.1.

Wenn waLBerla um Multithread-Fähigkeiten erweitert wird, dann ist der Manager auch für den korrekten Ablauf des forked Checkpointings zuständig. Während die Workers sich durch das Kopieren der Daten vorbereiten (vgl. Unterabschnitt 3.3.3) pausiert die Simulation. Sind die Vorbereitungen abgeschlossen, dann erzeugt der Manager einen neuen Thread, in dem die Workers die Daten schreiben können. Während

dieser Zeit kann die Simulation bereits weiterlaufen. Soll ein neuer Checkpoint erstellt werden, während die Erstellung des vorherigen noch nicht abgeschlossen ist, so muss der Manager auf diesen Konflikt reagieren und die Erstellung entsprechend verzögern oder ablehnen. Hierfür prüft der Manager vorab, ob alle eingebundenen Workers das Kopieren der Daten unterstützen. Ist dies nicht der Fall, dann schaltet er auf serielles Checkpointing um. In beiden Fällen handelt es sich um synchrones Checkpointing.

3.3.2 Strategy

Die *Strategy* regelt das Verhalten des Checkpointing-Moduls. Sie entscheidet, ob ein Restart erfolgt und wann ein Checkpoint erstellt wird. Wie aus Abbildung 3.2 ersichtlich legt sie u. a. die Dateinamen fest und definiert, ob eine bereits bestehende Datei überschrieben werden soll oder nicht. Darüber hinaus kann eigene Funktionalität eingefügt werden, die vor und nach der Erstellung bzw. dem Laden eines Checkpoints ausgeführt wird. Ein Anwendungsbeispiel wäre das Rotieren der Checkpoints, d. h. nur die letzten n Dateien würden aufgehoben und ältere Dateien gelöscht werden.

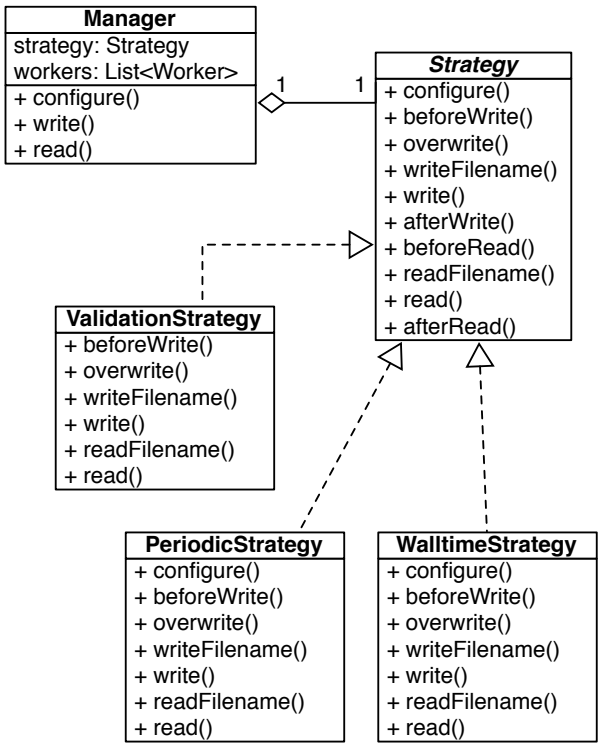

Abb. 3.2: Das Verhalten des Checkpointing-Moduls wird über eine Strategy bestimmt. Es stehen eine Vielzahl verschiedener Strategien zur Auswahl.

Die Anwendung registriert beim Manager eine Reihe von infrage kommenden Strategien. Die aus dieser Auswahl zu verwendende wird in der Konfigurationsdatei festgelegt. Mögliche Strategien sind in Abschnitt C.1 aufgeführt und umfassen u. a. die periodische Erstellung alle n Zeitschritte, die Erstellung eines Checkpoints anhand der Laufzeit der Simulation und die Validierung der Workers (vgl. Abschnitt 3.5).

Umgesetzt sind die verschiedenen Strategien als jeweils eigene Klassen, die von der abstrakten Basisklasse `Strategy` erben. Diese definiert einige Methoden, die überschrieben werden müssen:

- Mittels `write` gibt die Strategy an, ob zum Zeitpunkt des Aufrufs ein Checkpoint erstellt werden soll. In Implementierungen wird zur Prüfung üblicherweise auf den aktuellen Zeitschritt der Simulation zurückgegriffen.
- `writeFilename` definiert den Dateinamen des zu erstellenden Checkpoints. Der Manager verbindet diesen mit dem in der Konfigurationsdatei angegeben Verzeichnis. Diese Methode wird aufgerufen, wenn `write` den Wert `true` zurücklieferte.
- `read` gibt an, ob versucht werden soll, einen Checkpoint zu laden. Diese Methode wird einmalig am Ende der Initialisierung des Managers aufgerufen.
- `readFilename` spezifiziert den Dateinamen des zu ladenden Checkpoints. Diese Methode wird nur aufgerufen, wenn `read` den Wert `true` zurücklieferte. Der Manager ergänzt den Dateinamen um das konfigurierte Checkpointing-Verzeichnis.

Daneben existieren einige weitere Methoden, deren Implementierung in abgeleiteten Klassen optional ist:

- `configure` wird zu Beginn der Programmausführung aufgerufen, um beispielsweise die Konfigurationsdatei einzulesen.
- `overWrite` legt fest, ob eine Checkpoint-Datei überschrieben werden soll, falls sie bereits existiert. Standardmäßig ist dies nicht der Fall.
- `beforeWrite` und `afterWrite` werden vor bzw. nach der Erstellung des Checkpoints aufgerufen. Zu diesen Zeitpunkten ist die Checkpoint-Datei noch nicht oder nicht mehr geöffnet.
- Analog dazu werden `beforeRead` und `afterRead` vor bzw. nach dem Einlesen des Checkpoints aufgerufen. Die Checkpoint-Datei ist zu diesem Zeitpunkt noch nicht oder nicht mehr geöffnet.

Im Rahmen dieser Arbeit wurden folgende Strategien implementiert:

- Mit der *DisableStrategy* werden keine Checkpoints eingelesen oder erstellt. Das Checkpointing-Modul wird somit faktisch deaktiviert.
- Die *PeriodicStrategy* erstellt periodisch alle n Zeitschritte einen Checkpoint. In der Konfigurationsdatei der Anwendung lassen sich n und der Dateiname angeben (vgl. Anhang C.3).
- Die *WalltimeStrategy* erlaubt es, eine Ziel-Laufzeit für die Anwendung anzugeben (vgl. Anhang C.4). Ist diese Zeit abgelaufen, dann wird ein Checkpoint erstellt und die Anwendung beendet.
- Die *PeriodicWalltimeStrategy* kombiniert die periodische und die Walltime-Strategie: Solange die definierte Ziel-Laufzeit nicht abgelaufen ist werden in festgelegten Intervallen Checkpoints erstellt, andernfalls wird ein abschließender Checkpoint erstellt und die Anwendung beendet.
- *ValidationStrategy* und *IntenseValidationStrategy* dienen zur Validierung der Workers (vgl. Abschnitt 3.5).
- Zur Ermittlung der erzielten Bandbreiten (vgl. Abschnitt 4.3) dienen die *WriteBenchmarkStrategy* und die *ReadBenchmarkStrategy*. Sie messen die für den reinen Schreib- bzw. Lesevorgang benötigte Zeitdauer.

3.3.3 Workers

Die *Workers* erledigen die eigentliche Arbeit: Bei der Checkpoint-Erstellung schreiben sie die Daten in die Checkpoint-Datei, bei einem Restart stellen sie den alten Zustand aus dieser wieder her. Zusätzlich prüfen die Workers, ob der zu ladende Checkpoint zum aktuellen Anwendungszustand kompatibel ist.

Die Workers verfügen über einen Vorbereitungsmechanismus für forked Checkpointing. Vor dem eigentlichen Schreiben des Checkpoints wird vom Manager eine Kopier-Methode aufgerufen. Über diese werden die zu sichernden Daten kopiert, damit sie im Hintergrund - während die Simulation bereits wieder läuft - geschrieben werden können. Da auch sog. *DirectWorkers* denkbar sind, die Daten direkt schreiben ohne sie vorab gepuffert zu haben, müssen alle Workers dem Manager mitteilen, ob sie für forked Checkpointing geeignet sind.

Neben dem Konzept des normalen Workers existiert eine weitere, spezialisierte Form, die *ValidationWorkers*. Diese dienen der verbesserten Validierung (vgl. Abschnitt 3.5) der regulären Workers. Sie sollen Daten schreiben, die normalerweise nicht persistent gespeichert werden müssen, da sie aus anderen Daten wiederhergestellt werden können. Aus diesem Grund lesen die ValidationWorkers bei einem

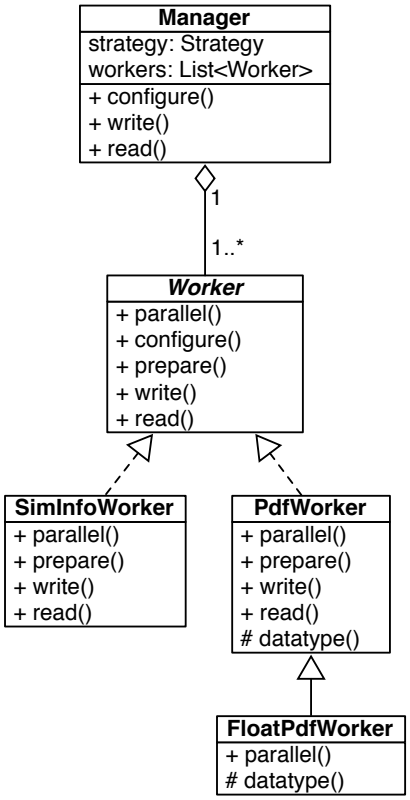

Abb. 3.3: Workers erben von einer abstrakten Basisklasse und werden vom Manager zum Lesen/Schreiben eines Checkpoints aufgerufen.

Restart keine Daten ein. Die geschriebenen Daten können aber verglichen werden, um durch einen Restart verursachte Datenänderungen zu entdecken. Eine genauere Beschreibung der Funktionsweise erfolgt in Abschnitt 3.5. Ein Beispiel für diese Daten sind in einer LBM-Simulation die Geschwindigkeits- und Dichtefelder, die aus den PDF-Feldern (*Particle Distribution Functions*) errechnet werden.

Der Modularität von waLBerla folgend soll per Konvention jeder Worker nur für einen einzigen Typ der globalen Datenobjekte (GDO) zuständig sein und innerhalb einer einzigen HDF5-Gruppe operieren.

Alle Workers müssen von der abstrakten Basisklasse `Worker` erben und mindestens folgende Methoden überschreiben:

- `parallel` gibt an, ob dieser Worker forked Checkpointing unterstützt. Die Methode wird vom Manager verwendet. Diese Eigenschaft kann bspw. auch flexibel von der aktuellen Anwendungskonfiguration abhängen.
- `prepare` kopiert die zu schreibenden Daten vorab in einen Puffer, damit sie im Hintergrund geschrieben werden können.
- `write` schreibt die gepufferten Daten in die Checkpoint-Datei. Hierfür können vorgefertigte, allgemein gehaltene Funktionen des Checkpointing-Moduls verwendet werden. Alternativ können auch die Daten ohne Pufferung direkt aus den Datenstrukturen der Anwendung geschrieben werden, bspw. abhängig vom in `parallel` zurückgegebenen Wert.
- `read` füllt mit den aus der Checkpoint-Datei gelesenen Daten die entsprechenden Datenstrukturen der Anwendung. Es können ebenfalls vorgefertigte, allgemein gehaltene Funktionen verwendet werden.

Optional können Workers die Methode `configure` überschreiben, um beispielsweise die Konfigurationsdatei einzulesen. Diese Methode wird vom Manager einmalig zum Start der Anwendung in seiner eigenen Konfigurationsphase aufgerufen.

3.3.4 Zusammenfassung

Durch die Aufteilung der Aufgaben in die Strategy und die spezialisierten Workers ist es möglich, jede Anwendung um eine maßgeschneiderte Checkpointing-Funktionalität zu erweitern. Das Verhalten des Checkpointing-Moduls lässt sich über eine eigene Strategie mit wenigen Zeilen Programmcode verändern und an eigene Wünsche anpassen. Durch das Konzept der Workers, die von einer Vielzahl verschiedener Anwendungen eingebunden werden, besteht ein hoher Wiederverwendungsgrad. Es muss also nur wenig neuer Programmcode geschrieben werden, um weitere Anwendungen sichern und wiederherstellen zu können. Der erforderliche Programmieraufwand wird durch die Bereitstellung fertiger Funktionen, die einfach aufgerufen werden können, weiter reduziert.

3.4 Einbindung und Ablauf

Unter der Prämisse, dass die Anwendung mit der selben Konfiguration wie beim vorherigen Programmlauf neu gestartet wird und keine wesentlichen Änderungen am Programmcode durchgeführt wurden, sind die strukturellen Daten, wie beteiligte Module oder Form und Größe des Gebiets, identisch. In einem Checkpoint muss also nur die Differenz zwischen Ausgangs- und fortgeschrittenem Zustand gesichert werden, wodurch sich sowohl die Schreib- als auch die Ladezeit verringern (vgl. inkrementelles Checkpointing, Unterabschnitt 2.1.5).

Um das Checkpointing-Modul nutzbar zu machen, muss der CheckpointingManager als letztes der GDO, aus denen die Anwendung besteht, eingebunden und initialisiert werden. Hierbei werden auch die zur Anwendung und ihren Datenstrukturen kompatiblen Workers und Strategies integriert und beim Manager über die Methode `reg` registriert. Die Konfigurationsdatei der Anwendung muss entsprechend ergänzt werden, indem die aufzurufenden Workers spezifiziert sowie ggf. ihre Konfigurationsparameter angegeben werden. Wird die Anwendung gestartet, dann kann davon ausgegangen werden, dass alle anderen benötigten Module bereits fertig konfiguriert sind, so dass nach dem Einlesen der Checkpointing-Konfiguration, falls vorhanden, mit dem Laden eines Checkpoints begonnen werden kann. Danach kann die Simulation ab dem geladenen Punkt wieder aufgenommen werden.

Der Schreibbefehl des Managers wird von der Anwendung am Ende jedes Zeitschritts aufgerufen. Der Manager entscheidet dann, ob ein Checkpoint erstellt werden soll (vgl. Strategy, Unterabschnitt 3.3.2). Fällt das Ergebnis positiv aus, dann erstellt er eine Datei, in die nacheinander die beteiligten Workers schreiben. Anschließend schließt der Manager die Datei und die Simulation kann weiterlaufen.

Wurde waLBerla um Multithreading-Fähigkeiten erweitert, dann kommt ein zusätzlicher Schritt hinzu: Nach erfolgreicher Prüfung, ob ein Checkpoint erstellt werden soll, werden die Vorbereitungsmethoden der Workers aufgerufen. Nachdem diese alle benötigten Daten kopiert haben kann die Simulation fortgeführt werden. Parallel dazu wird, wie im seriellen Fall, die Checkpoint-Datei geschrieben.

3.5 Validierung

Die *Validierung* soll zeigen, dass die Simulationsergebnisse stets gleich sind, unabhängig davon, ob die Anwendung neu gestartet wurde oder nicht. Sie unterstützt dadurch die Entwicklung neuer Workers.

Der Ablauf der Validierung, als Diagramm dargestellt in Abbildung 3.4, entspricht der regulären Programmausführung und kommt ohne separate Prüfungsfunktionalität in den Workers oder Änderungen im Programmcode aus. Die Anwendung berechnet einige Zeitschritte, um so die relevanten Datenstrukturen zu füllen, und erstellt dann den Checkpoint $C_{0,0}$. Anschließend läuft die Simulation ohne Neustart weiter

und schreibt am Programmende Checkpoint $C_{0,1}$. Die Anwendung wird mit identischer Konfigurationsdatei erneut (ggf. mit anderer Prozessoranzahl) gestartet. Sie liest dabei den ersten Checkpoint $C_{0,0}$ ein. Die verbleibenden Zeitschritte werden auf Basis der eben geladenen Daten berechnet und am Ende wird wiederum ein Checkpoint ($C_{1,1}$) erstellt. Bei fehlerfreier Funktion der beteiligten Workers sind $C_{0,1}$ und $C_{1,1}$ identisch.

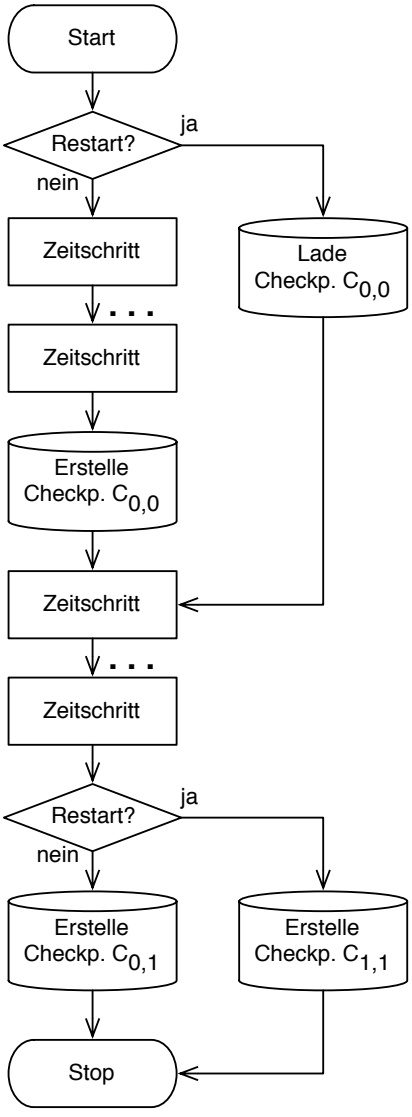

Abb. 3.4: Ablaufdiagramm der Validierung. Es wird geprüft, ob Daten- bzw. Ergebnisdifferenzen durch Restarts auftreten. Verglichen werden die jeweils am Ende der Programmausführung entstandenen Checkpoints.

Neben der Standard-Validierung existiert noch ein „intensiver“ Validierungsmodus: In dieser Betriebsart wird - wie beim Standardverfahren - nach einigen Zeitschritten

ein Checkpoint erstellt, der als Ausgangsbasis für den Neustart der Simulation dient. Es folgt jedoch nicht ein einzelner Checkpoint am Ende der Ausführung, sondern es werden kontinuierlich nach jedem Zeitschritt Checkpoints erstellt. Ebenso werden am Ende jeden Zeitschritts der neu gestarteten Simulation Checkpoints erstellt, die dann mit den Referenzdaten der Simulation ohne Neustart verglichen werden können.

Umgesetzt wird der Vergleich der erstellten Dateien mit dem Programm `h5diff`, das sich im Lieferumfang der HDF5-Bibliothek befindet. Es vergleicht zwei Dateien und meldet Unterschiede zwischen diesen. Es ist möglich, den Vergleich nur auf bestimmte Teile der Datei einzuschränken (Angabe der jeweiligen Pfade nach den beiden Dateinamen) oder Teile auszuschließen (`--exclude-path`), sich die Unterschiede detailliert anzusehen (`-v`) oder einen Mindestwerteunterschied für den Vergleich anzugeben (`-d`). In Listing A.7 findet sich ein Bash-Skript, welches ausgehend von der ValidationStrategy vollautomatisch eine Validierung durchführt. Es können dabei Anwendung, Name der Konfigurationsdatei, sowie MPI-Parameter, wie beispielsweise die Anzahl der zu verwendenden Prozesse, eingestellt werden.

Die erstellten Dateien können mit `h5dump` betrachtet werden. Es bietet Parameter an, um nur den Aufbau der Datei anzuzeigen (`-H`) oder nur ausgewählte Gruppen (`-g`) bzw. Datasets (`-d`) anzuzeigen. Daneben existieren noch viele weitere Optionen, beispielsweise können die Daten als XML (`-x`) oder binär (`-b`) exportiert werden.

3.6 Beispielumsetzung eines Workers

Im Folgenden wird exemplarisch die Implementierung eines Workers erläutert. Als Anschauungsobjekt dient der *PdfWorker*, der die Verteilungsfunktionen jeder Zelle sichert und lädt. Des Weiteren wird auf die Implementierung des *FloatPdfWorkers* eingegangen. Dieser speichert die Werte stets mit einfacher Genaugkeit, unabhängig von der für die gesamte Anwendung eingestellten Genauigkeit (single oder double precision).

3.6.1 Deklarationen

Da sich beide Workers nur im ausgegebenen Datentyp unterscheiden, liegt es nahe, den FloatPdfWorker als Spezialisierung des PdfWorkers zu sehen und daher als abgeleitete Klasse zu realisieren. In Listing 3.1 und Listing 3.3 sind die Deklarationen der Klassen zu sehen.

Im PdfWorker werden die vier wichtigsten Methoden aus der abstrakten Basisklasse Worker überschrieben: `parallel`, `prepare`, `write` und `read`. Da keine besondere Initialisierung von Member-Variablen nötig ist und auch keine Konfigurationsdaten gelesen werden müssen, kann auf einen eigenen Konstruktor sowie auf die Überladung von `configure` verzichtet werden. In der Folge wird eine Methode `getType` definiert, die den zu verwendenden HDF5-Datentyp zurückliefert. Da sie vom abgeleiteten

FloatPdfWorker überschrieben werden soll wird sie als `protected` markiert. Schließlich wird ein Puffer deklariert, der Werte vom Typ `real_t` aufnimmt. Auf den Typ `Buffer<T>` wird in Unterabschnitt 3.6.2 genauer eingegangen. `real_t` ist ein Datentyp für Fließkommazahlen und wird zur Übersetzungszeit auf einfache oder doppelte Genauigkeit (`double` oder `float`) festgelegt.

Listing 3.1: Header-File des PdfWorkers. Vier abstrakte Methoden der Basisklasse Worker werden überschrieben. Daneben wird eine Methode zur Bestimmung des HDF5-Datentyps und ein Puffer für forked Checkpointing deklariert.

```
1  class PdfWorker : public Worker {
2  public:
3     virtual bool parallel() const;
4     virtual void prepare();
5     virtual void write(const hid_t h5_file);
6     virtual void read(const hid_t h5_file);
7
8  protected:
9     virtual hid_t getType() const;
10
11 private:
12    Buffer<real_t>::Type buffer_;
13 };
```

Listing 3.2 zeigt die Methode `parallel` des PdfWorkers. Durch das zurückliefern von `true` wird dem Manager signalisiert, dass dieser Worker forked Checkpointing unterstützt. Würde hier `false` zurückgegeben werden, dann müsste der Manager auf serielles Checkpointing umschalten.

Listing 3.2: Kennzeichnung des PdfWorkers für forked Checkpointing. Es wird signalisiert, dass dieser Worker für forked Checkpointing geeignet ist.

```
1  bool PdfWorker::parallel() const {
2     return true;
3  }
```

In der Deklaration des FloatPdfWorkers werden die Methoden `prepare` und `write` des PdfWorkers überschrieben. Die Methode `parallel` wird nicht überschrieben, da auch der PdfWorker forked Checkpointing unterstützt und entsprechend in seiner `parallel`-Methode `true` zurückliefert. Außerdem kann auf die Überschreibung von `read` verzichtet werden. Der Grund dafür wird in Unterabschnitt 3.6.3 erläutert. Da ein anderer HDF5-Datentyp verwendet werden soll wird die Methode `getType` überschrieben. Zusätzlich wird ein eigener Puffer deklariert, der nicht mehr Daten vom Typ `real_t` speichert, sondern unabhängig von der konfigurierten Fließkommapräzision stets nur einfache Genauigkeit bietet.

Listing 3.3: Header-File des FloatPdfWorkers. Hier wird die Lese-Methode der Basisklasse nicht überschrieben. Es wird ein eigener Puffer deklariert.

```
1  class FloatPdfWorker : public PdfWorker {
2  public:
3     virtual void prepare();
4     virtual void write(const hid_t h5_file);
5
6  protected:
7     virtual hid_t getType() const;
8
9  private:
10    Buffer<float>::Type buffer_;
11 };
```

3.6.2 Schreibvorgang

Der Schreibvorgang teilt sich beim forked Checkpointing in zwei logische Einheiten auf: Das Kopieren der Daten vorab, hier umgesetzt über die Methode `prepare`, sowie das persistente Schreiben, hier via `write`.

Zum Zwischenspeichern dient eine Struktur vom Typ `Buffer`, der in Listing 3.4 abgebildet ist. Er ist Teil des Checkpointing-Moduls und daher allgemeingültig gehalten, um von vielen Workers verwendet werden zu können. Die Definition dieses Typs erfolgt in der Datei `FieldWorkerHelper.h`. Er ist wie folgt aufgebaut:

Listing 3.4: Typdeklarationen der Pufferstruktur zum Zwischenspeichern der Daten, bevor diese persistent geschrieben werden.

```
1  template<typename T> struct Buffer {
2     typedef boost::array<uint_t, 6> Array;       // Position und Groesse eines Blocks
3     typedef std::pair< Array, std::vector<T> > Pair; // entspricht einem Block
4     typedef std::list<Pair> Type;                    // repraesentiert alle Bloecke
5  };
```

In Variablen vom Typ `Array` wird die Position und die Größe des Blocks gespeichert, der die zu sichernden Daten enthält. Die zugehörigen Nutzdaten werden linear in einem `std::vector` gehalten. Aus diesen beiden Teilen wird ein `std::pair` gebildet. Alle Paare werden in einer `std::list` gespeichert.

In der Methode `prepare` muss eine solche Pufferstruktur gefüllt werden. Wie in Listing 3.5 ersichtlich wird die Funktion `prepareWriteImpl` aufgerufen, die allgemeingültig und daher ebenfalls in der Datei `FieldWorkerHelper.h` definiert ist. Dabei sind einige Template-Parameter zu übergeben. Konkret sind dies der Typ des zu kopierenden Feldes (hier `lbm::sweep::D3Q19PDFField`), der zu verwendende Datentyp (`real_t`) und die Anzahl der Elemente pro Zelle (19). An dynamischen Parametern (und daher nicht per Template umsetzbar) kommen die Kennzeichnung der zu sichernden Daten innerhalb des Blocks (`lbm::getSrcUID()`) und die Puffer-Instanz, die die Daten halten soll (`buffer_`), hinzu.

Listing 3.5: Vorbereitungsmethode des PdfWorkers. Es wird eine vom Checkpointing-Modul bereitgestellte Methode mit entsprechenden Parametern aufgerufen.

```
1  void PdfWorker::prepare() {
2     prepareWriteImpl<lbm::sweep::D3Q19PDFField, real_t, 19>(lbm::getSrcUID(),
3                                                             buffer_);
4  }
```

Gemäß Listing 3.6 sieht der Aufruf im FloatPdfWorker fast identisch aus. Es muss nur der Datentyp von `real_t` auf `float` geändert werden. Da die Bezeichnung des Puffers mit der der Basisklasse übereinstimmt, muss hier keine Änderung vorgenommen werden. Die restlichen Angaben beziehen sich auf das zu sichernde Feld und ändern sich damit nicht. Es wäre auch möglich, die Vorbereitungsmethode des PdfWorkers zu verwenden, doch würde dadurch Arbeitsspeicher verschwendet werden, wenn `real_t` mit doppelter Genauigkeit definiert ist.

Listing 3.6: Vorbereitungsmethode des FloatPdfWorkers. Einziger Unterschied im Vergleich zum PdfWorker ist der Wechsel des Datentyps.

```
1  void FloatPdfWorker::prepare() {
2     prepareWriteImpl<lbm::sweep::D3Q19PDFField, float, 19>(lbm::getSrcUID(),
3                                                            buffer_);
4  }
```

Die allgemein gehaltene und vom Checkpointing-Modul bereit gestellte Vorbereitungsfunktion `prepareWriteImpl` ist in Listing 3.7 abgebildet. Zu Beginn werden der Puffer geleert und die Voraussetzungen geschaffen, um auf die Blockdaten zugreifen zu können. Danach wird über alle im lokalen Prozess allozierten Blöcke iteriert. Um auf teure Reallozierungen verzichten zu können wird vorab der benötigte Speicher reserviert. Danach werden die Position des Blocks und seine Größe (abzüglich des Ghost-Layers) in den Puffer geschrieben. Schließlich wird der reservierte Speicher mit den ausgelesen Werten gefüllt.

Listing 3.7: Die vom Checkpointing-Modul bereitgestellte, allgemein gehaltene Vorbereitungsfunktion, um zu sichernde Daten vorab in einen Puffer zu kopieren.

```
1  template<typename FIELD_T, typename VALUE_T, unsigned int SIZE>
2  void prepareWriteImpl(const core::BDUID fieldUid,
3                        typename Buffer<VALUE_T>::Type& target) {
4     target.clear(); // Puffer leeren
5
6     gd::PatchManagerID manager =
7        gd::getObject<gd::PatchManager>(gd::getPatchManagerUID());
8     pd::PatchID patch = manager->getRootPatch();
9     pd::BlockField& field = patch->getBlockField();
10    pd::BlockFieldDataID fielddata = field.getData();
11
12    for(pd::BListConstIter blockIt = field.allocBegin(); blockIt!=field.allocEnd();
13        blockIt++) {
14       boost::shared_ptr<FIELD_T> src = (*blockIt)->getBD<FIELD_T>(fieldUid);
15
16
```

```
17          // Groessen ermitteln
18          const uint_t xSize = src->xSize();
19          const uint_t ySize = src->ySize();
20          const uint_t zSize = src->zSize();
21
22          // Speicher reservieren
23          target.push_back(
24             typename Buffer<VALUE_T>::Pair(
25                typename Buffer<VALUE_T>::Array(),
26                std::vector<VALUE_T>((xSize - 2) * (ySize - 2) * (zSize - 2) * SIZE)
27             )
28          );
29
30          // Pointer auf reservierten Speicher
31          typename Buffer<VALUE_T>::Pair* pair = &(target.back());
32
33          // Koordinaten und Groesse des Blocks speichern
34          pair->first[0] = ((*blockIt)->getID()[0] - 1) * (xSize - 2);
35          pair->first[1] = ((*blockIt)->getID()[1] - 1) * (ySize - 2);
36          pair->first[2] = ((*blockIt)->getID()[2] - 1) * (zSize - 2);
37          pair->first[3] = src->xSize() - 2;
38          pair->first[4] = src->ySize() - 2;
39          pair->first[5] = src->zSize() - 2;
40
41          // Werte speichern
42          uint_t idx = 0;
43          for(uint_t z = 1; z < zSize - 1; ++z) {
44             for(uint_t y = 1; y < ySize - 1; ++y) {
45                for(uint_t x = 1; x < xSize - 1; ++x) {
46                   for(uint_t f = 0; f < SIZE; ++f) {
47                      pair->second[idx++] =
48                         static_cast<VALUE_T>(src->get(x, y, z, f));
49                   }
50                }
51             }
52          }
53       }
54    }
```

Ist der Puffer gefüllt, dann kann der Checkpoint im Hintergrund geschrieben werden. Der PdfWorker stellt hierzu die in Listing 3.8 abgebildete `write`-Methode zur Verfügung. Es ist zu sehen, dass zuerst eine HDF5-Gruppe namens „PDF“ erstellt wird. Danach wird eine vom Checkpointing-Modul bereitgestellte, allgemein gehaltene Funktion `bufferedWriteImpl` aufgerufen. Dieser werden, analog zur Vorbereitungsfunktion, der Typ der Daten im Speicher (`real_t`) und die Anzahl der Werte pro Zelle (19) übergeben, sowie die Angabe, ob die Daten - sofern möglich - komprimiert geschrieben werden sollen. An dynamischen Parametern kommen zwei HDF5-Parameter dazu: Die zu verwendende Gruppe (`h5_group`) und der zu verwendende Datentyp (via `getType()`, vgl. Listing 3.9). Im Allgemeinen sollten nur die „nativen“ HDF5-Datentypen verwendet werden, da diese nicht vor dem Schreiben konvertiert werden müssen und dadurch keine zusätzlichen Performance-Einbußen verursachen. Den letzten Parameter bildet der zu schreibende Puffer, der in `prepare` mit Daten gefüllt wurde. Nach dem Schreiben der Daten muss die HDF5-Gruppe wieder geschlossen werden.

Listing 3.8: Schreib-Aufruf des PdfWorkers. Zuerst wird die HDF5-Gruppe erstellt, dann erfolgt der eigentliche Schreibbefehl über eine vom Checkpointing-Modul bereitgestellte, allgemeine Funktion.

```
1 void PdfWorker::write(const hid_t h5_file) {
2    hid_t h5_group = H5Gcreate(h5_file, "PDF", H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
3    bufferedWriteImpl<real_t, 19, false>(h5_group, getType(), buffer_);
4    H5Gclose(h5_group);
5 }
```

Listing 3.9: Der HDF5-Datentyp, der beim PdfWorker zum Schreiben der Daten verwendet werden soll, ist je nach Konfiguration mit einfacher oder doppelter Genauigkeit definiert.

```
1 hid_t PdfWorker::getType() const {
2    return H5T_NATIVE_REAL; // schreibe real_t (double oder float, je nach Konfig.)
3 }
```

Beim FloatPdfWorker sieht die Schreibmethode, abgebildet in Listing 3.10, wieder nahezu identisch aus. Die Gruppenbezeichnung ändert sich nicht, da die Lesemethode des PdfWorkers die Daten bei Bedarf von einfacher auf doppelte Genauigkeit konvertiert. Beim eigentlichen Schreibaufruf wird lediglich der Datentyp des Puffers auf `float` beschränkt. `getType` (vgl. Listing 3.11) liefert in diesem Fall ebenfalls einen Datentyp mit einfacher Genauigkeit zurück.

Listing 3.10: Schreib-Aufruf des FloatPdfWorkers. Im Vergleich zum PdfWorker ändert sich lediglich der Datentyp des Puffers im Arbeitsspeicher.

```
1 void FloatPdfWorker::write(const hid_t h5_file) {
2    hid_t h5_group = H5Gcreate(h5_file, "PDF", H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
3    bufferedWriteImpl<float, 19, false>(h5_group, getType(), buffer_);
4    H5Gclose(h5_group);
5 }
```

Listing 3.11: Der FloatPdfWorker verwendet zum Schreiben der Daten stets nur einfache Genauigkeit.

```
1 hid_t FloatPdfWorker::getType() const {
2    return H5T_NATIVE_FLOAT;
3 }
```

Die tatsächliche Schreibfunktion ist in Listing 3.12 abgebildet. Es wird zunächst die Zugriffsart und die globale Gesamtgröße der zu schreibenden Daten ermittelt und ein HDF5-Datentyp für Arrays der entsprechenden Größe definiert, um einen entsprechenden Dataspace und ein Dataset zu erstellen. Um die Daten korrekt schreiben zu können, wird ein HDF5-Datentyp für Arrays der entsprechenden Größe definiert. Zur Verbesserung der Schreib-Performance wird dieser als *chunked*, d.h. in

Blöcke unterteilt, definiert. Die Größe jedes Dataset-Blocks entspricht der waLBerla-Blockgröße. Der zugehörige Cache soll maximal zwei derartiger Blöcke umfassen. Die geschriebenen Blöcke werden in einer großen Hash-Tabelle verwaltet, so dass keine aufwändigen Hash-Kollisionen behandelt werden müssen. Durch die Unterteilung in Blöcke wird auch die per Parameter aktivierbare Komprimierung der Daten beim Schreiben möglich. Aktuell wird nur die Komprimierung von einfachen Datentypen von HDF5 unterstützt, daher erfolgt eine entsprechende Prüfung [HDF12b]. Schließlich wird über die im Puffer gehaltenen Daten iteriert. Mit den Koordinaten und Abmessungen der Blöcke, sowie der Anzahl der Werte pro Zelle, wird eine Auswahl (*Hyperslab*) auf das erstellte Dataset definiert, mit welcher die gepufferten Daten an die richtige Stelle in der Datei geschrieben werden. Zum Schluss werden die verwendeten Handles wieder geschlossen.

Listing 3.12: Die vom Checkpointing-Modul bereitgestellte, allgemein gehaltene Schreibfunktion, um die vorab gepufferten Daten in eine HDF5-Datei zu schreiben.

```
1  template<typename VALUE_T, unsigned int SIZE, bool COMPRESSION>
2  void bufferedWriteImpl(const hid_t h5_location, const hid_t h5_type,
3                         typename Buffer<VALUE_T>::Type& buffer) {
4     gd::PatchManagerID manager =
5        gd::getObject<gd::PatchManager>(gd::getPatchManagerUID());
6     pd::PatchID patch = manager->getRootPatch();
7     pd::BlockField& field = patch->getBlockField();
8     pd::BlockFieldDataID fielddata = field.getData();
9
10    // Zugriffsart ermitteln
11    hid_t h5_dataaccess = Manager::instance()->H5DataAccess();
12
13    // Datentyp fuer Array anlegen
14    hsize_t h5t_mytype_dims[1] = {SIZE};
15    hid_t h5t_mytype = H5Tarray_create(h5_type, 1, h5t_mytype_dims);
16
17    // Gesamtgroesse des Feldes ermitteln
18    hsize_t h5_dims[] = {fielddata->xSize() * fielddata->xNumBlocks(),
19                         fielddata->ySize() * fielddata->yNumBlocks(),
20                         fielddata->zSize() * fielddata->zNumBlocks()};
21
22    // Dataspace mit ermittelten Dimensionen anlegen
23    hid_t h5_file_dataspace = H5Screate_simple(3, h5_dims, NULL);
24    if(h5_file_dataspace < 0) {
25       LOG_ERROR("checkpointing::FieldWorkerHelper::bufferedWriteImpl(): " <<
26                 "Invalid h5_file_dataspace!");
27    }
28
29    // Chunked Dataset anlegen
30    pd::BListConstIter blockIt = field.allocBegin();
31    hid_t h5p_dataset = H5Pcreate(H5P_DATASET_CREATE);
32    hsize_t h5_chunk_dims[] = {(*blockIt)->xSize(), (*blockIt)->ySize(),
33                               (*blockIt)->zSize()};
34    H5Pset_chunk(h5p_dataset, 3, h5_chunk_dims);
35
36    // Komprimierung funktioniert nur mit einfachen Datentypen
37    BOOST_STATIC_ASSERT_MSG(!COMPRESSION ||
38       (COMPRESSION && (SIZE == 1) && is_arithmetic<VALUE_T>::value),
39       "Compression may only be used with arithmetic data types!");
40
41
```

```
42    // ggf. Komprimierung aktivieren
43    if(COMPRESSION) {
44       if(checkpointing::Manager::instance()->getFileAccess() == POSIX) {
45          H5Pset_szip (h5p_dataset, H5_SZIP_EC_OPTION_MASK,
46                       min(h5_chunk_dims[0] * h5_chunk_dims[1] *
47                           h5_chunk_dims[2] * SIZE, 32));
48       } else {
49          LOG_WARNING("checkpointing::FieldWorkerHelper::bufferedWriteImpl(): " <<
50                      "Parallel file access, no compression support.");
51       }
52    }
53
54    hid_t h5p_dataset_access = H5Pcreate(H5P_DATASET_ACCESS);
55
56    hsize_t h5p_dataset_chunkcache = h5_chunk_dims[0] * h5_chunk_dims[1] *
57       h5_chunk_dims[2] * SIZE * sizeof(VALUE_T) * 2;
58    hsize_t h5p_dataset_chunkslots = fielddata->xNumBlocks() *
59       fielddata->yNumBlocks() * fielddata->zNumBlocks() * 200 + 1;
60    H5Pset_chunk_cache(h5p_dataset_access, h5p_dataset_chunkslots,
61                       h5p_dataset_chunkcache, 1);
62    hid_t h5_dataset = H5Dcreate(h5_location, "Data", h5t_mytype, h5_file_dataspace,
63                                 H5P_DEFAULT, h5p_dataset, h5p_dataset_access);
64    if(h5_dataset < 0) {
65       LOG_ERROR("checkpointing::FieldWorkerHelper::bufferedWriteImpl(): " <<
66                 "Invalid h5_dataset!");
67    }
68
69    // ueber gepufferte Daten iterieren und diese schreiben
70    for(typename Buffer<VALUE_T>::Type::const_iterator it = buffer.begin();
71        it != buffer.end(); it++) {
72       hsize_t h5_file_offset[] = { it->first[0], it->first[1], it->first[2] };
73       hsize_t h5_file_length[] = { it->first[3], it->first[4], it->first[5] };
74
75       // Hyperslab in der Datei ermitteln
76       hid_t h5_hyperslab_file = H5Dget_space(h5_dataset);
77       if(h5_hyperslab_file < 0) {
78          LOG_ERROR("checkpointing::FieldWorkerHelper::bufferedWriteImpl(): " <<
79                    "Invalid h5_hyperslab_file!");
80       }
81
82       if(H5Sselect_hyperslab(h5_hyperslab_file, H5S_SELECT_SET, h5_file_offset,
83                              NULL, h5_file_length, NULL) < 0) {
84          LOG_ERROR("checkpointing::FieldWorkerHelper::bufferedWriteImpl(): " <<
85                    "Could not select H5 hyperslab!")
86       }
87
88       // Memoryspace ermitteln
89       hsize_t h5_memory_length[] =
90          {static_cast<hsize_t>(it->first[3] * it->first[4] * it->first[5])};
91       hid_t h5_memoryspace = H5Screate_simple(1, h5_memory_length, NULL);
92
93       if(H5Dwrite(h5_dataset, h5t_mytype, h5_memoryspace, h5_hyperslab_file,
94                   h5_dataaccess, &(it->second.front())) < 0) {
95          LOG_ERROR("checkpointing::FieldWorkerHelper::bufferedWriteImpl(): " <<
96                    "Could not write to HDF5 file!")
97       }
98       H5Sclose(h5_memoryspace);
99       H5Sclose(h5_hyperslab_file);
100   }
101
102   // Alle Handles schliessen
103   H5Dclose(h5_dataset);
104   H5Pclose(h5p_dataset);
```

```
105     H5Sclose(h5_file_dataspace);
106     H5Tclose(h5t_mytype);
107     H5Pclose(h5_dataaccess);
108 }
```

3.6.3 Lesevorgang

Der PdfWorker kann Daten in einfacher oder doppelter Fließkomma-Genauigkeit einlesen. Ermöglicht wird dies durch die HDF5-Bibliothek, die Daten aus dem Datentyp der Quelle (in diesem Fall die Checkpoint-Datei) in den angegebenen Ziel-Datentyp konvertiert. Aus diesem Grund kann, wie in Listing 3.3 zu sehen, auf eine Implementierung der `read`-Methode im FloatPdfWorker verzichtet werden.

Die Lesemethode des PdfWorkers ist in Listing 3.13 abgebildet. Zuerst wird die zu lesende HDF5-Gruppe geöffnet. Anschließend wird eine vom Checkpointing-Modul bereitgestellte, allgemeine Funktion aufgerufen, deren Parameter denen der Vorbereitungs- und der Schreiben-Funktion entsprechen. Sie werden daher an dieser Stelle nicht mehr gesondert erläutert. Hervorzuheben ist, dass die Daten unabhängig von dem beim Schreiben verwendeten Worker (und in Folge dessen unabhängig davon, ob die Daten in einfacher oder doppelter Fließkomma-Genauigkeit vorliegen), stets in den Datentyp `real_t` konvertiert werden. Nach dem Lesevorgang wird die HDF5-Gruppe wieder geschlossen.

Listing 3.13: Lese-Methode des PdfWorkers. Zuerst wird die HDF5-Gruppe geöffnet, dann erfolgt der eigentliche Lesebefehl in Form einer vom Checkpointing-Modul bereitgestellten, allgemeinen Funktion.

```
1 void PdfWorker::read(const hid_t h5_file) {
2     hid_t h5_group = H5Gopen(h5_file, "PDF", H5P_DEFAULT);
3     readImpl<lbm::sweep::D3Q19PDFField, real_t, 19>(h5_group, getType(),
4                                                     lbm::getSrcUID());
5     H5Gclose(h5_group);
6 }
```

Die allgemein gehaltene Lesefunktion ist in Listing 3.14 abgebildet. Analog zur Schreibfunktion werden die Zugriffsart ermittelt und das Dataset und der zugehörige Dataspace geöffnet. Dann wird geprüft, ob die Gesamtgröße des einzulesenden Gebiets mit der aktuellen Konfiguration übereinstimmt. Ist dies nicht der Fall wird der Ladevorgang abgebrochen. Nachdem ein Datentyp für Arrays der übergebenen Länge angelegt wurde, wird mit dem Einlesen der Daten für die prozesslokal allozierten Blöcke begonnen. Hierfür wird Offset und Länge der Elemente in der Datei berechnet, um diese entsprechend mittels Hyperslab auswählen zu können. Schließlich werden die Daten in einen Puffer gelesen, bevor sie in die Blockstrukturen kopiert werden. Zum Schluss werden die HDF5-Handles geschlossen. Es ist keine separate Behandlung von komprimierten Datensätzen nötig.

Listing 3.14: Die vom Checkpointing-Modul bereitgestellte, allgemein gehaltene Lesefunktion, um Daten aus der Checkpoint-Datei zu laden.

```
1  template<typename FIELD_T, typename VALUE_T, unsigned int SIZE>
2  void readImpl(const hid_t h5_location, const hid_t h5_type,
3                const core::BDUID fieldUid) {
4     gd::PatchManagerID manager =
5        gd::getObject<gd::PatchManager>(gd::getPatchManagerUID());
6     pd::PatchID patch = manager->getRootPatch();
7     pd::BlockField& field = patch->getBlockField();
8     pd::BlockFieldDataID fielddata = field.getData();
9
10    // Zugriffsart ermitteln
11    hid_t h5_dataaccess = Manager::instance()->H5DataAccess();
12
13    // Dataset oeffnen
14    hid_t h5_dataset = H5Dopen(h5_location, "Data", H5P_DEFAULT);
15    if(h5_dataset < 0) {
16       LOG_ERROR("checkpointing::FieldWorkerHelper::readImpl(): " <<
17                 "Invalid h5_dataset!");
18    }
19
20    hid_t h5_hyperslab_file = H5Dget_space(h5_dataset);
21    if(h5_hyperslab_file < 0) {
22       LOG_ERROR("checkpointing::FieldWorkerHelper::readImpl(): " <<
23                 "Invalid h5_hyperslab_file!");
24    }
25
26    // Gesamtgroesse des Feldes ermitteln und pruefen
27    hsize_t mem_dims[] = {fielddata->xSize() * fielddata->xNumBlocks(),
28                          fielddata->ySize() * fielddata->yNumBlocks(),
29                          fielddata->zSize() * fielddata->zNumBlocks()};
30
31    hsize_t h5_dims[] = {0, 0, 0};
32    H5Sget_simple_extent_dims(h5_hyperslab_file, h5_dims, NULL);
33
34    if(h5_dims[0] != mem_dims[0] || h5_dims[1] != mem_dims[1]
35       || h5_dims[2] != mem_dims[2]) {
36       LOG_ERROR("checkpointing::FieldWorkerHelper::readImpl(): " <<
37                 "Dimensions of checkpoint and current memory do not match!");
38    }
39
40    // Datentyp fuer Array anlegen
41    hsize_t h5t_mytype_dims[1] = {SIZE};
42    hid_t h5t_mytype = H5Tarray_create(h5_type, 1, h5t_mytype_dims);
43
44    for(pd::BListConstIter blockIt = field.allocBegin(); blockIt!=field.allocEnd();
45        blockIt++) {
46       boost::shared_ptr<FIELD_T> src = (*blockIt)->getBD<FIELD_T>(fieldUid);
47
48       const uint_t xSize = src->xSize();
49       const uint_t ySize = src->ySize();
50       const uint_t zSize = src->zSize();
51
52       // Puffer anlegen
53       std::vector<VALUE_T> temp((xSize - 2) * (ySize - 2) * (zSize - 2) * SIZE);
54
55       hsize_t h5_file_offset[] = {
56                static_cast<hsize_t>(((*blockIt)->getID()[0] - 1) * (xSize - 2)),
57                static_cast<hsize_t>(((*blockIt)->getID()[1] - 1) * (ySize - 2)),
58                static_cast<hsize_t>(((*blockIt)->getID()[2] - 1) * (zSize - 2))
59       };
60
```

```
61        hsize_t h5_file_length[] = {xSize - 2, ySize - 2, zSize - 2};
62        if(H5Sselect_hyperslab(h5_hyperslab_file, H5S_SELECT_SET, h5_file_offset,
63                               NULL, h5_file_length, NULL) < 0) {
64           LOG_ERROR("checkpointing::FieldWorkerHelper::readImpl(): " <<
65                     "Could not select H5 hyperslab!")
66        }
67
68        // Memory space auswaehlen
69        hsize_t h5_memory_length[] =
70           {static_cast<hsize_t>((xSize - 2) * (ySize - 2) * (zSize - 2))};
71        hid_t h5_memoryspace = H5Screate_simple(1, h5_memory_length, NULL);
72
73        if(H5Dread(h5_dataset, h5t_mytype, h5_memoryspace, h5_hyperslab_file,
74                   h5_dataaccess, &(temp.front())) < 0) {
75
76           LOG_ERROR("checkpointing::FieldWorkerHelper::readImpl(): " <<
77                     "Could not read from HDF5 file!")
78        }
79
80        // Daten aus Puffer zuweisen
81        uint_t idx = 0;
82        for(uint_t z = 1; z < zSize - 1; ++z) {
83           for(uint_t y = 1; y < ySize - 1; ++y) {
84              for(uint_t x = 1; x < xSize - 1; ++x) {
85                 for(uint_t f = 0; f < SIZE; ++f) {
86                    src->get(x, y, z, f) =
87                       static_cast<VALUE_T>(temp[idx]);
88                    idx++;
89                 }
90              }
91           }
92        }
93     }
94
95     // Alle Handles schliessen
96     H5Tclose(h5t_mytype);
97     H5Sclose(h5_hyperslab_file);
98     H5Dclose(h5_dataset);
99     H5Pclose(h5_dataaccess);
100 }
```

3.7 Beispielumsetzung einer Strategy

Im Folgenden wird die Umsetzung einer Strategy am Beispiel der PeriodicWalltime-Strategy (vgl. Unterabschnitt 3.3.2) erklärt.

Die PeriodicWalltimeStrategy definiert u. a. die in Listing 3.15 abgebildeten Member-Variablen. Diese werden von der Methode `configure` im Rahmen der vom CheckpointingManager initiierten Konfigurationsphase mit Werten aus der Konfigurationsdatei der Anwendung gefüllt. Da dies über die zum waLBerla-Kern gehörenden Komponenten geschieht, wird darauf nicht näher eingegangen.

Listing 3.15: Variablen für Konfigurationsparameter der PeriodicWalltimeStrategy.

```
1 uint_t tspacing_;      // Anzahl Zeitschritte zwischen Checkpoints. Aus Konfig-Datei.
2 uint_t tduration_;     // Ziel-Laufzeit der Simulation in Sekunden. Aus Konfig-Datei.
3 std::string filename_; // Ziel-Dateiname der Checkpoints. Aus Konfig-Datei.
```

Die Methode `write` dient zur Bestimmung, ob ein Checkpoint erstellt werden soll. Im vorliegenden Fall muss dafür, wie in Listing 3.16 zu sehen, eine von zwei Bedingungen erfüllt sein:

Die erste Möglichkeit ist, dass die Ziel-Laufzeit abgelaufen ist. Dies wird über die Zeitdifferenz zwischen dem Ende der Konfigurationsphase (gespeichert in der Member-Variable `start_`) und der aktuellen Uhrzeit ermittelt. Ist dies der Fall, dann wird über die Variable `exit_` vorgemerkt, dass nach der Erstellung des Checkpoints die Simulation beendet werden soll. Danach wird die Methodenausführung über die Rückgabe von `true` beendet.

Anderenfalls wird geprüft, ob die zweite Bedingung erfüllt ist und der aktuelle Zeitschritt gesichert werden soll. Dafür wird die Nummer des aktuellen Zeitschritts, gespeichert im GDO `SimInfo`, gegen die in der Konfigurationsdatei angegebene Periodendauer mittels einer Modulo-Operation verglichen. Je nach Ergebnis der Rechnung, also ob ein Rest übrig blieb, liefert der folgende Vergleich auf Gleichheit mit Null `true`, sonst `false`. Um die Methodenausführung zu beenden wird dieser Wert zurückgegeben.

Listing 3.16: Methode für die Bestimmung, ob ein Checkpoint erstellt werden soll. Geprüft wird, ob die konfigurierte Anzahl Zeitschritte erreicht oder die Ziel-Laufzeit abgelaufen ist.

```
1  virtual bool write() {
2     // Walltime-Pruefung
3     struct timeval tp;
4     gettimeofday(&tp, NULL);
5     unsigned long long now = static_cast<unsigned long long>(tp.tv_sec);
6     if(now - start_ >= tduration_) { // Zeit abgelaufen => Checkpoint erstellen
7        exit_ = true; // Beenden, wenn Checkpoint fertig
8        return true;
9     }
10
11    // Periodische Pruefung
12    const SimInfoID siminfo = getObject<SimInfo>(getSimInfoUID());
13    return siminfo->timestep_ > 0 && siminfo->timestep_ % tspacing_ == 0;
14 }
```

Soll ein neuer Checkpoint erstellt werden, dann muss dem Manager der zu verwendende Dateiname mitgeteilt werden und ob eine eventuell vorhandene Datei ersetzt werden soll. Der Dateiname wurde aus der Konfigurationsdatei eingelesen und wird, wie in Listing 3.17 zu sehen, ohne weitere Änderungen weitergegeben. Aus Listing 3.18 ist erkennbar, dass existierende Dateien stets überschrieben werden sollen.

Listing 3.17: Methoden zur Bestimmung der Dateinamen für den Schreib- und den Lesezugriff. Es wird in beiden Fällen der in der Konfigurationsdatei definierte Name zurückgegeben.

```
1 virtual const std::string writeFilename() {
2    return filename_;
3 }
4
5 virtual const std::string readFilename() {
6    return filename_;
7 }
```

Listing 3.18: Methode für die Bestimmung, ob eine vorhandene Datei überschrieben werden soll. Eine vorhandene Datei soll immer überschrieben werden.

```
1 virtual const bool overwrite() {
2    return true;
3 }
```

Nachdem der Checkpoint erstellt wurde ruft der Manager die in Listing 3.19 abgebildete Methode `afterWrite` auf. War das Ergebnis der Prüfung, ob ein Checkpoint erstellt werden soll, aufgrund der abgelaufenen Zeit positiv, dann ist die Member-Variable `exit_` auf `true` gesetzt. In diesem Fall wird die Simulation beendet, indem über das *SimInfo*-Objekt die Anzahl der zu rechnenden Zeitschritte auf den aktuellen Zeitschritt herabgesetzt wird. Dadurch werden beim Programmende auch offene Dateien geschlossen sowie abschließende Statistiken ermittelt und gespeichert.

Listing 3.19: Methode, die nach der Erstellung des Checkpoints ausgeführt wird. Es wird ggf. die Simulation beendet, indem die Zielanzahl der Zeitschritte auf den aktuellen Zeitschritt herabgesetzt wird.

```
1 virtual void afterWrite() {
2    if(exit_) {
3       LOG_WARNING("checkpointing::PeriodicWalltimeStrategy::afterWrite(): " <<
4                   "Time elapsed; ending simulation!");
5       const SimInfoID siminfo = getObject<SimInfo>(getSimInfoUID());
6       siminfo->numTimeSteps_ = siminfo->timestep_;
7    }
8 }
```

Bei Anwendungsstart wird, wie in Listing 3.20 zu sehen, anhand der Existenz der Checkpoint-Datei entschieden, ob ein Restart versucht werden soll. Da der Strategy lediglich der Dateiname, nicht aber der komplette Pfad, bekannt ist bietet der Manager eine entsprechende Prüfmethode an. Das hartkodierte Verzeichnistrennzeichen `/` wird dabei entsprechend der Betriebssystemkonvention aufgelöst.

Listing 3.20: Methoden für die Bestimmung, ob ein Restart erfolgen soll. Falls eine entsprechende Datei existiert soll versucht werden, den Checkpoint zu laden.

```
1 virtual bool read() {
2    return Manager::instance()->fileExists(this->readFilename());
3 }
4
5 bool Manager::fileExists(std::string filename) const {
6    std::stringstream ss;
7    ss << this->getPath() << "/" << filename;
8    return filesystem::exists(ss.str()) && filesystem::is_regular_file(ss.str());
9 }
```

3.8 Zusamenfassung

Die Architektur des Checkpointing-Moduls folgt dem üblichen Aufbau von Modulen in waLBerla. Die interne Organisation gewährleistet hohe Flexibilität. Um einzelne Anwendungen gezielt und schnell um Checkpointing-Funktionalität erweitern zu können, müssen nur spezielle Teile implementiert werden. Im Allgemeinen genügt es, in den selbst erstellten Workers fertige Funktionen aufzurufen, um Daten sichern und laden zu können. Um die korrekte Arbeitsweise zu überprüfen wird den Entwicklern mit der Validierung ein einfaches, schnelles und sicheres Verfahren an die Hand gegeben. Zur Modifizierung des Verhaltens des Checkpointing-Moduls können eigene Strategien mit wenigen Zeilen Programmcode geschrieben und eingebunden werden. Um ein tieferes Verständnis für die Funktionsweisen des Moduls aus Anwendungssicht zu ermöglichen wurden die Implementierungen zweier Workers und einer Strategy besprochen.

4 Evaluierung

In diesem Kapitel werden die erreichten Ergebnisse dieser Arbeit vor allem in Hinblick auf die in Abschnitt 1.2 postulierten Ziele untersucht.

4.1 Einbindung und Benutzung

Die einfache Einbindung in neue und bestehende Anwendungen und eine möglichst intuitive Benutzung sind wesentliche Kriterien für waLBerla-Module. Das Checkpointing-Modul bildet da keine Ausnahme.

Die eingesetzte HDF5-Bibliothek ist eine Standardbibliothek des wissenschaftlichen Rechnens und daher kann davon ausgegangen werden, dass sie auf wissenschaftlich genutzten Clustern bereits installiert ist. Ist dies nicht der Fall, dann ist eine eigene Übersetzung in wenigen Minuten möglich. Sie besitzt keine besonderen Abhängigkeiten, so dass die Installation in wenigen Schritten erledigt ist.

Das Checkpointing-Modul kann in bestehende waLBerla-Anwendungen über die üblichen Aufrufe eingebunden werden. Mittels einer zusätzlichen `init`-Funktion werden benötigte Unterkomponenten hinzugefügt. Eine solche Funktion ist beispielhaft in Listing 4.1 abgebildet: die Funktion fügt zwei Strategies und drei Workers hinzu. Aus den zur Verfügung stehenden Unterkomponenten werden die konkret zu verwendenden Objekte in der Konfigurationsdatei der Anwendung festgelegt (vgl. Abschnitt C.1). Dieses Vorgehen lehnt sich an die Initialisierung anderer waLBerla-Module, wie beispielsweise des Paraview-Moduls, an.

Listing 4.1: Initialisierungsfunktion für das Checkpointing-Modul in einer Anwendung. Es werden zwei Strategies und drei Workers eingebunden, abschließend wird die Konfiguration des Managers aufgerufen.

```
1  void initCheckpointing() {
2     // Strategien hinzufuegen
3     Manager::instance()->reg("Validation", StrategyID(new ValidationStrategy()));
4     Manager::instance()->reg("PeriodicWalltime", StrategyID(new
5        PeriodicWalltimeStrategy()));
6
7     // Worker hinzufuegen
8     Manager::instance()->reg("SimInfo", WorkerID(new SimInfoWorker()));
9     Manager::instance()->reg("Pdf", WorkerID(new PdfWorker()));
10    Manager::instance()->reg("FloatPdf", WorkerID(new FloatPdfWorker()));
11
12    // Konfiguration aufrufen
13    Manager::instance()->configure();
14 }
```

Neue Workers können rasch erstellt werden. Das Checkpointing-Modul bietet fertige Datenstrukturen und Funktionen an, mit denen Felder und Attribute in wenigen Zeilen Programmcode gespeichert und geladen werden können. Auf die Implementierung eines Workers wird intensiv und mit Programmcode-Ausschnitten in Abschnitt 3.6 eingegangen. Oft müssen nur wenige Zeilen Programmcode geschrieben werden, um andere Datenstrukturen und Anwendungen um C/R zu erweitern. Das Erstellen neuer Strategien ist ebenfalls mit einem Minimum an Codierung möglich. In Abschnitt 3.7 wird dies anhand eines konkreten Beispiels im Detail dargestellt.

Nicht nur die Implementierung, sondern auch die erstellten Checkpoints sind höchst portabel. Unabhängig von der verwendeten Hardware-Architektur, Prozessanzahl und waLBerla-Version können Checkpoints geladen und die Berechnungen dadurch ab dem gespeicherten Punkt weiter ausgeführt werden.

4.2 Korrektheit

Alle mit dem Checkpointing-Modul gelieferten Workers arbeiten korrekt, d. h. durch ihren Einsatz beim Restart einer waLBerla-Anwendung werden keine Daten verfälscht und das berechnete Ergebnis verändert sich nicht. Bei der Entwicklung neuer Workers hilft der in Abschnitt 3.5 vorgestellte Validierungsmodus, indem er die Ergebnisse einer Simulation ohne und mit Neustart vergleicht. Zusätzlich können über Hilfswerkzeuge eventuell auftretende Differenzen untersucht werden, um Rückschlüsse auf die Fehlerursache ziehen zu können.

4.3 Performance

In diesem Abschnitt wird die Leistungsfähigkeit der vorgeschlagenen und umgesetzten Architektur für das Checkpointing-Modul belegt. Zu Beginn wird die verwendete Testumgebung vorgestellt und auf verwendete Begriffe und relevante waLBerla-Interna eingegangen. Danach werden die Ergebnisse von Messungen mit verschiedenen Konfigurationen graphisch aufbereitet dargestellt und erläutert. Darauf aufbauend werden allgemeine Schlussfolgerungen gezogen, um auch bei anderen waLBerla-Anwendungen und -Konfigurationen möglichst optimale Performance zu ermöglichen.

4.3.1 Testumgebungen und Grundlagen

Die Geschwindigkeit des Checkpointing-Moduls wurde auf dem Cluster *LiMa* des *RRZE* gemessen. Er verfügt über 500 Rechenknoten, bestehend aus jeweils zwei Prozessoren mit je sechs physikalischen Kernen. Jeder physikalische Kern kann mittels zwei logischer Kerne adressiert werden, so dass dem Benutzer insgesamt 24 logische Kerne pro Knoten zur Verfügung stehen. Die Knoten verfügen über jeweils 24 GB

RAM und sind untereinander per *Infiniband* mit einer Bandbreite von 40 GBit/s verbunden. Das Storage-System verfügt über eine Gesamtkapazität von 100 TB und einer theoretischen, aggregrierten Bandbreite von 3,0 GB/s [Zei10, Reg12]. Als Dateisystem wird *LXFS*, eine Variante von *Lustre* verwendet [NEC09].

Zum Einsatz kommt Revision 993 aus dem *trunk* des waLBerla-Subversion (SVN) sowie Revision 166 des Checkpointing-Moduls, ebenfalls aus dem *trunk*. Übersetzt wurde die Software mit dem vom RRZE bereitgestellten Intel-Compiler in Version 12.1up11. Dabei kamen HDF5 Version 1.8.8 und Intel MPI 4.0.3.008, beide bereits vorinstalliert, zum Einsatz. Die Prozesse wurden, soweit mit der verwendeten Prozessanzahl möglich, ausschließlich auf physikalischen Prozessorkerne verteilt. Die Zuteilung erfolgte blockweise, d. h. es wurden auf jedem Rechenknoten die selbe Anzahl Prozesse mit jeweils benachbarten MPI-Ranks platziert. Die Prozesse wurden auf die ihnen jeweils zugeordneten Kerne gepinnt.

Da in der Regel mehr Checkpoints erstellt als gelesen werden, kommt es stärker auf die beim Schreiben erzeugte Latenz an als auf die Verzögerungen durch den Einlesevorgang. Der Schwerpunkt der folgenden Messungen liegt entsprechend auf der Dauer des Schreibvorgangs. Es wurde die vom Checkpointing-Modul insgesamt benötigte Zeit gemessen und mittels der Größe des erzeugten Checkpoints in Bandbreite umgerechnet. „Gesamt“ bedeutet hierbei, dass sowohl die Initialisierung des Managers, der Strategy und der Workers, als auch die Zeit zum Schreiben der Attribute beim SimInfoWorker mit einfließen. Im Gegensatz dazu heißt „nur Zellen“, dass lediglich die für die Schreibaufrufe des PdfDirectWorkers benötigte Zeitspanne betrachtet wird.

Für die Messungen wurden dieser DirectWorker und nicht der in Abschnitt 3.6 vorgestellte PdfWorker verwendet, da waLBerla derzeit noch ohne Multithreading-Fähigkeiten nicht in der Lage ist, forked Checkpointing umzusetzen. Um die Messergebnisse aber nicht durch das überflüssige Kopieren der Daten zu verfälschen wurde darauf verzichtet; es kam also serielles Checkpointing zum Einsatz.

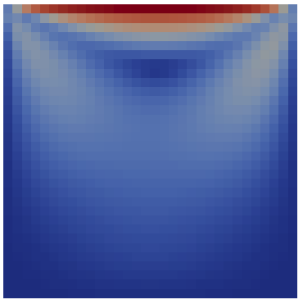

Abb. 4.1: Geschwindigkeitsfeld einer lid-driven cavity, 2D

Zum Test wurde das Standardbeispiel *lid-driven cavity* gewählt: Dabei wird die im Gebiet vorhandene, zu Beginn ruhende Flüssigkeit an einer Grenzfläche um einen konstanten Wert beschleunigt, so dass sich langsam ein Wirbel bildet. Es verändern sich dabei Druck und Geschwindigkeit der Flüssigkeit.

Das Gebiet ist so dimensioniert, dass jeder Prozess die selbe Datenmenge zur Berechnung und damit auch zur Speicherung erhält. Abbildung 4.2 stellt eine solche Konfiguration schematisch dar: Das Rechengebiet teilt sich in insgesamt $4 \cdot 3 \cdot 2 = 24$ jeweils gleich große *Blöcke* auf. Jeder Block besteht aus $3 \cdot 4 \cdot 2 = 24$ *Zellen*, das gesamte Gebiet ist also $(4 \cdot 3) \cdot (3 \cdot 4) \cdot (2 \cdot 2) = 576$ Zellen groß. Die Aufteilung der Daten auf die Prozesse erfolgt dabei über die Blöcke. Im

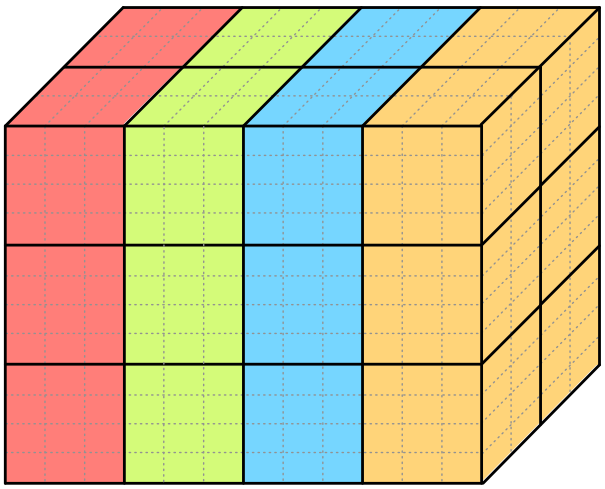

Abb. 4.2: Beispielhafter Aufbau eines quaderförmigen Gebiets in 4x3x2 Blöcke mit jeweils 3x4x2 Zellen; aufgeteilt auf 4 Prozesse.

vorliegenden Beispiel erhält jeder der vier Prozesse jeweils sechs Blöcke und damit $6 \cdot 24 = 144$ Zellen. Mit waLBerla realisierte Simulationen bestehen aus mehreren Tausend Blöcken mit jeweils mehreren Hunderttausend Zellen.

Die nachfolgenden Messungen wurden auf Basis von realitätsnahen Block- und Zellenzahlen durchgeführt. Die verwendeten Werte sind dabei in den Diagrammen festgehalten. Es wurde für jede Zelle lediglich das aus jeweils 19 Fließkomma-Werten bestehende PDF-Feld, das die Verteilungsfunktionen beinhaltet, geschrieben und gelesen: Alle anderen Werte sind entweder davon abgeleitet (wie Geschwindigkeiten und Drücke) oder konstant (Typen der Zellen, bspw. „Flüssigkeitszelle“ oder „Randzelle“). Alle Messungen wurden zu unterschiedlichen Tageszeiten und Wochentagen mehrmals durchgeführt, um negative Einflüsse durch die parallele Nutzung der Systeme durch andere Benutzer zu minimieren. Die dargestellten Ergebnisse sind die daraus resultierenden Durchschnittswerte.

4.3.2 Erstellen eines Checkpoints

In Abbildung 4.3 ist die gesamte, durch das Checkpointing-Modul erzeugte Latenz, umgerechnet in erzielte Bandbreite, abgebildet. Dafür wurde die Größe der erzeugten Datei zugrunde gelegt. Um die Größenordnung der erreichten Performance vergleichen zu können wurde die mit dem zur HDF5-Bibliothek gehörenden Benchmark-Programms *h5perf* erzielte Bandbreite ebenfalls einzeichnet. Die Werte wurden jeweils mit der selben Anzahl Prozesse und Blockgrößen ermittelt, lassen sich aber nicht exakt mit denen des Moduls vergleichen, da die geschriebenen Daten in Bezug auf Inhalt, Gesamtgröße und Anzahl der Attribute nicht identisch sind.

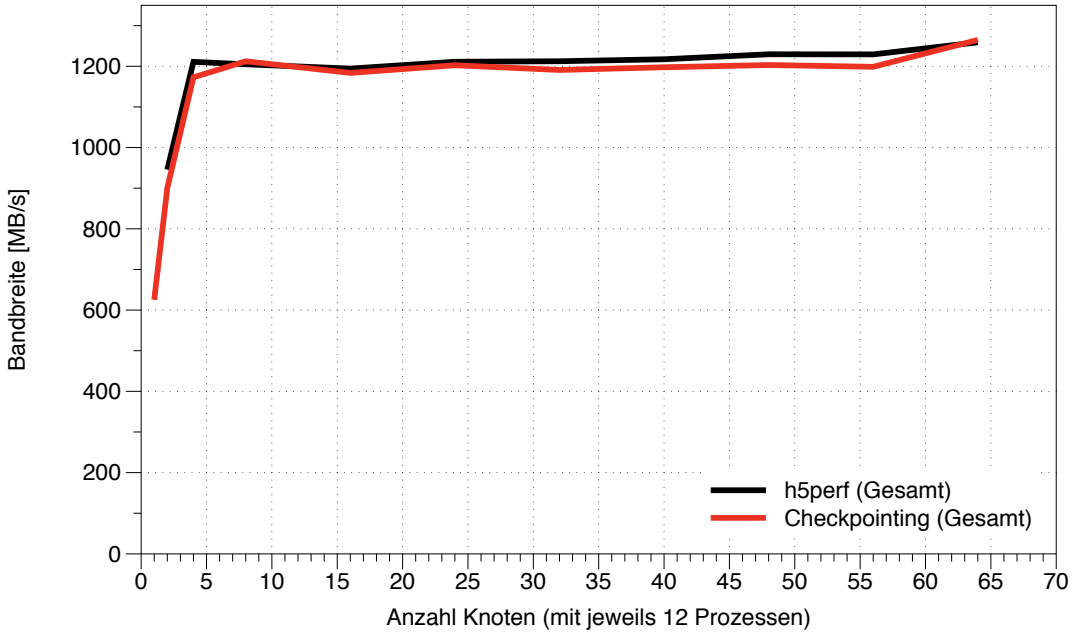

Abb. 4.3: Vergleich der Schreib-Bandbreite von Checkpointing-Modul und HDF5-Benchmark bei variierender Anzahl von Rechenknoten und 12 Prozessen pro Knoten. Blockgröße ca. 145 MB; je ein Block pro Prozess.

Es ist gut erkennbar, dass zu Beginn mit zunehmender Knotenzahl die erzielte Bandbreite stark ansteigt: Mit einem Knoten liegt die erreichte Gesamt-Bandbreite bei ca. 620 MB/s, während sie sich bei der Verwendung von 4 Knoten auf 1175 MB/s fast verdoppelt hat. Der Einsatz weiterer Knoten bringt keine weiteren Verbesserungen, erst bei dem Maximum von 64 Knoten kann nochmals ein Anstieg auf 1250 MB/s beobachtet werden. Möglicherweise ist die (praktisch erreichbare) Bandbreite des I/O-Systems erschöpft oder die verwendeten Rechenknoten liegen alle innerhalb des selben Bereichs des Clusters, so dass an einem Switch ein Engpass auftritt. Eine andere mögliche Erklärung ist, dass eine Limitierung der HDF5-Bibliothek oder der darunter arbeitenden MPI-I/O-Bibliothek bei der Kommunikation und Synchronisierung der Prozesse untereinander auftritt. Möglicherweise bremsen auch Metadaten-Operationen, das Setzen und Entfernen von Sperren oder die Synchronisation der I/O-Knoten untereinander.

In Abschnitt 3.6 wurde als eine mögliche Optimierung ein Worker vorgestellt, der die PDF-Felder unabhängig von der zur Berechnung verwendeten Genauigkeit stets in einfacher Genauigkeit speichert. Abbildung 4.4 zeigt die dadurch erreichte Gesamt-Bandbreite in Vergleich zum Schreiben der Daten mit doppelter Genauigkeit. Die Performance ist etwas geringer, da sich bei dieser Betrachtungsweise die halbierte Datenmenge mit der ebenfalls halbierten Dauer aufheben, aber der Aufwand zur Konvertierung der Daten bestehen bleibt. Die geschriebenen Datenblöcke sind jeweils

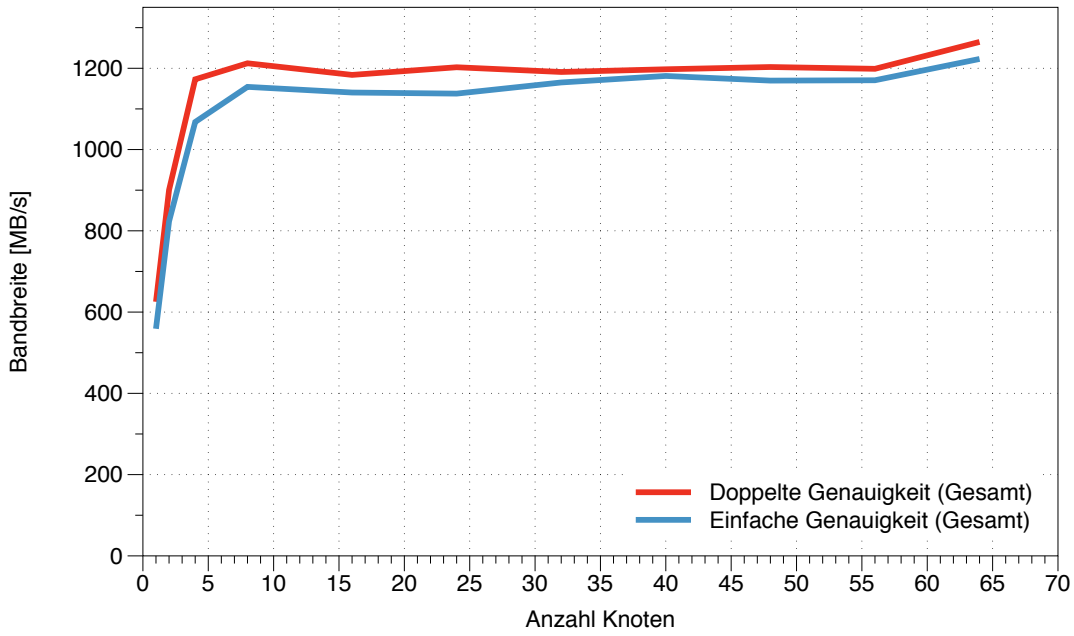

Abb. 4.4: Vergleich der Schreib-Bandbreiten beim Schreiben der Daten in doppelter oder einfacher Genauigkeit. Variierende Anzahl Knoten mit jeweils 12 Prozessen. Blockgröße ca. 145 MB bzw. 72,5 MB; je ein Block pro Prozess.

nur halb so groß, wodurch sich im Verhältnis zusätzlich der durch die (Metadaten-) Operationen erzeugte Overhead vergrößert.

Sinnvoller erscheint es, wie in Abbildung 4.5 den Durchsatz über die Anzahl der geschriebenen Zellen pro Zeiteinheit zu betrachten. Zur besseren Vergleichbarkeit wurden die Werte bei doppelter Genauigkeit verzweifacht als theoretischer Durchsatz eingezeichnet.

Bei der Programmausführung auf Rechen-Clustern ordnet man üblicherweise jedem physikalischem Rechen-Kern einen Prozess zu. Jeder Prozess ist dabei für die selbe Anzahl Blöcke zuständig. Durch diese gleichmäßige Verteilung stellt man sicher, dass ein einzelner Prozess nicht den gesamten Simulationsfortschritt dadurch verlangsamt, dass er in jedem Zeitschritt doppelt so viele Berechnungen durchführen muss und dafür insgesamt die doppelte Zeitdauer benötigt. Die anderen Prozesse müssten sonst auf diesen Prozess warten, bevor die Ergebnisse untereinander ausgetauscht werden können. Die selbe Logik greift auch beim hier gemessenen seriellen Checkpointing. Bei forked Checkpointing sind die beschriebenen Auswirkungen weniger ausgeprägt, da sich die Latenz nur in abgeschwächter Form beim vorangehenden Kopieren der Daten zeigen würde. Das Phänomen würde erst wieder beim Schreiben der Daten auf persistenten Speicher stärker auftreten, da dieser Vorgang langsamer als die Duplikation im Arbeitsspeicher abläuft. Die Latenz ist dann aber nicht mehr relevant, da bereits wieder Berechnungen durchgeführt werden.

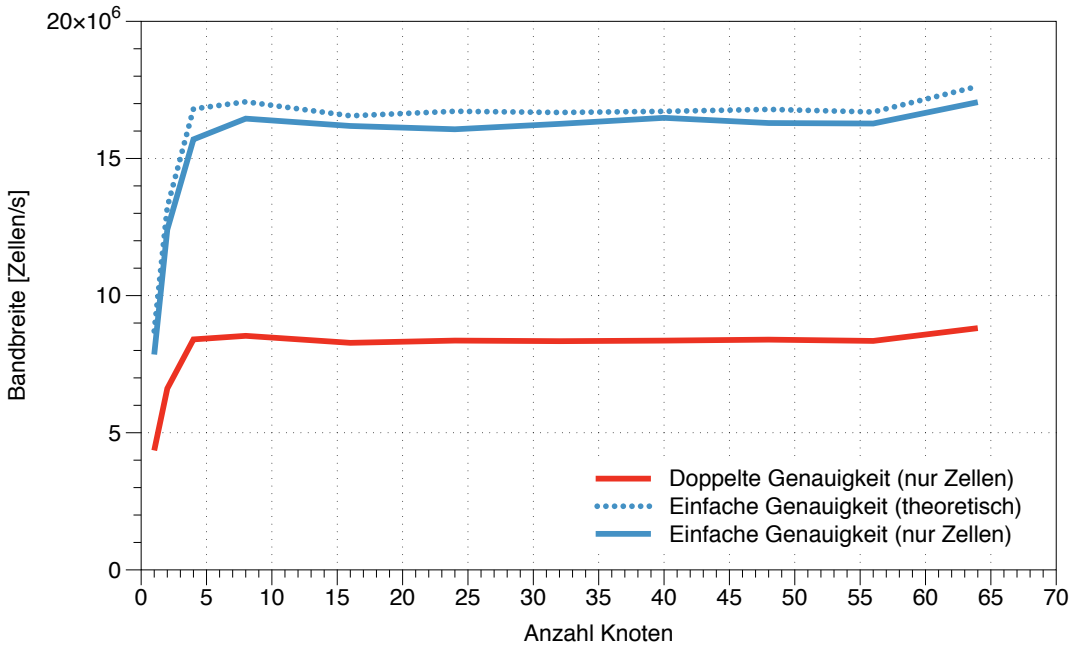

Abb. 4.5: Vergleich des Schreib-Durchsatzes beim Schreiben der Daten in doppelter oder einfacher Genauigkeit. Variierende Anzahl Knoten mit jeweils 12 Prozessen. Blockgröße jeweils 10^9 Zellen; je ein Block pro Prozess.

Ein weiterer Grund, warum eine unterschiedliche Blockanzahl pro Prozess vermieden werden sollte ist, dass dann nicht mehr der kollektive Schreibmodus der HDF5-Bibliothek verwendet werden kann. Unterscheidet sich beim kollektiven Schreiben die Zahl von Schreibaufrufen pro Prozess, dann kann es zu einem Hängen der Bibliothek beim Schließen der Datei kommen [Koz11]. Bis diese Beschränkung durch die HDF5-Entwickler aufgehoben wird prüft daher der Checkpointing-Manager, ob alle Prozesse über die selbe Anzahl Blöcke verfügen und schaltet ggf. in einen anderen Modus. Dieser kann in schlechterer Schreibgeschwindigkeit resultieren. Messungen im Rahmen der Entwicklung und der Evaluierung haben gezeigt, dass diese Einbußen durchschnittlich bei bis zu 15% liegen können, abhängig von der Blockgröße und Prozessanzahl.

Im Folgenden sollen die Auswirkungen der Prozessanzahl und der Blockgröße auf die Schreib-Geschwindigkeit untersucht werden. Es werden auf vier, acht oder 16 Knoten Messungen mit variiender Anzahl von Prozessen pro Knoten und mit unterschiedlichen Blockgrößen durchgeführt. Anhand der Ergebnisse sollen allgemeine Aussagen und Richtlinien für möglichst optimale Aufteilungen und Zuordnungen abgeleitet werden. Für alle nachfolgenden Messungen werden den Prozessen die jeweils gleiche Anzahl Blöcke zugeordnet, so dass stets im kollektiven Modus gearbeitet werden kann. Alle Daten werden in doppelter Genauigkeit geschrieben. In allen Dia-

grammen ist die Prozessanzahl markiert, bis zu der ausschließlich physikalische Kerne verwendet werden.

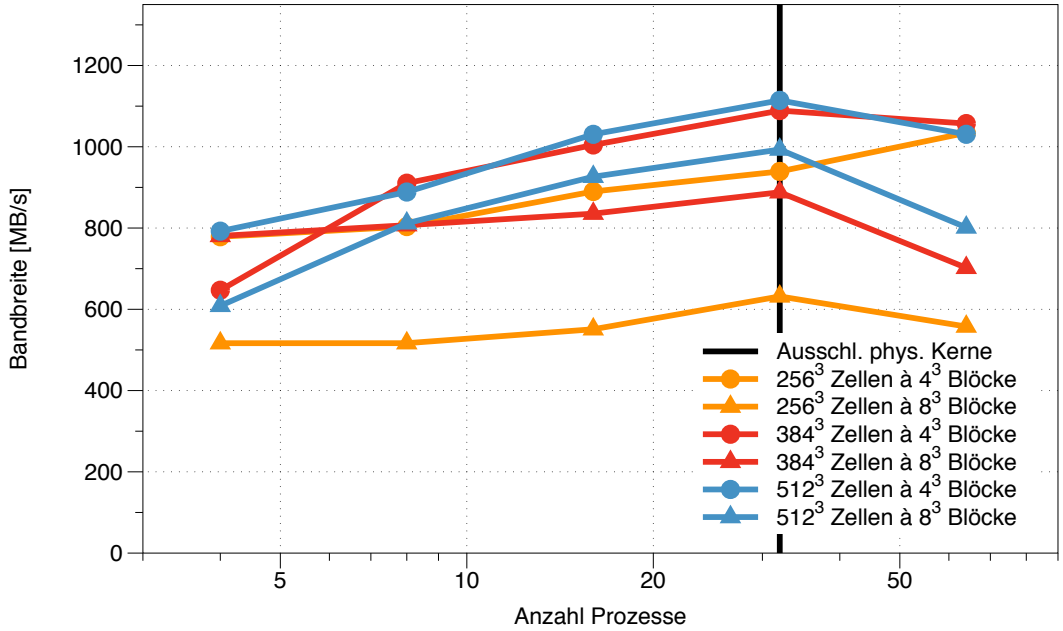

Abb. 4.6: Vergleich Schreib-Bandbreiten bei 4 Knoten und variierender Anzahl von Prozessen und Blockgrößen.

Abbildung 4.6 zeigt eine Gegenüberstellung der Bandbreite bei verschiedenen Block- und Gesamtgrößen und variierender Anzahl Prozessen bei der Verwendung von vier Rechenknoten. Bemerkenswert ist die Auswirkung der unterschiedlichen Blockgrößen bei der Verwendung von 384^3 Zellen: Bei $4^3 = 64$ Blöcken (ca. 128 MB pro Block) liegt die erzielte Bandbreite unterhalb der der anderen Konfigurationen, steigt aber mit zunehmender Prozessanzahl stark an und ist fast deckungsgleich mit der Bandbreite der größeren Blockgröße von 512^3 Zellen (ca. 304 MB pro Block). Eine Erhöhung der Blockgröße führt hier also zu keiner weiteren Erhöhung der Bandbreite. Senkt man hingegen die Blockgröße, dann sinkt auch die erzielte Bandbreite. Bei einer Verachtfachung der Blockanzahl auf 8^3 Blöcke bei 384^3 Zellen (ca. 16 MB pro Block) wird zwar mit geringer Prozessanzahl eine höhere Bandbreite erzielt, doch steigt die resultierende Kurve von allen Graphen am geringsten und sie erleidet auch am Ende den höchsten Einbruch.

Es gibt offensichtlich mehrere Effekte, die sich gegenseitig überlagern können:

1. Bei einer geringen Prozess- bzw. Knotenanzahl wird die verfügbare Bandbreite zum I/O-System nicht ausgelastet. Dadurch kommt es zu Performance-Einbußen.

2. Durch eine Vielzahl von Prozessen erhöht sich der gegenseitige Synchronisations- und Kommunikationsaufwand bspw. beim Setzen und Entfernen von Sperren. Dies senkt die erzielte Bandbreite.

3. Bei einer hohen Prozessanzahl pro Rechenknoten muss die verfügbare Bandbreite zwischen Arbeitsspeicher und CPU-Kernen geteilt werden, so dass hier und im PCIe-Bus zwischen Prozessoren und Netzwerkkarte Engpässe auftreten und die erzielte Bandbreite sinkt. Außerdem kann es verstärkt zu Thrashing und Overhead durch die benötigte Synchronisation der Prozesse kommen.

4. Die I/O-Systeme von Clustern sind für große Mengen kontinuierlich zu schreibender Daten optimiert. Das Schreiben von kleinen Datenmengen kann daher Performance-Einbußen nach sich ziehen (vgl. [Reg12]).

5. Bei geringen Datenmengen (und daraus resultierend kürzeren Schreibzugriffen) fallen Meta-Operationen wie das Initialisieren der beteiligten Komponenten oder das Setzen und Entfernen von Sperren stärker ins Gewicht.

Für alle in Abbildung 4.6 abgebildeten Kurven gilt, dass zu Beginn mit zunehmender Prozessanzahl die erzielte Bandbreite steigt, die erreichbare Bandbreite also noch nicht ausgeschöpft ist (Effekt 1). Fast alle Kurven brechen bei der Verwendung von 128 Prozessen (16 Prozessen pro Knoten) ein, hier machen sich die Effekte 2 und 3 bemerkbar. Im Vergleich besonders stark bricht die Kurve mit 384^3 Zellen und 8^3 Blöcken ein, da hier zusätzlich noch die Effekte 4 und 5 hinzukommen.

Die Kurve zu 256^3 Zellen und 4^3 Blöcken stellt eine Ausnahme dar: Zum einen entspricht die Blockgröße von 16 MB genau der spezifizierten *Stripe*[1]-Größe, zum anderen entspricht sie auch dem *Alignment*[2] des verwendeten Dateisystems. Dadurch erfolgen Zugriffe extrem schnell und die Daten werden kontinuierlich, d. h. ohne zwischenzeitliche Neupositionierung, geschrieben.

Ein vergleichbares Bild zeigt sich bei der Betrachtung von Abbildung 4.7, welche die gemessene Bandbreite bei verschiedenen Blockgrößen und Prozesszahlen unter der Verwendung von acht Rechenknoten darstellt. Bei 640^3 Zellen spielt es keine Rolle, ob die Daten in 4^3 oder 8^3 Blöcke partitioniert werden, die resultierende Bandbreite ist nahezu gleich. In etwa dem selben Bereich liegen die erzielten Bandbreiten für kleinere Blockgrößen bei der Aufteilung in 4^3 Blöcke. Die Konfiguration 512^3 Zellen à 4^3 Blöcke (Blockgröße 304 MB = 19·16 MB) profitiert wieder vom schnelleren Zugriff durch exaktes Alignment. In geringerem Maße trifft dies auch bei 384^3 Zellen

[1] Ein *Stripe* ist die Einheit, in der Daten geschrieben werden; in etwa vergleichbar einer Blockgröße bei herkömmlichen Dateisystemen

[2] Als *Alignment* bezeichnet man die Grenzen, an denen Datenelemente in einem Speicher ausgerichtet werden. Dieses ist oft durch technische Gegebenheiten definiert. In der Regel erfolgen Zugriffe mit passendem Alignment deutlich schneller als Zugriffe ohne passendes Alignment.

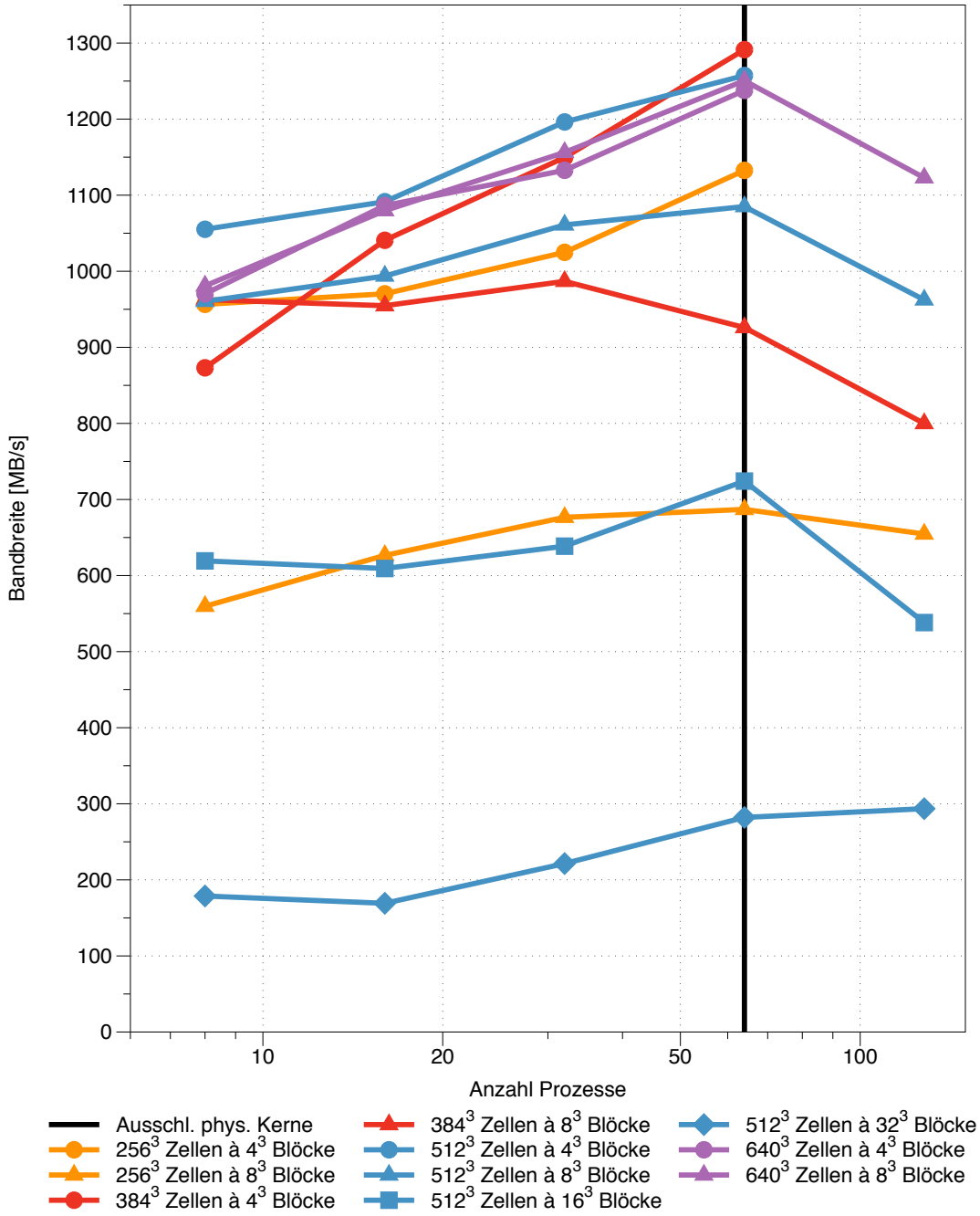

Abb. 4.7: Vergleich Schreib-Bandbreiten bei 8 Knoten und variierender Anzahl von Prozessen und Blockgrößen.

à 4^3 Blöcken (128,25 MB pro Block) zu, da das Alignment hier nicht ganz so exakt getroffen wird. Zusätzlich tritt bei geringer Prozesszahl Effekt 1 auf.

Die in Abbildung 4.8 abgebildeten Ergebnisse beim Einsatz von 16 Rechenknoten bestätigen die gewonnen Erkenntnisse: Je geringer die Blockgröße, desto stärker bricht die erzielte Bandbreite bei einer hohen Zahl von Prozessen pro Knoten ein. Die Gesamtperformance liegt beim Einsatz 16 Knoten etwas höher als bei acht Knoten und deutlich höher als bei vier Knoten, hier kommt ganz deutlich Effekt 1 zum Tragen.

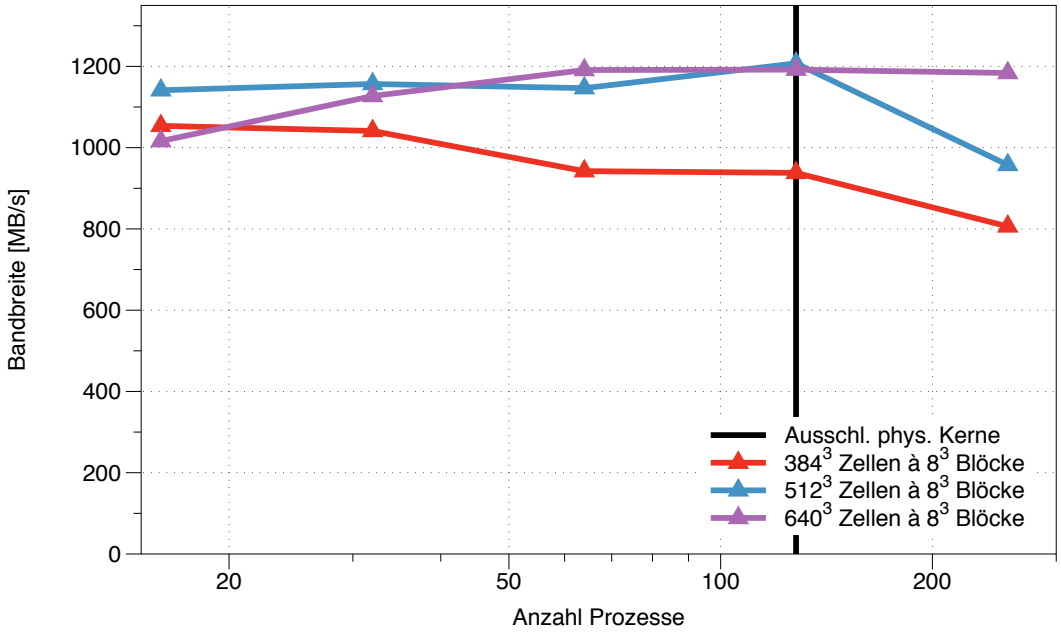

Abb. 4.8: Vergleich Schreib-Bandbreiten bei 16 Knoten und variierender Anzahl von Prozessen und Blockgrößen.

Im Laufe der Messungen hat sich gezeigt, dass für die Rechenknoten von LiMa eine 1:1-Zuordnung von Blöcken zu Prozessen optimal ist. Pro Rechenknoten sollten maximal 12 Prozesse zum Einsatz kommen. Dies entspricht der Zahl der physikalisch vorhandenen Kerne und ist daher ohnehin das für die Berechnungen von Simulationen postulierte Optimum. Die verwendeten Striping- und Alignment-Werte können und sollten für jede Cluster-Konfiguration individuell angepasst werden, um eine optimale Bandbreite zu erzielen.

4.3.3 Laden eines Checkpoints

Auch wenn der Einlesevorgang weniger oft wiederholt wird als das Schreiben eines Checkpoints, so ist die dabei erreichte Geschwindigkeit dennoch eine wichtige Größe.

Abbildung 4.9 zeigt die erzielte Bandbreite bei der Verwendung von bis zu 64 Knoten mit jeweils 12 Prozessen. Es zeigt sich ein ähnliches Bild wie beim Schreiben: Mit geringer Knotenzahl kann die erreichbare Bandbreite bei weitem nicht ausgeschöpft werden, durch zusätzliche Knoten steigt die erzielte Geschwindigkeit aber stark an. Das Maximum wird beim Einsatz von acht Knoten erreicht, ab dann stagniert die Bandbreite bei ca. 1,5 GB/s. Dieser Wert liegt erwartungsgemäß höher als der beim Schreiben gemessene Wert von etwa 1,2 GB/s, da das Lesen von Daten weniger Overhead erzeugt. Es kann dabei bspw. auf das Setzen von Sperren verzichtet werden. Die Daten werden auf verteilte Storage-Systeme geschrieben. Bei Lesevorgängen ist hier im Gegensatz zu Schreibvorgängen keine Synchronisierung der I/O-Knoten nötig. Auf einen Vergleich mit dem HDF5-Benchmarkprogramm muss hier verzichtet werden. Bei einem Messlauf werden zuerst Daten geschrieben und sofort wieder eingelesen, dabei wird die jeweils benötigte Zeitdauer gemessen. Die zu lesenden Daten sind dabei aber noch im Cache vorhanden, was sich in unrealistisch hohen Bandbreiten weit über dem theoretischen Maximum der erreichbaren Bandbreite von 3,0 GB/s niederschlägt.

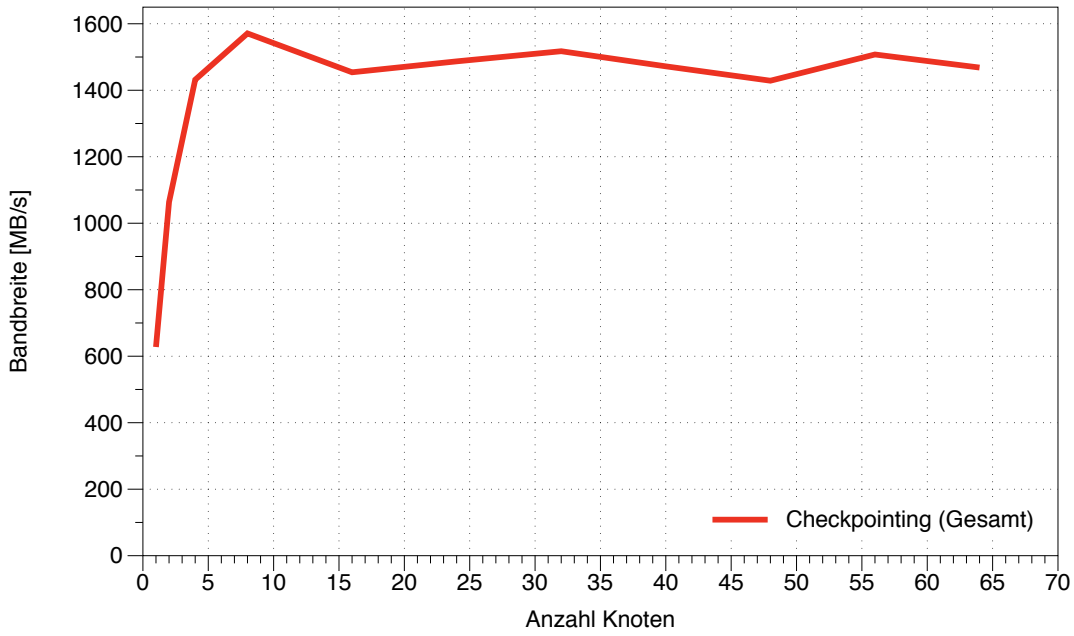

Abb. 4.9: Vergleich der Lese-Bandbreiten bei Daten in doppelter Genauigkeit. Variierende Anzahl Knoten mit jeweils 12 Prozessen. Blockgröße ca. 145 MB; je ein Block pro Prozess.

4.3.4 Zusammenfassung

Die umgesetzte Architektur und die bereitgestellten Workers und Hilfsfunktionen sind schnell genug, um das Checkpointing in Simulationen unter realistischen Bedingungen einsetzen zu können. Aus den gemessenen Daten ergeben sich drei allgemeine Schlussfolgerungen:

1. Die Daten sollten auf wenige, sehr große Blöcke aufgeteilt werden.
2. Es sollten möglichst viele Prozesse parallel schreiben.
3. Es gibt eine optimale Anzahl von Prozessen pro Rechenknoten.

4.4 Zusammenfassung

Die Evaluierung der vorgeschlagenen und umgesetzten Lösung zeigt, dass die in Abschnitt 1.2 gesteckten Ziele erfüllt werden konnten und waLBerla um eine benutzer- und entwicklerfreundliche, erweiterbare, portable und performante synchrone Checkpointing-Funktionalität für erweitert wurde. Da es sich bei dem genutzten Dateiformat um ein Standardformat handelt, ist es sogar möglich, die erzeugten Checkpoints in anderen Programmen weiterzuverwenden, bspw. um die erzeugten Daten zu visualisieren.

5 Zusammenfassung

Dieses Kapitel gibt eine Übersicht über die Ergebnisse, die im Rahmen dieser Arbeit erzielt wurden. Abschließend wird auf mögliche weiterführende Themen und Aufgabenstellungen eingegangen, die auf den Ergebnissen der Arbeit aufbauen können.

5.1 Erreichter Beitrag

Das waLBerla-Framework wurde um ein performantes und zuverlässiges Checkpointing-Modul erweitert, mit dem laufende Simulationen gesichert werden können, um im Fehlerfall ab diesem Punkt die Berechnung fortzuführen. Das Modul und die erstellten Checkpoint-Dateien sind so portabel wie waLBerla selbst, so dass die auf einem Cluster begonnene Simulation auf einem anderen Cluster weitergeführt werden kann. Durch die Nutzung eines wissenschaftlichen Standarddateiformats können die erzeugten Dateien auch in anderen Programmen genutzt werden, um so beispielsweise bessere Visualisierungsmöglichkeiten zu schaffen.

Der interne Aufbau des Moduls ist dabei so einfach gehalten, dass problemlos neue Anwendungen mit eigenen Datenstrukturen gesichert und geladen werden können. Bei der Erweiterung des zu sichernden Datenumfangs unterstützt das Modul den Entwickler mit eingebauten Validierungsstrategien, die sicher stellen, dass sich durch zwischenzeitliche Restarts das Simulationsergebnis nicht ändert. Spezielle Strategien, wann ein Checkpoint erstellt werden soll, lassen sich ebenfalls leicht implementieren.

In der vorliegenden Arbeit wurden Grundlagen des Checkpointings erklärt, von der Einordnung in das Gebiet der Fehlertoleranz über verschiedene Funktionsweisen hin zu Performance-Auswirkungen in der Praxis. Es wurden Optimierungsansätze gezeigt, darunter die Bestimmung eines optimalen Checkpoint-Intervalls. Danach wurden Grundkonzepte von waLBerla vorgestellt und die Architektur und Umsetzung des Checkpointing-Moduls beschrieben. Abschließend wurde die erarbeitete Lösung unter anderem in Hinblick auf Korrektheit und Performance evaluiert.

5.2 Weiterführende Arbeiten

Die im Rahmen dieser Arbeit entstandenen Komponenten, die vorrangig die Datenstrukturen einer einfachen Lattice-Boltzmann-Simulation sichern und laden, sollten auf weitere Anwendungen und Datenstrukturen adaptiert und erweitert werden. Diese Aufgabe obliegt den Entwicklern der jeweils auf waLBerla aufbauenden An-

wendungen, da nur diese entscheiden können, welche Daten relevant sind und damit gesichert und geladen werden müssen.

Es wäre denkbar, ein Hardware-Monitoring-System in waLBerla zu integrieren, welches die „Systemgesundheit“ prüfen soll. Verändern sich bestimmte Parameter, so dass von einem drohenden Ausfall ausgegangen werden kann, dann könnte prophylaktisch ein Checkpoint erstellt und ggf. die Simulation beendet werden. Auch könnte die anhand der aktuellen Auslastung des Storage-Systems eine Abschätzung getroffen werden, wie lang die Erstellung eines Checkpoints dauern würde und entsprechend einer vorgegebenen insgesamten Ziel-Laufzeit der Simulation der Checkpoint-Zeitpunkt adaptiv angepasst werden.

Weiterhin könnte bei einem Abbruch die Simulation automatisch ab dem letzten Checkpoint neu gestartet werden. Aktuell obliegt es dem Nutzer, den Abbruch zu erkennen und die Anwendung erneut zu starten. Hierfür wäre ein Wrapper notwendig, der sich um die Programmausführung kümmert und am Abbruch erkennt, ob ein (wiederholt auftretender) Programmfehler oder tatsächlich ein Hardware-Fehler vorlag.

Denkbar wäre auch, einige der in Unterabschnitt 2.1.5 vorgestellten Optimierungen, wie beispielsweise inkrementelles Checkpointing, umzusetzen. Es müssten hierfür geeignete Workers implementiert werden, die entstandene Änderungen entdecken und nur diese Differenzen speichern.

Abschließend stellt sich die Frage, ob die bei einem Checkpoint erstellten Dateien nicht auch anderweitig genutzt werden könnten. Da ein wissenschaftliches Standardformat zum Speichern der Daten verwendet wurde, wäre es denkbar, dass bestehende Viewer und andere Anwendungen in das Postprocessing der Simulationsergebnisse mit einbezogen werden könnten.

Anhang

A Verwendete Beispielprogramme

A.1 boost::serialization Beispielprogramm

Listing A.1: Beispielprogramm zur (De-)Serialisierung einer Klasse mit abstrakter Basisklasse via boost::serialization.

```
1  #include <fstream>
2  #include <boost/archive/text_oarchive.hpp>
3  #include <boost/archive/text_iarchive.hpp>
4  #include <boost/serialization/base_object.hpp>
5  #include <boost/serialization/export.hpp>
6  #include <boost/serialization/serialization.hpp>
7  #include <boost/serialization/vector.hpp>
8
9  // Abstrakte Basisklasse
10 class AbstractData {
11 private:
12     std::string id_;
13 protected:
14     AbstractData() : id_("unknown") { }
15 public:
16    AbstractData(std::string id) : id_(id) { }
17    virtual const int getTotalSize() const = 0;
18 private:
19    friend class boost::serialization::access; // Zugriff auf private Member-Var.
20    template<class Archive> void serialize(Archive& ar, const unsigned int version){
21       ar & id_; // Member-Variable (de-)serialisieren
22    }
23 };
24 BOOST_CLASS_EXPORT(AbstractData)
25
26 // abgeleitete Klasse
27 class Data : public AbstractData {
28 private:
29    std::vector<double> data_;
30    Data() : AbstractData() { }
31 public:
32    Data(std::string id, int size) : AbstractData(id) { data_.resize(size); }
33    void set(int idx, double value) { data_[idx] = value; }
34    double get(int idx) { return data_[idx]; }
35    virtual const int getTotalSize() const { return data_.size(); }
36 private:
37    friend class boost::serialization::access;
38    template<class Archive> void serialize(Archive& ar, const unsigned int version){
39       ar & boost::serialization::base_object<AbstractData>(*this);
40       ar & data_;
41    }
42 };
43 BOOST_CLASS_EXPORT(Data)
44 BOOST_CLASS_TRACKING(Data, boost::serialization::track_never)
45
46 int main() {
```

```
47      Data data("myData", 16);
48      // Ggf. Werte setzen...
49       std::ofstream ofs("archive.txt"); // Ausgabedatei und -strom oeffnen
50      boost::archive::text_oarchive oa(ofs); // Als lesbaren Text serialisieren
51      oa << data; // Serialisierung eines Objekts
52       ofs.close();
53
54       Data loaded("loaded", 1); // Zielpunkt fuer Deserialisierung
55       std::ifstream ifs("archive.txt"); // Eingabedatei und -strom oeffnen
56       boost::archive::text_iarchive ia(ifs); // Aus lesbarem Text oeffnen
57       ia >> loaded; // Deserialisierung in dafuer vorgesehenes Objekt
58       ifs.close();
59
60       assert(data.getTotalSize() == loaded.getTotalSize());
61       // Ggf. Werte pruefen...
62       return 0;
63  }
```

A.2 HDF5 Beispielprogramm

Listing A.2: Beispielprogramm zur Definition einer HDF5-Datei entsprechend Abbildung 3.1.

```
1   #include <hdf5.h>
2
3   int main(int argc, char* argv[]) {
4       // Allgemein
5       hid_t h5_file_id;
6       hsize_t h5_dims[] = {255, 255, 255}; // Dimension: 255 x 255 x 255
7       hid_t h5_string_t; // Datentyp fuer String der Laenge 50
8       hid_t h5_filespace_id_daten;
9       hid_t h5_filespace_id_attribute;
10
11      // Wurzelelement
12      hid_t h5_attribute_id_ersteller;
13
14      // Gruppe Messdaten
15      hid_t h5_group_id_messdaten;
16      hid_t h5_dataset_id_zeitschritt_1, h5_dataset_id_zeitschritt_2;
17      hid_t h5_attribute_id_laborant;
18
19      // Gruppe Simulation
20      hid_t h5_group_id_simulation;
21      hid_t h5_dataset_id_lauf_1;
22      hid_t h5_attribute_id_revision;
23
24      // ########################################
25      // Datei erstellen
26      h5_file_id = H5Fcreate("example.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
27
28      // Filespace anlegen
29      h5_filespace_id_daten = H5Screate_simple(3, h5_dims, h5_dims);
30      h5_filespace_id_attribute = H5Screate(H5S_SCALAR);
31
32      // Datentyp fuer String mit 50 Zeichen und Null-Terminierung erstellen
33      h5_string_t = H5Tcopy(H5T_C_S1);
```

```
34      H5Tset_size(h5_string_t, 50);
35      H5Tset_strpad(h5_string_t, H5T_STR_NULLTERM);
36
37      // ##########################################
38      // Gruppe Messdaten
39      h5_group_id_messdaten = H5Gcreate(h5_file_id, "Messdaten", H5P_DEFAULT,
40                                        H5P_DEFAULT, H5P_DEFAULT);
41
42      // Datasets anlegen
43      h5_dataset_id_zeitschritt_1 = H5Dcreate(h5_group_id_messdaten, "Zeitschritt_1",
44                                              H5T_NATIVE_DOUBLE,
45                                              h5_filespace_id_daten, H5P_DEFAULT,
46                                              H5P_DEFAULT, H5P_DEFAULT);
47
48      h5_dataset_id_zeitschritt_2 = H5Dcreate(h5_group_id_messdaten, "Zeitschritt_2",
49                                              H5T_NATIVE_DOUBLE,
50                                              h5_filespace_id_daten, H5P_DEFAULT,
51                                              H5P_DEFAULT, H5P_DEFAULT);
52
53      // Attribut anlegen
54      h5_attribute_id_laborant = H5Acreate(h5_group_id_messdaten, "Laborant",
55                                           h5_string_t, h5_filespace_id_attribute,
56                                           H5P_DEFAULT, H5P_DEFAULT);
57
58      // Gruppe schliessen
59      H5Gclose(h5_group_id_messdaten);
60      // ##########################################
61
62
63      // ##########################################
64      // Gruppe Simulation
65      h5_group_id_simulation = H5Gcreate(h5_file_id, "Simulation", H5P_DEFAULT,
66                                         H5P_DEFAULT, H5P_DEFAULT);
67
68      // Dataset anlegen
69      h5_dataset_id_lauf_1 = H5Dcreate(h5_group_id_simulation, "Lauf_1",
70                                       H5T_NATIVE_DOUBLE,
71                                       h5_filespace_id_daten, H5P_DEFAULT,
72                                       H5P_DEFAULT, H5P_DEFAULT);
73
74      // Attribut anlegen
75      h5_attribute_id_revision = H5Acreate(h5_group_id_simulation, "Revision",
76                                           H5T_NATIVE_UINT,h5_filespace_id_attribute,
77                                           H5P_DEFAULT, H5P_DEFAULT);
78
79      // Gruppe schliessen
80      H5Gclose(h5_group_id_simulation);
81      // ##########################################
82
83      // ##########################################
84      // Wurzelelement: Attribut anlegen
85      h5_attribute_id_ersteller = H5Acreate(h5_file_id, "Ersteller", h5_string_t,
86                                            h5_filespace_id_attribute, H5P_DEFAULT,
87                                            H5P_DEFAULT);
88
89      // ##########################################
90
91      H5Sclose(h5_filespace_id_daten);
92      H5Fclose(h5_file_id);
93
94      return 0;
95  }
```

Listing A.3: Mit dem Programm aus Listing A.2 erzeugte Dateistruktur. Ausgabe via *h5dump*.

```
HDF5 "example.h5" {
GROUP "/" {
   ATTRIBUTE "Ersteller" {
      DATATYPE  H5T_STRING {
            STRSIZE 50;
            STRPAD H5T_STR_NULLTERM;
            CSET H5T_CSET_ASCII;
            CTYPE H5T_C_S1;
         }
      DATASPACE  SCALAR
   }
   GROUP "Messdaten" {
      ATTRIBUTE "Laborant" {
         DATATYPE  H5T_STRING {
               STRSIZE 50;
               STRPAD H5T_STR_NULLTERM;
               CSET H5T_CSET_ASCII;
               CTYPE H5T_C_S1;
            }
         DATASPACE  SCALAR
      }
      DATASET "Zeitschritt_1" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 255, 255, 255 ) / ( 255, 255, 255 ) }
      }
      DATASET "Zeitschritt_2" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 255, 255, 255 ) / ( 255, 255, 255 ) }
      }
   }
   GROUP "Simulation" {
      ATTRIBUTE "Revision" {
         DATATYPE  H5T_STD_U32LE
         DATASPACE  SCALAR
      }
      DATASET "Lauf_1" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 255, 255, 255 ) / ( 255, 255, 255 ) }
      }
   }
}
}
```

A.3 netCDF Beispielprogramm

Listing A.4: Beispielprogramm zur Definition einer NetCDF-Datei entsprechend Abbildung 3.1.

```
1  #include <netcdf.h>
2
3  int main(int argc, char* argv[]) {
4      // Allgemein
5      int nc_file_id;
6      unsigned int nc_dims[] = {255, 255, 255}; // Dimension: 255 x 255 x 255
7      int nc_dims_id[3];
8
9      // Gruppe Messdaten
10     int nc_group_id_messdaten;
11     int nc_dataset_id_zeitschritt_1, nc_dataset_id_zeitschritt_2;
12
13     // Gruppe Simulation
14     int nc_group_id_simulation;
15     int nc_dataset_id_lauf_1;
16
17     // ##########################################
18     // Datei erstellen
19     nc_create("example.nc", NC_NETCDF4|NC_CLOBBER, &nc_file_id);
20     nc_redef(nc_file_id); // Define-Modus
21
22     // Dimensionen anlegen
23     nc_def_dim(nc_file_id, "x", nc_dims[0], &(nc_dims_id[0]));
24     nc_def_dim(nc_file_id, "y", nc_dims[1], &(nc_dims_id[1]));
25     nc_def_dim(nc_file_id, "z", nc_dims[2], &(nc_dims_id[2]));
26
27     // ##########################################
28     // Gruppe Messdaten
29     nc_def_grp(nc_file_id, "Messdaten", &nc_group_id_messdaten);
30
31     // Datasets anlegen
32     nc_def_var(nc_group_id_messdaten, "Zeitschritt_1", NC_DOUBLE, 3, nc_dims_id,
33                &nc_dataset_id_zeitschritt_1);
34     nc_def_var(nc_group_id_messdaten, "Zeitschritt_2", NC_DOUBLE, 3, nc_dims_id,
35                &nc_dataset_id_zeitschritt_2);
36
37     // Attribut anlegen
38     nc_put_att_text(nc_group_id_messdaten, NC_GLOBAL, "Laborant", 0, NULL);
39     // ##########################################
40
41
42     // ##########################################
43     // Gruppe Simulation
44     nc_def_grp(nc_file_id, "Simulation", &nc_group_id_simulation);
45
46     // Dataset anlegen
47     nc_def_var(nc_group_id_simulation, "Lauf_1", NC_DOUBLE, 3, nc_dims_id,
48                &nc_dataset_id_lauf_1);
49
50     // Attribut anlegen
51     nc_put_att_uint(nc_group_id_simulation, NC_GLOBAL, "Revision", NC_UINT, 0,
52                     NULL);
53     // ##########################################
54
55     // ##########################################
```

```
56      // Wurzelelement: Attribut anlegen
57      nc_put_att_text(nc_file_id, NC_GLOBAL, "Ersteller", 0, NULL);
58      // ########################################
59
60      nc_enddef(nc_file_id); // Define-Modus verlassen
61      nc_close(nc_file_id); // Datei schliessen
62
63      return 0;
64  }
```

Listing A.5: Mit dem Programm aus Abschnitt A.3 erzeugte Dateistruktur. Ausgabe via *ncdump*.

```
1   netcdf example {
2   dimensions:
3      x = 255 ;
4      y = 255 ;
5      z = 255 ;
6
7   // global attributes:
8         :Ersteller = "" ;
9
10  group: Messdaten {
11    variables:
12     double Zeitschritt_1(x, y, z) ;
13     double Zeitschritt_2(x, y, z) ;
14
15    // group attributes:
16        :Laborant = "" ;
17    } // group Messdaten
18
19  group: Simulation {
20    variables:
21     double Lauf_1(x, y, z) ;
22
23    // group attributes:
24        :Revision = "" ;
25    } // group Simulation
26  }
```

Listing A.6: Mit dem Programm aus Abschnitt A.3 erzeugte Dateistruktur. Ausgabe via *h5dump*.

```
HDF5 "example.nc" {
GROUP "/" {
   ATTRIBUTE "Ersteller" {
      DATATYPE  H5T_STRING {
         STRSIZE 1;
         STRPAD H5T_STR_NULLTERM;
         CSET H5T_CSET_ASCII;
         CTYPE H5T_C_S1;
      }
      DATASPACE  NULL
   }
   GROUP "Messdaten" {
      ATTRIBUTE "Laborant" {
         DATATYPE  H5T_STRING {
            STRSIZE 1;
            STRPAD H5T_STR_NULLTERM;
            CSET H5T_CSET_ASCII;
            CTYPE H5T_C_S1;
         }
         DATASPACE  NULL
      }
      DATASET "Zeitschritt_1" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 255, 255, 255 ) / ( 255, 255, 255 ) }
         ATTRIBUTE "DIMENSION_LIST" {
            DATATYPE  H5T_VLEN { H5T_REFERENCE { H5T_STD_REF_OBJECT }}
            DATASPACE  SIMPLE { ( 3 ) / ( 3 ) }
         }
      }
      DATASET "Zeitschritt_2" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 255, 255, 255 ) / ( 255, 255, 255 ) }
         ATTRIBUTE "DIMENSION_LIST" {
            DATATYPE  H5T_VLEN { H5T_REFERENCE { H5T_STD_REF_OBJECT }}
            DATASPACE  SIMPLE { ( 3 ) / ( 3 ) }
         }
      }
   }
   GROUP "Simulation" {
      ATTRIBUTE "Revision" {
         DATATYPE  H5T_STD_U32LE
         DATASPACE  NULL
      }
      DATASET "Lauf_1" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 255, 255, 255 ) / ( 255, 255, 255 ) }
         ATTRIBUTE "DIMENSION_LIST" {
            DATATYPE  H5T_VLEN { H5T_REFERENCE { H5T_STD_REF_OBJECT }}
            DATASPACE  SIMPLE { ( 3 ) / ( 3 ) }
         }
      }
   }
   DATASET "x" {
      DATATYPE  H5T_IEEE_F32BE
      DATASPACE  SIMPLE { ( 255 ) / ( 255 ) }
      ATTRIBUTE "CLASS" {
         DATATYPE  H5T_STRING {
            STRSIZE 16;
            STRPAD H5T_STR_NULLTERM;
            CSET H5T_CSET_ASCII;
```

```
61              CTYPE H5T_C_S1;
62           }
63           DATASPACE  SCALAR
64        }
65        ATTRIBUTE "NAME" {
66           DATATYPE  H5T_STRING {
67              STRSIZE 64;
68              STRPAD H5T_STR_NULLTERM;
69              CSET H5T_CSET_ASCII;
70              CTYPE H5T_C_S1;
71           }
72           DATASPACE  SCALAR
73        }
74        ATTRIBUTE "REFERENCE_LIST" {
75           DATATYPE  H5T_COMPOUND {
76              H5T_REFERENCE { H5T_STD_REF_OBJECT } "dataset";
77              H5T_STD_I32LE "dimension";
78           }
79           DATASPACE  SIMPLE { ( 3 ) / ( 3 ) }
80        }
81     }
82     DATASET "y" {
83        DATATYPE  H5T_IEEE_F32BE
84        DATASPACE  SIMPLE { ( 255 ) / ( 255 ) }
85        ATTRIBUTE "CLASS" {
86           DATATYPE  H5T_STRING {
87              STRSIZE 16;
88              STRPAD H5T_STR_NULLTERM;
89              CSET H5T_CSET_ASCII;
90              CTYPE H5T_C_S1;
91           }
92           DATASPACE  SCALAR
93        }
94        ATTRIBUTE "NAME" {
95           DATATYPE  H5T_STRING {
96              STRSIZE 64;
97              STRPAD H5T_STR_NULLTERM;
98              CSET H5T_CSET_ASCII;
99              CTYPE H5T_C_S1;
100          }
101          DATASPACE  SCALAR
102       }
103       ATTRIBUTE "REFERENCE_LIST" {
104          DATATYPE  H5T_COMPOUND {
105             H5T_REFERENCE { H5T_STD_REF_OBJECT } "dataset";
106             H5T_STD_I32LE "dimension";
107          }
108          DATASPACE  SIMPLE { ( 3 ) / ( 3 ) }
109       }
110    }
111    DATASET "z" {
112       DATATYPE  H5T_IEEE_F32BE
113       DATASPACE  SIMPLE { ( 255 ) / ( 255 ) }
114       ATTRIBUTE "CLASS" {
115          DATATYPE  H5T_STRING {
116             STRSIZE 16;
117             STRPAD H5T_STR_NULLTERM;
118             CSET H5T_CSET_ASCII;
119             CTYPE H5T_C_S1;
120          }
121          DATASPACE  SCALAR
122       }
123       ATTRIBUTE "NAME" {
```

```
124           DATATYPE  H5T_STRING {
125              STRSIZE 64;
126              STRPAD H5T_STR_NULLTERM;
127              CSET H5T_CSET_ASCII;
128              CTYPE H5T_C_S1;
129           }
130           DATASPACE  SCALAR
131        }
132        ATTRIBUTE "REFERENCE_LIST" {
133           DATATYPE  H5T_COMPOUND {
134              H5T_REFERENCE { H5T_STD_REF_OBJECT } "dataset";
135              H5T_STD_I32LE "dimension";
136           }
137           DATASPACE  SIMPLE { ( 3 ) / ( 3 ) }
138        }
139     }
140  }
141  }
```

A.4 Skript für Validierung

Listing A.7: Skript zur Validierung mittels ValidationStrategy.

```
1  #!/bin/bash
2
3  APP="students"
4  CONFIG="input.dat"
5  MPI="mpirun -np 2"
6  H5DIFF=`which h5diff`
7
8  # delete old checkpoints
9  rm validation_complete_nonrestarted.h5
10 rm validation_complete_restarted.h5
11 rm validation_half.h5
12
13 # first run, hide output but show errors
14 ${MPI} ${APP} ${CONFIG} > /dev/null
15
16 # second run, hide output but show errors
17 ${MPI} ${APP} ${CONFIG} > /dev/null
18
19 # compare files - will output differences
20 ${H5DIFF} validation_complete_nonrestarted.h5 validation_complete_restarted.h5
21 if [[ $? -eq 0 ]]
22 then
23    echo "Files are identical!"
24 fi
```

B Softwarelizenzen

B.1 Boost License

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

B.2 HDF5 License

Redistribution and use in source and binary forms, with or without modification, are permitted for any purpose (including commercial purposes) provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or materials provided with the distribution.

3. In addition, redistributions of modified forms of the source or binary code must carry prominent notices stating that the original code was changed and the date of the change.

4. All publications or advertising materials mentioning features or use of this software are asked, but not required, to acknowledge that it was developed by The HDF Group and by the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign and credit the contributors.

5. Neither the name of The HDF Group, the name of the University, nor the name of any Contributor may be used to endorse or promote products derived from this software without specific prior written permission from The HDF Group, the University, or the Contributor, respectively.

DISCLAIMER: THIS SOFTWARE IS PROVIDED BY THE HDF GROUP AND THE CONTRIBUTORS “AS IS” WITH NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. In no event shall The HDF Group or the Contributors be liable for any damages suffered by the users arising out of the use of this software, even if advised of the possibility of such damage.

B.3 netCDF License

C Konfigurationsparameter

Im folgenden werden verfügbare Parameter des Checkpointing-Moduls für die Konfigurationsdatei der Anwendungen aufgeführt. Unterstrichene Parameternamen kennzeichnen Pflicht-Angaben; unterstrichende Werte werden als Standardwert angenommen. Die Parameter werden unterschieden nach Einzelangabe (E) oder Block (B). Eine Unterscheidung nach Groß-/Kleinschreibung findet nicht statt.

C.1 CheckpointingManager

Die Parameter werden aus dem ersten Block mit dem Namen `checkpointing` der Konfigurationsdatei ausgelesen.

Parameter	Typ	Wert	Erklärung
`alignment`	E		Beim Schreiben zu verwendendes Datei-Alignment.
`buffersize`	E		Beim Schreiben zu verwendende MPI-I/O-Puffer-Größe.
`fileaccess`	E	`independent`	Prozesse sind bei I/O unabh. voneinander[1]
		`collective`	kollektive MPI-IO-Aufrufe[1]
		`lustre`	Wie `collective`, aber optimiert für das Dateisystem *Lustre*
		`parallelposix`	parallele POSIX Dateisystem-Aufrufe[1]
		`gpfs`	Wie `parallelposix`, aber nutze auch GPFS-Hints[1]
`parallel`	E	`true`/`false`	Legt fest, ob Checkpoint im Hintergrund erstellt werden soll
`path`	E		Ziel-/Quellverzeichnis für Checkpoints

[1] Siehe auch [YK04].

Parameter	Typ	Wert	Erklärung
`strategy`	E	`disable`[2]	Deaktiviere Checkpointing[3].
		`periodic`[2]	Verwende periodische Checkpointing-Strategie[3].
		`walltime`[2]	laufzeitabhängige Checkpointing-Strategie[3].
		`periodicwalltime`[2]	laufzeitabhängige periodische Checkpointing-Strategie[3].
		`validation`[2]	Validierungs-Strategie[3].
		`intensevalidation`[2]	intensive Validierungs-Strategie[3].
`workers`	B	`siminfo`[2]	SimInfo-Daten
		`pdf`[2]	PDF-Feld
		`floatpdf`[2]	PDF-Feld, schreibe immer als `float`
		`bcflagfield`[2]	Flags

C.2 FlagFieldWorker, BCFlagFieldWorker

Parameter	Typ	Wert	Erklärung
`filename`	E		Dateiname des Checkpoints
`tspacing`	E		Anzahl der Zeitschritte zwischen zwei Checkpoints

C.3 PeriodicStrategy

Die Parameter werden aus dem ersten Block mit dem Namen `periodic` innerhalb des ersten `checkpointing`-Blocks der Konfigurationsdatei ausgelesen.

Parameter	Typ	Wert	Erklärung
`filename`	E		Dateiname des Checkpoints
`tspacing`	E		Anzahl der Zeitschritte zwischen zwei Checkpoints

[2]Der Parameterwert wird durch die Anwendung bestimmt und kann abweichen.
[3]Für Erläuterungen siehe Abschnitt 3.5.

C.4 WalltimeStrategy

Die Parameter werden aus dem ersten Block mit dem Namen `walltime` innerhalb des ersten `checkpointing`-Blocks der Konfigurationsdatei ausgelesen.

Parameter	Typ	Wert	Erklärung
`filename`	E		Dateiname des Checkpoints
`tduration`	E		Ziellaufzeit der Simulation in Sekunden

C.5 PeriodicWalltimeStrategy

Die Parameter werden aus dem ersten Block mit dem Namen `periodicwalltime` innerhalb des ersten `checkpointing`-Blocks der Konfigurationsdatei ausgelesen.

Parameter	Typ	Wert	Erklärung
`filename`	E		Dateiname des Checkpoints
`tduration`	E		Ziellaufzeit der Simulation in Sekunden
`tspacing`	E		Anzahl der Zeitschritte zwischen zwei Checkpoints

D Verwendete Konfigurationen

D.1 Evaluierung

Die referenzierten Dateien entsprechen denen der *Checkpointing*-App und sind im entsprechenden SVN-Repository verfügbar (Revision 95).

Listing D.1: Zur Evaluierung verwendete Anwendungskonfiguration

```
1  Application{
2      Application          students;
3      Functionality        fsTRT;
4      Hardware             CPU;
5  }
6
7  SimInfo{
8      NumTimeSteps         'timesteps';
9  }
10
11 LBMParams {
12     trt{
13         lambda_e         'lambda_e';
14         lambda_D         'lambda_D';
15     }
16     dump;
17 }
18
19 SetupData{
20     dx                   'DX';
21     dt                   'DT';
22 }
23
24 PerfLogger{
25     TriggerInterval 301;
26 }
27
28 checkpointing {
29     fileaccess           lustre;
30     strategy             writebenchmark;
31 //      strategy            readbenchmark;
32     path                 /lxfs/iwia/iwia73;
33     workers {
34         siminfo;
35         pdfdirect;
36     }
37     alignment            16777216;
38     buffer               65536;
39 }
40
41 Patch{
42     xCellsBlock          'cellsX'/'xNumBlocks';
43     yCellsBlock          'cellsY'/'yNumBlocks';
44     zCellsBlock          'cellsZ'/'zNumBlocks';
45     AABB{
```

```
46          LowerPoint          <0,0,0>;
47          UpperPoint          <'xLength','yLength','zLength'>;
48      }
49  }
50
51  Logging{
52      Type                    nolog;
53      writeFile               noFile;
54  }
55
56  init{
57      // NoSlip West Wall //
58      Cells{
59          x                   0..0;
60          y                   0..'cellsY'+1;
61          z                   0..'cellsZ'+1;
62          NoSlip{}
63      }
64
65      // NoSlip East Wall //
66      Cells{
67          x                   'cellsX'+1..'cellsX'+1;
68          y                   0..'cellsY'+1;
69          z                   0..'cellsZ'+1;
70          NoSlip{}
71      }
72
73      // UBB North Wall //
74      Cells{
75          x                   1..'cellsX';
76          y                   'cellsY'+1..'cellsY' + 1;
77          z                   0..'cellsZ'+1;
78          NoSlip{}
79      }
80
81      // NoSlip South Wall //
82      Cells{
83          x                   1..'cellsX';
84          y                   0..0;
85          z                   0..'cellsZ'+1;
86          NoSlip{}
87      }
88
89
90      // NoSlip Bottom Wall //
91      Cells{
92          x                   1..'cellsX';
93          y                   1..'cellsY';
94          z                   0..0;
95          NoSlip{}
96      }
97
98      // UBB Top Wall //
99      Cells{
100         x                   1..'cellsX';
101         y                   1..'cellsY';
102         z                   'cellsZ'+1..'cellsZ'+1;
103         UBB{
104             u               <1,0.0,0.0>;
105         }
106     }
107 }
108
```

```
109
110 Physical_Check{
111     Parameters {
112         DX              1;
113         DT              0.43;
114         timesteps       100;
115         xLength         512;
116         yLength         512;
117         zLength         512;
118         xNumBlocks      16;
119         yNumBlocks      16;
120         zNumBlocks      16;
121
122         rho             1000;
123         tau             1.7;
124     }
125     Equations {
126         include         DIMEquations.inc;
127         include         LBMEquations.inc;
128     }
129     Units {
130         include         DIMUnits.inc;
131         include         LBMUnits.inc;
132     }
133 }
```

Abkürzungsverzeichnis

API	Application Programming Interface ... 17
CDF	Common Data Format ... 20
C/R	Checkpoint/Restart ... i
CRS	Checkpoint/Restart Service ... 5
HDF	Hierarchical Data Format ... 17
HDF5	Hierarchical Data Format 5 ... vii
HPC	High Performance Computing ... 14
GDO	Global Data Object ... 15
GPL	GNU General Public License ... 16
LBM	Lattice-Boltzmann-Methode ... 13
MMOG	Massively Multiplayer Online Game ... 13
MPI	Message Passing Interface ... 15
MTBF	Mean Time Between Failures ... 6
NASA	National Aeronautics and Space Administration ... 20
NCSA	National Center for Supercomputing Applications ... 17
netCDF	Network Common Data Form ... 20
NPB	NAS Parallel Benchmarks ... 13
PDF	Particle Distribution Functions ... 27
RRZE	Regionales Rechenzentrum Erlangen ... 46
STL	Standard Template Library ... 16
SVN	Subversion ... 47
VTK	Visualization Toolkit ... 15
waLBerla	widely applicable Lattice Boltzmann solver from Erlangen ... 1
XML	Extensible Markup Language ... 16

Literaturverzeichnis

[BCR+11] Bougeret, Marin ; Casanova, Henri ; Rabie, Mikael ; Robert, Yves ; Vivien, Frédéric: Checkpointing strategies for parallel jobs. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA : ACM, 2011 (SC '11). – ISBN 978–1–4503–0771–0, 33:1–33:11

[BGG+09] Bent, John ; Gibson, Garth ; Grider, Gary ; McClelland, Ben ; Nowoczynski, Paul ; Nunez, James ; Polte, Milo ; Wingate, Meghan: PLFS: a checkpoint filesystem for parallel applications. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA : ACM, 2009 (SC '09). – ISBN 978–1–60558–744–8, 21:1–21:12

[BMO06] Bauke, Heiko ; Mertens, Stephan ; Osterhage, Wolfgang W.: Checkpoint-Restart. Version: 2006. `http://dx.doi.org/10.1007/3-540-29928-9_13`. In: *Cluster Computing*. Springer Berlin Heidelberg, 2006 (X.systems.press). – ISBN 978–3–540–29928–8, 393-408. – 10.1007/3-540-29928-9_13

[BMPS03] Bronevetsky, Greg ; Marques, Daniel ; Pingali, Keshav ; Stodghill, Paul: C^3: A system for automating application-level checkpointing of MPI programs. In: *16TH INTERNATIONAL WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTERS (LCPC'03*, 2003, S. 357–373

[Dal03] Daly, John: A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps. Version: 2003. `http://dx.doi.org/10.1007/3-540-44864-0_1`. In: Sloot, Peter (Hrsg.) ; Abramson, David (Hrsg.) ; Bogdanov, Alexander (Hrsg.) ; Gorbachev, Yuriy (Hrsg.) ; Dongarra, Jack (Hrsg.) ; Zomaya, Albert (Hrsg.): *Computational Science — ICCS 2003* Bd. 2660. Springer Berlin / Heidelberg, 2003. – ISBN 978–3–540–40197–1, 724-724. – 10.1007/3-540-44864-0_1

[DAR12] Dawes, Beman ; Abrahams, David ; Rivera, Rene: *Boost Library Documentation*. Version: 2012. `http://www.boost.org/doc/libs/`. Online

[Don11] Donath, Stefan: *Wetting Models for a Parallel High-Performance Free Surface Lattice Boltzmann Method*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., 2011. – 1–222 S

[ESB99] Evan, Hazim Abdel-Shafi ; Speight, Evan ; Bennett, John K.: Efficient User-Level Thread Migration and Checkpointing on Windows NT Clusters. In: *In Usenix 1999 (3rd Windows NT Symposium*, 1999, S. 1–10

[Fei12] Feichtinger, Christian: *Entwurf und Performance-Evaluierung eines Software-Frameworks für Multi-Physik-Simulationen auf heterogenen Supercomputern*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., 2012

[FHK+11] Folk, Mike ; Heber, Gerd ; Koziol, Quincey ; Pourmal, Elena ; Robinson, Dana: An overview of the HDF5 technology suite and its applications. In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. New York, NY, USA : ACM, 2011 (AD '11). – ISBN 978–1–4503–0614–0, 36–47

[FRB+11] Ferreira, Kurt B. ; Riesen, Rolf ; Brighwell, Ron ; Bridges, Patrick ; Arnold, Dorian: libhashckpt: hash-based incremental checkpointing using GPU's. In: *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*. Berlin, Heidelberg : Springer-Verlag, 2011 (EuroMPI'11). – ISBN 978–3–642–24448–3, 272–281

[FSF12] Free Software Foundation, Inc.: *Verschiedene Lizenzen und Kommentare*. Version: Juli 2012. `http://www.gnu.org/licenses/license-list.html`, Abruf: 14.07.2012. Online

[HD06] Hargrove, Paul H. ; Duell, Jason C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters. In: *Journal of Physics: Conference Series* 46 (2006), Nr. 1, 494. `http://stacks.iop.org/1742-6596/46/i=1/a=067`

[HDF12a] HDF Group: *HDF5 User's Guide*. Version: Mai 2012. `http://www.hdfgroup.org/HDF5/doc/UG/`, Abruf: 14.07.2012. Online

[HDF12b] HDF Group Help Desk: *HDF5: API Specification Reference Manual*. Version: 05 2012. `http://www.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html`, Abruf: 01.10.2012. Online

[HR08] Hartnett, Edward ; Rew, R.K.: *Experience with an enhanced netCDF data model and interface for scientific data access*. 2008

[HRC09] Hacker, Thomas J. ; Romero, Fabian ; Carothers, Christopher D.: An analysis of clustered failures on large supercomputing systems. In: *J. Parallel Distrib. Comput.* 69 (2009), Juli, Nr. 7, 652–665. `http://dx.doi.org/10.1016/j.jpdc.2009.03.007`. – DOI 10.1016/j.jpdc.2009.03.007. – ISSN 0743–7315

[Hur10] Hursey, Joshua: *Coordinated Checkpoint/Restart Process Fault Tolerance for MPI Applications on HPC Systems.* Bloomington, IN, USA, Indiana University, Diss., Juli 2010

[Koz11] Koziol, Quincey: *[Hdf-forum] H5Fclose hangs when using parallel HDF5.* Online. `http://mail.hdfgroup.org/pipermail/hdf-forum_hdfgroup.org/2011-September/005093.html`. Version: 26.09.2011, Abruf: 25.10.2012

[LES10] Litvinova, Antonina ; Engelmann, Christian ; Scott, Stephen L.: A Proactive Fault Tolerance Framework for High-Performance Computing. In: *Proceedings of the 9th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2010.* Innsbruck, Austria : ACTA Press, Calgary, AB, Canada, Februar 16-18, 2010. – ISBN 978–0–88986–783–3,

[LF90] Li, C.-C.J. ; Fuchs, W.K.: CATCH-compiler-assisted techniques for checkpointing. In: *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, 1990, S. 74–81

[LH10] Laadan, Oren ; Hallyn, Serge E.: Linux-CR: Transparent Application Checkpoint-Restart in Linux. In: *The Linux Symposium 2010* Linux Symposium Inc., 2010, S. 159–172

[LTBL97] Litzkow, Michael ; Tannenbaum, Todd ; Basney, Jim ; Livny, Miron: Checkpoint and Migration of UNIX Processes in the COndor Distributed Processing System / University of Wisconsin - Madison Computer Sciences Department. 1997 (UW-CS-TR-1346). – Forschungsbericht. –

[Mil85] Mili, A.: Towards a Theory of Forward Error Recovery. In: *Software Engineering, IEEE Transactions on* SE-11 (1985), aug., Nr. 8, S. 735 – 748. `http://dx.doi.org/10.1109/TSE.1985.232523`. – DOI 10.1109/TSE.1985.232523. – ISSN 0098–5589

[NEC09] NEC Corporation: *Fast and reliable data storage in Computer-aided Automotive Engineering.* Version: 06 2009. `http://www.nec.com/en/de/en/prod/solutions/hpc-solutions/success-stories/material/lxfs_eng.pdf`, Abruf: 29.09.2012. Online

[NMES07] Nagarajan, Arun B. ; Mueller, Frank ; Engelmann, Christian ; Scott, Stephen L.: Proactive fault tolerance for HPC with Xen virtualization. In: *Proceedings of the 21st annual international conference on Supercomputing.* New York, NY, USA : ACM, 2007 (ICS '07). – ISBN 978–1–59593–768–1, 23–32

[PBKL95] Plank, J. S. ; Beck, M. ; Kingsley, G. ; Li, K.: Libckpt: Transparent Checkpointing under Unix. In: *Usenix Winter Technical Conference*, 1995, S. 213–223

[RD90] Rew, Russ ; Davis, Glenn: *NetCDF: An Interface for Scientific Data Access*. Juli 1990

[RDE+12] Rew, R. ; Davis, G. ; Emmerson, S. ; Davies, H. ; Hartnett, E.: *The NetCDF Users' Guide*. Juli 2012

[Reg12] Regionales Rechenzentrum Erlangen: *LiMa Cluster*. Version: März 2012. http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/systeme/lima-cluster.shtml, Abruf: 29.09.2012. Online

[Rew12] Rew, Russ: *NetCDF FAQ*. Version: 2012. http://www.unidata.ucar.edu/software/netcdf/#netcdf_faq, Abruf: 16.07.2012. Online

[RKBS10] Rajagopalan, Manoj ; Koziol, Quincey ; Benger, Werner ; Sharpe, James: *hdf-forum - HDF5 C++ API and programming model*. Version: April 2010. http://hdf-forum.184993.n3.nabble.com/HDF5-C-API-and-programming-model-td757443.html, Abruf: 26.07.2012. Online

[RMG+10] Rodríguez, Gabriel ; Martín, María J. ; González, Patricia ; Touriño, Juan ; Doallo, Ramón: CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications. In: *Concurr. Comput. : Pract. Exper.* 22 (2010), April, Nr. 6, 749–766. http://dx.doi.org/10.1002/cpe.v22:6. – DOI 10.1002/cpe.v22:6. – ISSN 1532–0626

[Rom02] Roman, Eric: A Survey of Checkpoint/Restart Implementations / Lawrence Berkeley National Laboratory, Tech. 2002 (LBNL-54942). – Forschungsbericht. –

[Sha11] Shahzad, Faisal: *Checkpoint/Restart for Fault Tolerant MPI Programs: A Case Study using a Lattice Boltzmann Code and the NAS Parallel Benchmarks*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diplomarbeit, 2011

[SPD+05] Sancho, Jose C. ; Petrini, Fabrizio ; Davis, Kei ; Gioiosa, Roberto ; Jiang, Song: Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In: *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18 - Volume 19*. Washington, DC, USA : IEEE Computer Society, 2005 (IPDPS '05). – ISBN 0–7695–2312–9, 300.2–

[Str98] Strumpen, Volker: *Compiler Technology for Portable Checkpoints*. 1998

[Top12] Top500: *Top 500 Supercomputer Sites*. Version: 2012. http://www.top500.org/, Abruf: 14.07.2012. Online

[Tro08] Troyer, Matthias: *Boost Mailing List: Any plan for mpi2 IO support?* Online. http://lists.boost.org/boost-users/2008/08/38934.php. Version: 05.08.2008, Abruf: 07.06.2012

[TSH+00] Takahashi, Toshiyuki ; Sumimoto, Shinji ; Hori, Atsushi ; Harada, Hiroshi ; Ishikawa, Yutaka: PM2: High Performance Communication Middleware for Heterogeneous Network Environments. In: *In SC'00*, 2000, S. 52–53

[Vai97] Vaidya, Nitin H.: Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme. In: *IEEE Trans. Comput.* 46 (1997), August, Nr. 8, 942–947. http://dx.doi.org/10.1109/12.609281. – DOI 10.1109/12.609281. – ISSN 0018–9340

[VSCS+09] Vaz Salles, Marcos ; Cao, Tuan ; Sowell, Benjamin ; Demers, Alan ; Gehrke, Johannes ; Koch, Christoph ; White, Walker: An evaluation of checkpoint recovery for massively multiplayer online games. In: *Proc. VLDB Endow.* 2 (2009), August, Nr. 1, 1258–1269. http://dl.acm.org/citation.cfm?id=1687627.1687769. – ISSN 2150–8097

[XCZ08] Xue, Ruini ; Chen, Wenguang ; Zheng, Weimin: CprFS: a user-level file system to support consistent file states for checkpoint and restart. In: *Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA : ACM, 2008 (ICS '08). – ISBN 978–1–60558–158–3, 114–123

[YK04] Yang, MuQun ; Koziol, Quincey: *Parallel HDF5 Hints*. Version: 2004. http://www.hdfgroup.org/HDF5/PHDF5/parallelhdf5hints.pdf. Online

[You74] Young, John W.: A first order approximation to the optimum checkpoint interval. In: *Commun. ACM* 17 (1974), September, Nr. 9, 530–531. http://dx.doi.org/10.1145/361147.361115. – DOI 10.1145/361147.361115. – ISSN 0001–0782

[Zan05] Zandy, Victor C.: *ckpt*. Version: 2005. http://pages.cs.wisc.edu/~zandy/ckpt/, Abruf: 14.07.2012. Online

[Zei10] Zeiser, Thomas: *Rechnerzuwachs und Generationswechsel bei den HPC-Clustern am RRZE*. Version: September 2010.

http://blogs.fau.de/zeiser/2010/09/21/rechnerzuwachs-und-generationswechsel-bei-den-hpc-clustern-am-rrze/, Abruf: 29.09.2012. Online