Duale Hochschule Baden-Württemberg Mannheim

Studiengang Informationstechnik

## Bachelorarbeit

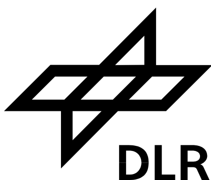
Bearbeitungszeitraum: 24. Juni 2013 - 16. September 2013

# Entwicklung eines systematischen Testkonzepts für ein hoch paralleles Softwarepaket aus dem Bereich der Materialwissenschaften


von **Timo Tischler**

*- Matrikelnr.: 6413549 -*

*- Kurs: TIT10ANS -*




Deutsches Zentrum für
Luft- und Raumfahrt e.V.
in der Helmholtz-Gemeinschaft

Standort: Köln

Einrichtung für Simulations- und Softwaretechnik
Abteilung: Verteilte Systeme und Komponentensoftware

Betreuer:
Dr. Jonas Thies
Christin Keutel (B. Sc.)
Prof. Dr. Harald Kornmayer

**Zusammenfassung**

Viele Software-Entwickler legen wenig Wert auf Tests, obwohl diese eine enorme Bedeutung für die Stabilität und Qualität von Software haben. In dieser Bachelorarbeit soll daher ein systematisches Testkonzept für das Projekt ESSEX entwickelt werden. Im Projekt sollen Eigenwertlöser für große, dünnbesetzte Matrizen implementiert werden. Diese sollen Exascale-Computer-fähig sein und sind daher hoch parallel. Die Grundlagen von Softwaretests werden beschrieben und stellen die Basis des Konzepts dar. Um die Software angemessen testen zu können, müssen mehrere Besonderheiten im Projekt ESSEX betrachtet werden. Wichtig sind vor allem Performance, Parallelität und Fehlertoleranz. Auf dieser Basis soll das Testkonzept entworfen werden. Dies geschieht mit der Unterstützung verschiedener Tools, die erläutert und evaluiert werden. Einige dieser Tests werden exemplarisch umgesetzt, um die Funktionalität der Tools zu beweisen. Weiterhin ist Testautomatisierung ein wichtiger Punkt in dieser Arbeit. Die Entwickler akzeptieren das entstandene Testkonzept und werden sich mit künftigen Testaktivitäten daran orientieren.

*Timo Tischler, September 2013*

**Abstract**

Many software developers think that software tests are a burden, although they are very important for the stability and quality of software products. This bachelor thesis is about developing a test concept for ESSEX. ESSEX is a project which deals with implementing eigenvalue solvers for large sparse matrices. These algorithms have to run on exascale computing systems. Therefore they are highly parallel. The basics of software testing are described and form the basis of the concept. Several particularities in the ESSEX project have to be analyzed and considered to test the software adequately. Important test criteria are performance, parallelism and fault tolerance. The test concept is designed according to this basis. Different tools support the testing process. These have to be evaluated before usage. Some examples are implemented to demonstrate the functionality of the tools. Furthermore, test automation is an important aspect of this thesis. The developers accept the resulting test concept and will base their future test activities on it.

*Timo Tischler, September 2013*

DHBW
Duale Hochschule
Baden-Württemberg
**Mannheim**

Studiengang Informationstechnik

## Bachelorarbeit

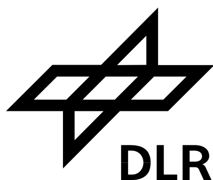
Bearbeitungszeitraum: 24. Juni 2013 - 16. September 2013

# Entwicklung eines systematischen Testkonzepts für ein hoch paralleles Softwarepaket aus dem Bereich der Materialwissenschaften


von **Timo Tischler**

*- Matrikelnr.: 6413549 -*

*- Kurs: TIT10ANS -*



Deutsches Zentrum für
Luft- und Raumfahrt e.V.
in der Helmholtz-Gemeinschaft

Standort: Köln

Einrichtung für Simulations- und Softwaretechnik
Abteilung: Verteilte Systeme und Komponentensoftware

Betreuer:
Dr. Jonas Thies
Christin Keutel (B. Sc.)
Prof. Dr. Harald Kornmayer

# Eidesstattliche Erklärung

Hiermit versichere ich, dass ich
die vorliegende Arbeit selbstständig und nur unter
Verwendung der angegebenen Quellen und
Hilfsmittel angefertigt habe.

---

Köln, der 13. September 2013

# Inhaltsverzeichnis

**Abbildungsverzeichnis** **IX**

**Listings** **X**

**Abkürzungsverzeichnis** **XI**

**1 Einleitung** **1**

1.1 Umfeld der Arbeit . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Motivation der Arbeit . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Aufgabenstellung . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.4 Vorgehensweise und Aufbau dieser Arbeit . . . . . . . . . . . . . . . . 5

**2 Grundlagen** **7**

2.1 Das Projekt ESSEX . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.1.1 Beschreibung des Projekts . . . . . . . . . . . . . . . . . . . . 7

2.1.2 Technische Grundlagen . . . . . . . . . . . . . . . . . . . . . 9

2.2 Testen allgemein . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.2.1 Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.2.2 Begriffe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.2.3 Testebenen . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.4 Arten von Tests . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.3 Vorgehensweise beim Testen . . . . . . . . . . . . . . . . . . . . . . . . 25
2.3.1 Proof of Concept . . . . . . . . . . . . . . . . . . . . . . . . 25
2.3.2 Proof of Performance . . . . . . . . . . . . . . . . . . . . . . 25
2.3.3 Testautomatisierung . . . . . . . . . . . . . . . . . . . . . . . 26
2.4 Besonderheiten bei Tests . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.4.1 Testen von numerischen Codes . . . . . . . . . . . . . . . . . . 27
2.4.2 Testen von parallelen Anwendungen . . . . . . . . . . . . . . . 28

**3 Entwicklung eines Testkonzepts 30**
3.1 Anforderungen an das Testkonzept . . . . . . . . . . . . . . . . . . . . 30
3.1.1 Nebenläufigkeit . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.1.2 Numerische Codes . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.1.3 Fehlertoleranz . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.1.4 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.2 Verwendung von Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
3.2.1 Google C++ Testing Framework . . . . . . . . . . . . . . . . . 37
3.2.2 Valgrind . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.2.3 Tool zum Testen von MPI-Anwendungen . . . . . . . . . . . . 39
3.3 Grundsätzliche Vorgaben im Testkonzept . . . . . . . . . . . . . . . . . 40
3.4 Komponententests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.5 Integrationstests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.6 Systemtests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.7 Continuous Integration . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**4 Exemplarische Umsetzung einiger Tests 47**
4.1 Erstellung von Tests mit Hilfe des Google C++ Testing Frameworks . . 47
4.2 Testfallerstellung für die Haupt-Orthogonalisierung-Routine im Projekt 50

4.3 Nebenläufigkeitstests . . . . . . . . . . 52
4.3.1 Auf Node-Ebene mittels Valgrind . . . . . . . . . . 52
4.3.2 Testen der MPI-Funktionen . . . . . . . . . . 58

**5 Fazit und Ausblick** **60**
5.1 Fazit . . . . . . . . . . 60
5.2 Ausblick . . . . . . . . . . 62

**A Testfälle für den Gram-Schmidt-Algorithmus** **63**
A.1 Spezifikation des Algorithmus . . . . . . . . . . 63
A.2 Testfälle . . . . . . . . . . 64

**B Testkonzept nach IEEE 829** **65**

**Literaturverzeichnis** **70**

# Abbildungsverzeichnis

1 Darstellung einer Node mit mehreren Prozessoren (Optional mit Grafikprozessor) . . . 2
2 10er-Regel der Fehlerkosten . . . 3
3 Parallelisierung mittels MPI und OpenMP . . . 12
4 Vereinfachtes Beispiel eines Kontrollflussgraphen . . . 23
5 Einfachster Fall eines Deadlocks . . . 28
6 Valgrind überwacht den gemeinsamen Speicher einer Node . . . 38
7 MUST analysiert die MPI-Kommunikation . . . 40
8 Beispiel: Ausgabedatei von MUST . . . 41
9 Performance-Überblick des Projekts FreeWake in der Jenkins Web-API 46

# Listings

1 Beispiel für eine Compilerdirektive von OpenMP . . . . . . . . . . . . . 12

2 Beispiel für einen Testfall in Google Test . . . . . . . . . . . . . . . . . 48

3 Besipiel für die Ausgabe von Google Test bei einem fehlerfreien Testdurchlauf . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

4 Google Test Ausgabe für einen Performancetest . . . . . . . . . . . . . 50

5 Beispiel einer suppression . . . . . . . . . . . . . . . . . . . . . . . . . 54

6 CMake-Test-Ausgabe für einen fehlerfreien Testdurchlauf . . . . . . . . 57

# Abkürzungsverzeichnis

**CMake** **C**ross-platform **Make**

**DFG** **D**eutsche **F**orschungs**g**emeinschaft

**DLR** **D**eutsches Zentrum für **L**uft- und **R**aumfahrt

**ESA** **E**uropean **S**pace **A**gency

**ESSEX** **E**quipping **S**parse **S**olvers for **Ex**ascale

**ESSR** **E**xascale **S**parse **S**olver **R**epository

**FLOPS** **Flo**ating **P**oint Operations per **S**econd

**HPC** **H**igh **P**erformance **C**omputing

**HTML** **H**yper**t**ext **M**arkup **L**anguage

**IDE** **I**ntegrated **D**evelopment **E**nvironment

**IEC** **I**nternational **E**lectrotechnical **C**ommission

**IEEE** **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers

**ISO** **I**nternational **O**rganization for **S**tandardization

**ISTQB** **I**nternational **S**oftware **T**esting **Q**ualification **B**oard

**MPI** **M**essage **P**assing **I**nterface

**OpenMP** **Open** **M**ulti **P**rocessing

**VM** **V**irtuelle **M**aschine

**XML** E**x**tensible **M**arkup **L**anguage

# 1 Einleitung

In diesem Kapitel wird zunächst das Umfeld der Arbeit beschrieben. Dabei handelt es sich um eine kurze Übersicht über die Praxiseinrichtung, in der diese Arbeit verfasst wurde. Daraufhin wird die Motivation gefolgt von der Vorgehensweise und dem Aufbau dieser Arbeit beschrieben.

## 1.1 Umfeld der Arbeit

Das Deutsche Zentrum für Luft- und Raumfahrt e.V. ist eine staatliche Forschungseinrichtung, die in den Bereichen Luftfahrt, Raumfahrt, Energie, Verkehr und Sicherheit tätig ist. Das DLR ist sowohl in nationale als auch in internationale Kooperationen, wie z.B. mit der **E**uropean **S**pace **A**gency (ESA)[ESA13] eingebunden. Außerdem ist das DLR zuständig für die Planung und Umsetzung der deutschen Raumfahrtaktivitäten. Das Unternehmen beschäftigt ca. 7400 Mitarbeiter und Mitarbeiterinnen in 32 Instituten und Einrichtungen und unterhält 16 Standorte in Deutschland, sowie 4 Außenbüros in Paris, Brüssel, Tokio und Washington D.C.[DLR13a]
Die Einrichtung "Simulations- und Softwaretechnik", in der ich während meiner Praxisphasen arbeite, befasst sich nicht nur mit der Entwicklung neuer Softwaretechnologien, die den Wissenschaftlern im DLR bei ihrer Arbeit helfen, sondern auch mit der Erforschung und Bereitstellung innovativer Software-Engineering-Technologien innerhalb und außerhalb des DLR[DLR13b].

Die Abteilung "Verteilte Systeme und Komponentensoftware" ist spezialisiert auf die Entwicklung komponentenbasierter Software für modernes ingenieurwissenschaftliches Arbeiten unter Nutzung service-orientierter Architekturen. Die Abteilung ist weiter unterteilt in drei Gruppen. Diese Arbeit wird im Umfeld der HPC-Gruppe (**H**igh **P**erformance **C**omputing) erstellt. Dort werden speziell Algorithmen für Hochleistungs- und Supercomputer entwickelt. Hierbei handelt es sich meist um parallele, numerische Algorithmen, die auf hunderten Nodes gleichzeitig laufen. Node ("Knoten") bezeichnet eine Einheit bei parallelen Berechnungen. Diese umfasst mehrere Rechnerkomponenten, die auf einer Platine verbaut sind. So kann eine Node zum Beispiel nur einen einzigen Prozessor, aber auch einen Mehrkernprozessor inklusive Grafikkarte enthalten (siehe *Abbildung 1*). Mehrere Nodes sind zum Beispiel über das Netzwerk verbunden. Auf diesen Nodes werden zusätzlich mehrere Prozesse ausgeführt. Dies führt zu einem hohen Performance-Gewinn gegenüber sequentiellen Algorithmen.

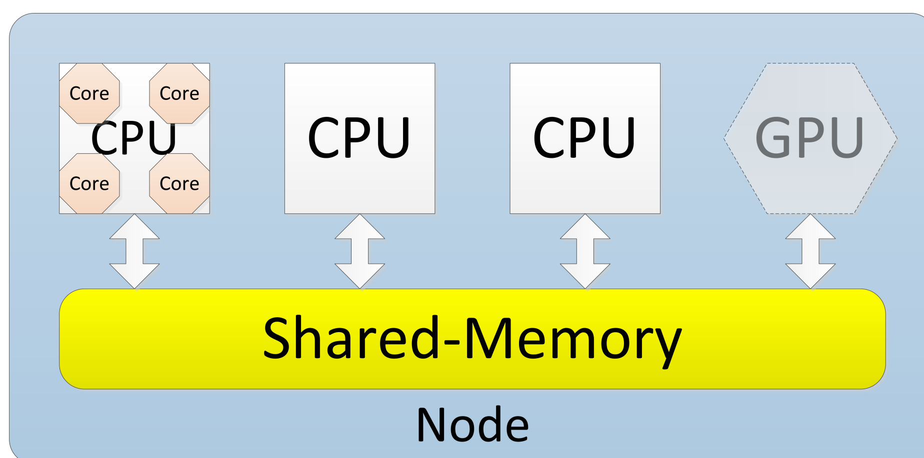



**Abbildung 1:** Darstellung einer Node mit mehreren Prozessoren (Optional mit Grafikprozessor)

## 1.2 Motivation der Arbeit

*"The fewer tests you write, the less productive you are and the less stable your code becomes."*[GHJV94] ist ein bekanntes Zitat von Erich Gamma zum Thema Softwaretest. In der Tat haben Softwaretests eine enorm wichtige Bedeutung für ein Softwareprodukt. Erst durch die Erstellung und Durchführung verschiedener Tests lässt sich ein qualitativ hochwertiges Produkt, welches verschiedenen Standards wie z.B. der ISO (**I**nternational **O**rganization for **S**tandardization[ISO13a]) 9126[BKL+09] gerecht wird, entwickeln. In der Praxis werden Softwaretests allerdings häufig aus Zeit- oder Geldmangel vernachlässigt bzw. ans Ende des Entwicklungsprozesses gestellt[DR06]. Dies führt jedoch später zu höheren Fehlerkosten. Man kann grob sagen, dass sich die Fehlerkosten mit jeder Phase der Entwicklung verzehnfachen (siehe *Abbildung 2*).

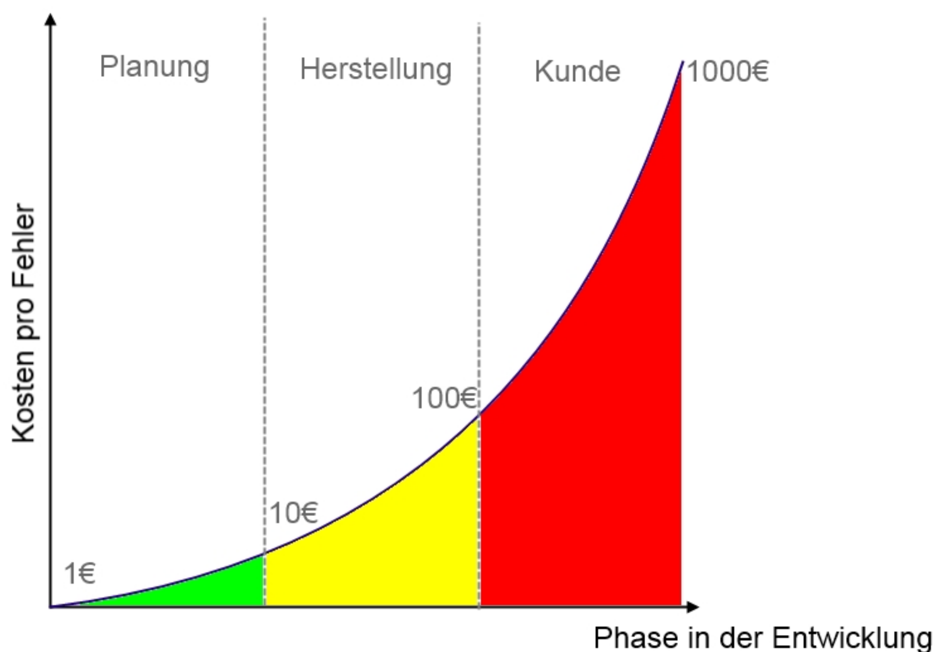



**Abbildung 2:** 10er-Regel der Fehlerkosten

Daher hat es sich bewährt, möglichst früh mit dem Testprozess zu beginnen und diesen möglichst parallel zur Softwareentwicklung laufen zu lassen.
Durch fehlerbehaftete Software kann es im Extremfall zu Schäden im Milliardenbereich kommen. Als Beispiel ist hier der Fehlstart der Ariane 5[1] zu nennen. Beim Start dieser Rakete kam es am 4. Juni 1996 durch einen Softwarefehler, der indirekt den Selbstzerstörungsmechanismus auslöste, zu einem Schaden in Höhe von 370 Millionen US-Dollar[Lan96]. Der Fehler trat auf, als eine 64-Bit Gleitkommazahl in einen 16-Bit Ganzzahlwert umgewandelt werden sollte. Dieses Softwarefragment wurde für die Vorgängerrakete Ariane 4, deren Maximalgeschwindigkeit jedoch geringer war, entwickelt[Lio96]. Dieser Fehler im Programmcode wäre wahrscheinlich durch einen simplen Test gefunden worden. Um ein Fehlverhalten der Algorithmen aus dem Projekt ESSEX zu verhindern und um eine angemessene Softwarequalität und eine hohe Ausfallsicherheit gewährleisten zu können, soll bereits in einer frühen Entwicklungsphase mit dem Testen begonnen werden.
Die Besonderheit an dieser Arbeit ist, dass im Projekt hoch parallele Algorithmen implementiert werden. Deren Testvorgehen ist noch nicht besonders weit erforscht. Ein Testkonzept erscheint an dieser Stelle besonders sinnvoll, damit ein einheitliches Testverfahren, an welchem sich Entwickler oder Tester orientieren können, umgesetzt werden kann. Gerade in dem Bereich des *High Performance Computing* ist ein gut durchdachtes Testkonzept essentiell, um eine stabile Software zu entwickeln. Bei jeder Berechnung werden enorme Mengen an Ressourcen benötigt. Tritt eine Fehlerwirkung auf, werden Ergebnisse unbrauchbar oder es kommt sogar zu Systemausfällen. In diesen Fällen muss eine neue Berechnung erfolgen, wobei die Ressourcen weiterhin belegt sind.

[1] Ariane 5 ist eine Trägerrakete, mit der Satelliten in die Erdumlaufbahn befördert werden[ari13] können.

## 1.3 Aufgabenstellung

Ziel der Arbeit ist es, ein systematisches Testkonzept für ein Softwarepaket aus dem Bereich der Materialwissenschaften zu entwickeln. Bei dieser Software handelt es sich um eine Sammlung hoch paralleler Algorithmen zur Lösung von Eigenwertproblemen dünnbesetzter Matrizen. Es ist zunächst wichtig, dass eine umfangreiche Literaturrecherche durchgeführt und deren Ergebnis dargestellt wird. Es sollen verschiedene Tools, die das Testen vereinfachen könnten, auf ihre Tauglichkeit untersucht werden. Wichtig ist neben dem Auffinden von Fehlern auch die Überprüfung der Performance der implementierten Algorithmen. Es soll daher ein Konzept für verschiedene Schichten der Software entwickelt werden. Dieses Konzept soll als Grundlage für die Entwickler bzw. Tester dienen, damit eine Software entsteht, die verschiedenen Qualitätsansprüchen genügt.

## 1.4 Vorgehensweise und Aufbau dieser Arbeit

In dieser Arbeit werden zunächst die theoretischen Grundlagen ausführlich erläutert. Dabei steht die Beschreibung von verschiedenen Begriffen und Methoden, die für die Entwicklung des Testkonzepts benötigt werden, im Vordergrund. Dies geschieht auf Basis einer umfangreichen Literaturrecherche zum Thema Softwaretest. Daraufhin soll der "state of the art"[2] des Softwaretestens vorgestellt werden. Insbesondere werden hier verschiedene theoretische Grundlagen und Probleme von Tests bei parallelem Code beschrieben. Hierbei werden ebenfalls Tools, die ein systematisches Testen erleichtern könnten, erläutert. Außerdem wird das Projekt ESSEX allgemein und mit seinen technischen Grundlagen vorgestellt.

Im nächsten Schritt werden die unterschiedlichen Methoden des Testens evaluiert und diejenigen herausgearbeitet werden, die für das Testkonzept relevant sind. Auf dieser

[2] der aktuelle Stand des Softwaretestens, also welche Verfahren werden aktuell angewandt usw.

Grundlage wird dann das Testkonzept erstellt. In diesem Konzept soll einerseits festgelegt werden, wie eine möglichst hohe Fehlererkennung erreicht werden und andererseits, wie die Performance[3] getestet werden kann. Ein weiterer Schwerpunkt ist das Testen der Nebenläufigkeit der parallelen Algorithmen. In dem Testkonzept soll ein einheitliches Verfahren für Komponenten-, Integrations- und Systemtests erstellt werden.
Es handelt sich bei einigen Punkten um eine theoretische Ausarbeitung der Möglichkeiten während andere Teile des Konzepts daraufhin exemplarisch umgesetzt und die Ergebnisse evaluiert werden. Abgeschlossen wird diese Arbeit mit einem Fazit und einem Ausblick in die Zukunft. Darin befindet sich auch eine Evaluation der gesamten Arbeit, mit der übergeordneten Fragestellung, ob das Testkonzept in der Praxis so angewendet werden kann und soll.

[3]die Ausführungsgeschwindigkeit der Algorithmen

# 2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen, die für diese Bachelorarbeit benötigt werden, erläutert. Zunächst wird das Projekt ESSEX und dessen Zusammenhänge im Groben vorgestellt. Daraufhin werden die Grundlagen des Softwaretestens und Vorgehensweisen für einen erfolgreichen Softwaretest beschrieben. Die Besonderheiten, die für den Test der ESSEX-Software beachtet werden müssen, werden im Anschluss betrachtet.

## 2.1 Das Projekt ESSEX

Im Projekt ESSEX (**E**quipping **S**parse **S**olvers for **Ex**ascale)[WBF+12] werden verschiedene numerische Algorithmen zur Lösung von großen, dünnbesetzen[4] Matrixproblemen, wie z.B. Eigenwertberechnungen, entwickelt. In diesem Kapitel werden zunächst das Projekt und der Projektaufbau beschrieben. Daraufhin werden die technischen Grundlagen erläutert und grundlegende Werkzeuge erklärt.

### 2.1.1 Beschreibung des Projekts

ESSEX ist ein Projekt der Deutschen Forschungsgemeinschaft[dfg13] (DFG), an dem verschiedene deutsche Forschungseinrichtungen, unter anderem auch das DLR, beteiligt

[4] Eine dünnbesetzte Matrix ist eine Matrix, die so viele Nullen enthält, dass es sich für die Berechnung lohnt diese auszunutzen[spa12]

sind. Das Besondere an den im Rahmen des Projekts entwickelten Algorithmen ist, dass sie hoch parallel[5] und für Exascale-Computer optimiert sind. Ein Exascale-Computer ist ein Rechnersystem, das eine Leistung von mindestens einem ExaFLOPS[6] (**Flo**ating **P**oint Operations per **S**econd[7]) erreicht. Alle erstellten Algorithmen sollen im ESSR (**E**xascale **S**parse **S**olver **R**epository) zusammengefasst werden. Die dort gesammelten Algorithmen werden zur Lösung verschiedener physikalischer Probleme genutzt, wie z.B. zur Untersuchung von Graphen-Strukturen. Graphen ist eine Modifikation von Kohlenstoff mit einer bienenwabenförmigen Struktur, die in der Physik eine hohe Bedeutung hat. Es ist zum Beispiel sehr stabil und dennoch sehr elastisch. Außerdem besitzt Graphen viele weitere Eigenschaften, unter anderem eine bessere Wärmeleitfähigkeit als Kupfer und eine extrem hohe elektrische Leitfähigkeit. Die Einsatzgebiete für Graphen sind sehr weitreichend. So ist es der Firma IBM[ibm12] gelungen, den schnellsten Transistor mit Hilfe dieses Materials zu entwickeln. Außerdem ist es möglich den Wirkungsgrad von Photovoltaikanlagen mittels Graphen auf 60% zu erhöhen[gra12]. Es wird häufig als "Wundermaterial des 21. Jahrhunderts"[8][CL12] bezeichnet.
Das DLR ist mit der Bearbeitung des Arbeitspakets *C-WP1* dieses Projekts beauftragt. In diesem Arbeitspaket soll der *Jacobi-Davidson-Algorithmus* für Eigenwertprobleme dünnbesetzter Matrizen umgesetzt werden[BDD+00]. Weitere Schwerpunkte sind die Entwicklung von Vorkonditionierern und iterativen Lösern für Gleichungssysteme (*Krylov-Verfahren*). Die Basisoperationen, wie Matrix-Vektor- oder Matrix-Matrix-Multiplikationen werden von den Entwicklern des Rechenzentrums Erlangen entwickelt. Die Implementierung basiert auf den so genannten Kernel-Interfaces. Diese Schnittstellen sollen es beispielsweise ermöglichen, die Algorithmen beibehalten zu können, wenn die Basisoperationen geändert werden. Alle Grundoperationen werden auf Basis dieses Interfaces implementiert. Daraus werden Bibliotheken erstellt, die die Algorithmen

[5] d.h. die Berechnung erfolgt parallel in bis zu 1 Mio. Prozessen gleichzeitig
[6] Exa = $10^{18}$
[7] Gleitkomma-Operationen pro Sekunde
[8] im Original: "miracle material[s] in the twenty-first century"

nutzen können. Eine dieser Bibliotheken ist die von der Universität Erlangen entwickelten Bibliothek **Ghost** (siehe Kapitel 2.1.2).

### 2.1.2 Technische Grundlagen

In dem folgenden Kapitel werden die technischen Grundlagen des Projekts ESSEX näher beschrieben. In diesem Zusammenhang werden Technologien und Werkzeuge erläutert, die den Softwareentwicklungsprozess unterstützen und vereinfachen. Ein einheitliches Vorgehensmodell wird im Projekt nicht verfolgt. Eine grobe Orientierung liefert allerdings das Modell der Einrichtung für Simulations- und Softwaretechnik[scW13]. Dessen Grundlage ist der *Agile Unified Process*[aup12]. Die *Construction*-Phase ist allerdings eher an *Scrum*[scr13] angelehnt.
Die Software wird zunächst ausschließlich für Linux-Distributionen entwickelt.

#### Build-Tool

Der build-Prozess im Projekt ESSEX wird durch CMake (**C**ross-platform **Make**)[cma] unterstützt. CMake wird verwendet, um Softwareprodukte auf verschiedenen Plattformen zu bauen. Es erstellt für die verwendete Linux-Distribution ein klassisches *make file*. Um dies zu realisieren, benötigt man das so genannte *CMake file*. In diesem sind alle Dinge, die zur Erstellung eines make files benötigt werden beschrieben. Darunter zählen unter anderem die Abhängigkeiten zu verschiedenen Bibliotheken oder die Pfade des jeweiligen source- oder build-Ordners. Des Weiteren bietet CMake die Möglichkeit, mittels des Befehls *make test* die angegebenen Testszenarien auszuführen. Dies wird benötigt um die Testausführung zu automatisieren. Außerdem können eigene Targets definiert werden. Dies bietet sich an um beispielsweise die Performance-Tests unabhängig ausführen zu können.

### Aufbau der Software

Die Softwarekomponenten, die im DLR entwickelt werden, sollen in den Programmiersprachen C und C++ implementiert werden. Für fertiggestellte Softwarekomponenten soll eine Fortran-Schnittstelle zur Verfügung gestellt werden. Die Entwicklung geschieht grundsätzlich auf einem Linux-Betriebssystem und es wird in der Regel keine Entwicklungsumgebung genutzt.

### Ghost

Die Bibliothek Ghost wird an der Universität Erlangen entwickelt. Sie enthält zum einen einige wichtige Grundoperationen wie z.B. das Matrix-Vektor-Produkt für dünnbesetzte Matrizen und vollbesetzte Vektoren und zum anderen das queueing-System. Die genannten Grundoperationen sind dabei so implementiert, dass sie möglichst effizient und performant auf unterschiedlichen Rechnerarchitekturen ausgeführt werden. Dazu zählen ebenfalls Systeme, bei denen die Berechnung durch Grafikprozessoren unterstützt wird. Weiterhin ist ein queueing-System[9] in dieser Bibliothek eingebaut. Dazu wird ein Thread-Pool initialisiert, in welchem alle Threads "warten". Dies wird über Semaphoren realisiert. Wird nun zur queue ein task[10] hinzugefügt, wird diese dem ersten aktiven Thread zugeordnet. Dieser arbeitet die ihm zugewiesene Aufgabe ab, wird wieder inaktiv und wartet auf die nächste Aufgabe.

### MPI

MPI (**M**essage **P**assing **I**nterface)[mpi] ist eine Schnittstelle zur Parallelisierung und ein Standard für den Nachrichtenaustausch von verteilten Systemen. MPI verwendet verschiedene Befehle, die das Senden und Empfangen von Nachrichten zwischen mehreren

[9] Eine queue ist eine Warteschlange. Sie funktioniert nach dem FILO-Prinzip (first in, last out)
[10] eine Aufgabe

Prozessen ermöglicht. Nachrichten können als Broadcast versendet werden oder einen expliziten Sender und Empfänger haben. Sie enthalten mehrere Parameter, mit denen die gewünschte Information übertragen werden kann. Es gibt sowohl MPI Implementierungen, deren Nutzung an Lizenzen gebunden ist, als auch freie Implementierungen. Welche dieser Implementierungen genutzt wird, ist plattformabhängig. Auf den Systemen, mit denen die Tests beispielhaft durchgeführt wurden, wurde OpenMP[ope] verwendet.
Zusätzlich zu MPI werden weitere Schnittstellen zur Parallelisierung verwendet. Zunächst liegt der Hauptaugenmerk auf der Verwendung von OpenMP. Gewünscht wird aber zusätzlich, dass Grafikkarten zur Berechnung hinzugezogen werden können. Dafür kann beispielsweise CUDA[nVI], auf das hier nicht weiter eingegangen wird, verwendet werden.

### OpenMP

OpenMP[ope] ist ebenfalls eine Schnittstelle zur Parallelisierung, allerdings auf Thread-Ebene. Sie ermöglicht die Shared-Memory-Programmierung[11] in den Sprachen C, C++ und Fortran. Um dies zu verwirklichen nutzt OpenMP Compilerdirektiven, d.h. im Programmcode wird angegeben, welche Teile der Software parallel ausgeführt werden sollen (Beispiel in Listing 1). OpenMP ist lizenzfrei und daher ohne Einschränkungen nutzbar.
Durch die Verwendung von MPI und OpenMP ist es möglich, Software für Supercomputer zu erstellen. OpenMP übernimmt dabei die Parallelisierung auf Node-Ebene, während MPI den Nachrichtenaustausch zwischen den Nodes ermöglicht (siehe *Abbildung 3*).

[11] Shared-Memory bedeutet, dass ein gemeinsamer Speicher, den sich mehrere Prozesse teilen, zur Verfügung steht.

```
1 #pragma omp parallel for // OpenMP Compilerdirektive fuer eine for-Schleife
2  for (i=0;i<n;i++)
3  {
4      [...]
5  }
```

**Listing 1:** Beispiel für eine Compilerdirektive von OpenMP

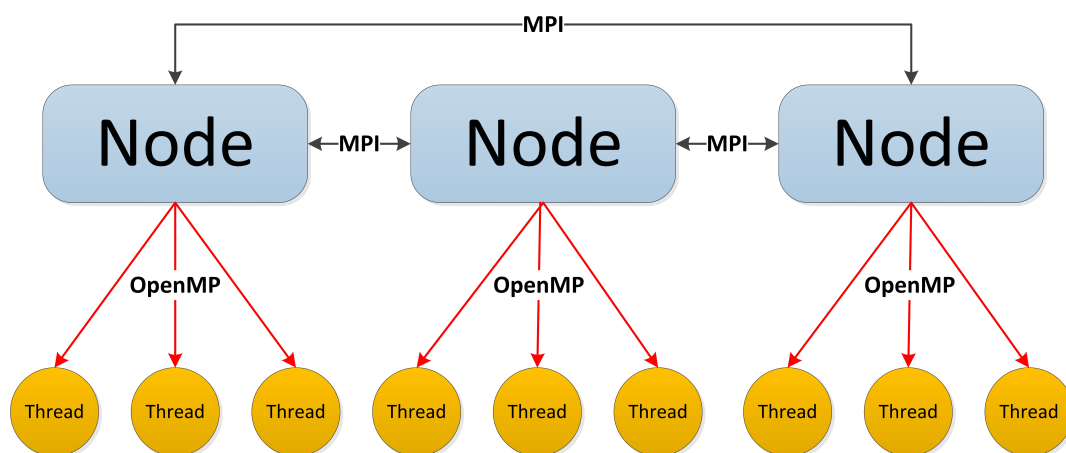



**Abbildung 3:** Parallelisierung mittels MPI und OpenMP

## 2.2 Testen allgemein

In diesem Kapitel wird der Begriff des Softwaretestens näher erläutert. Zunächst werden Definitionen von Softwaretests verglichen und eine für diese Arbeit gültige Definition aufgestellt. Anschließend werden grundlegende Begriffe aus dem Bereich des Softwaretestens erläutert. Daraufhin werden verschiedene Arten des Testens und die Vorgehensweise (der "state of the art", vgl. Kapitel 1.4) beim Testen und bei der Testautomatisierung vorgestellt. Außerdem gibt es einen Abschnitt, in dem die Besonderheiten bei Tests für die Software aus dem Projekt ESSEX beschrieben werden.

### 2.2.1 Definition

Es gibt verschiedene Möglichkeiten den Begriff *Softwaretest* zu definieren. Myers stellt bereits 1979 in der Erstausgabe des Buchs[12] *The Art of Software Testing*[Mye79] folgende Definition auf:
*"Testing is the process of executing a program with the intent of finding errors."*
Dabei handelt es sich nur um eine sehr grundlegende Definition, die nicht den vollen Umfang des Themas beschreibt. Das, was Myers als Definiton angibt, kann eher als eines von mehreren Zielen des Testens angesehen werden.
In einer etwas ausführlicheren Definiton wird der Softwaretest als *"Prozess des Planens, der Vorbereitung und der Messung"*[PKS02] beschrieben. Dieser Prozess verfolgt das Ziel, das IT-System zu analysieren und den Unterschied zwischen Ist- und Soll-Zustand aufzuzeigen. Wichtig hierbei ist, dass es sich beim Soll-Zustand nicht zwangsläufig um die Spezifikation des Kunden handelt, da diese häufig nicht fehlerfrei ist. Der Soll-Zustand umschreibt daher eher den erforderlichen Endzustand. Diese Defintion spezifiziert den Softwaretest genauer, sodass Kriterien, wie die umfangreiche Planung und Vorbereitung, für einen erfolgreichen Test mit aufgenommen und auch das Ziel des Testens detaillierter beschrieben werden. Letztere Definition soll für diese Bachelorarbeit genügen.
An dieser Stelle ist anzumerken, dass die Begriffe "Testen" und "Debuggen" häufig gleichgesetzt werden. Dies ist allerdings nicht korrekt. Beim Testen werden verschiedene Ziele, wie das Nachweisen und die Vorbeugung von Fehlerwirkungen, sowie die Bestimmung der Softwarequalität verfolgt. Erst beim Debugging[13] sollen die Fehlerursachen gefunden und korrigiert werden.
Als einer der wichtigsten Grundsätze beim Testen gilt, dass durch Testen nur die Anwesenheit und niemals die Abwesenheit von Fehlern nachgewiesen werden kann.

[12] aktuelle Ausgabe erschien 2011[MSB11]
[13] Debugging ist Aufgabe des Entwicklers, nicht des Testers[SL12]

Selbst durch eine Testabdeckung von 100% kann nie bewiesen werden, dass keine Fehlerwirkungen mehr auftreten.

### 2.2.2 Begriffe

Für das weitere Verständnis dieser Arbeit werden im Folgenden verschiedene Begriffe, die für Softwaretests eine große Bedeutung haben, erläutert. Die folgenden Erklärungen und Begriffsdefinitonen sind zum Großteil ISTQB-konform (**I**nternational **S**oftware **T**esting **Q**ualification **B**oard)[ist13][SL12].

#### Fehler

Unter **Fehler** wird allgemein die fehlerhafte Ausgabe einer Software bezeichnet. Um Ursache und Wirkung zu unterscheiden, ist es angebracht, den Fehlerbegriff weiter zu unterteilen.
Zunächst gibt es die **Fehlhandlung**[Ber13]. Diese bezeichnet eine Fehlleistung des Softwareentwicklers, die in den allermeisten Fällen die Ursache für einen **Fehlerzustand** ist. Der Begriff **Fehlerzustand** beschreibt beispielsweise die vergessene oder falsch programmierte Anweisung im Programm[SL12]. Ein **Fehlerzustand** führt (nicht zwangsläufig[14]) bei der Ausführung des Programms zur **Fehlerwirkung**[SL12][Ber13]. Die Fehlerwirkung ist also die nach außen sichtbare Erscheinung des Fehlers. Dabei kann es allerdings vorkommen, dass eine Fehlerwirkung eine oder mehrere weitere auslösen kann. Des Weiteren können Fehlerwirkungen auch durch Umwelteinflüsse auf die Hardware entstehen. Strahlung und Magnetismus[15] sind beispielsweise zwei Einflüsse, die zu einer Fehlerwirkung ohne Fehlhandlungen führen können. Außerdem ist es möglich, dass Fehlerwirkungen durch andere Fehlerzustände "verdeckt" werden.

[14] In einigen Fällen werden Fehlerwirkungen erst später zum Beispiel durch bestimmte Eingaben ausgelöst

[15] zum Beispiel bei Einwirkung auf eine Festplatte

Dabei spricht man von **Fehlermaskierung**[SL12]. Diese Fehler werden möglicherweise erst dann erkannt, wenn ein anderer behoben wird.
Zu unterscheiden von den Fehlern sind die so genannten **Mängel**. Diese beschreiben ein abweichendes Verhalten von den Spezifikationen[Com90]. Fehler und Mängel werden neben weiteren Definitionen von der IEEE (**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers)[iee13] unter dem Begriff **Softwareanomalien** zusammengefasst und klassifiziert [Soc10].

#### Qualität

Im Zusammenhang mit Softwaretests wird in dieser Arbeit häufig der Begriff **Qualität** erwähnt. Objektiv kann Qualität nur über verschiedene Standards und Normen bestimmt werden. Die ISO/IEC (**I**nternational **E**lectrotechnical **C**ommission)[IEC10] Normenreihe beginnend mit 250xx[iso13b] enthält viele Richtlinien, nach denen Software entwickelt werden sollte. Insbesondere die ISO-Norm 25010[iso13c], welche 2001 die Norm mit der Nummer 9126 abgelöst hat[iso13d], gibt verschiedene Faktoren vor, die für ein qualitativ hochwertiges Softwareprodukt bedeutsam sind. Dazu zählen Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit[SL12]. Diese Faktoren sollten bei der Entwicklung und auch beim Testen berücksichtigt werden.

### 2.2.3 Testebenen

Im Softwareentwicklungszyklus wird das Testen in verschiedene Ebenen aufgeteilt. Damit soll sichergestellt sein, dass sowohl die einzelnen Komponenten, als auch das System als Ganzes, getestet und weitgehend fehlerfrei sind. Man unterscheidet weiterhin zwischen funktionalen und nicht funktionalen Tests. Analog zu funktionalen und nicht funktionalen Anforderungen im Entwicklungsprozess, handelt es sich um verschiedene

Aspekte, die beim Testen beachtet werden. Beim funktionalen Testen wird die Richtigkeit der Software überprüft. Nicht funktionale Tests umfassen beispielsweise Anforderungen wie Benutzbarkeit und Performance.

### Komponententests

Die unterste Ebene stellen die **Komponenten**- bzw. **Unit-Tests** dar. Dabei werden die einzelnen Komponenten isoliert von allen anderen Softwarebausteinen getestet. Wichtig hierbei ist, dass die Komponente definiert wird. Eine Komponente umfasst üblicherweise eine Funktion. Der Tester kann auf die Eingabeschnittstelle der Funktion zugreifen und die Funktion mit den erforderlichen Informationen ausführen. Er vergleicht daraufhin das Ergebnis der Funktion mit dem, welches er manuell im Vorhinein erstellt hat. Hierbei werden häufig Berechnungsfehler und nicht korrekt implementierte Sonderfälle erkannt. Auf dieser Ebene bietet es sich ebenfalls an, die Effizienz der Komponente zu überprüfen. In höheren Schichten lässt sich diese nur mit einem erheblich höheren Aufwand bewerkstelligen[SL12].
Die Teststrategie gibt vor, wie das Testen auf Komponentenebene durchgeführt wird. Es besteht die Möglichkeit, Blackbox- und Whitebox-Verfahren für die Testfallerstellung zu verwenden. Allerdings bietet es sich gerade beim Unit-Test an, Whitebox-Verfahren zu benutzen, da der Zugang zum Source-Code gegeben ist und so sehr entwicklungsnah gearbeitet werden kann. Die Erstellung von Komponententests nimmt in der Regel den Großteil der Zeit in Anspruch. Häufig wird sogar 80% der Testzeit für die Entwicklung und Implementierung der Komponententests genutzt. Dadurch, dass die Komponenten gut durchgetestet sind, ist es weniger aufwändig, die Tests der höheren Ebenen durchzuführen.

### Integrationstests

Beim **Integrationstest** werden einzelne Komponenten im Zusammenspiel überprüft. Dabei kann es sich beispielsweise um vollständige Algorithmen, die verschiedene Funktionen (also Komponenten) nutzen, handeln. Selbst, wenn ein vollständiger Komponententest durchgeführt wurde, können zum Beispiel auf dem Kommunikationsweg zwischen zwei Komponenten Fehler auftreten. Ursachen dafür können inkompatible Schnittstellen sein[16].

Für einen erfolgreichen Integrationstest müssen die Komponenten schrittweise zusammengebaut werden. Für jedes so entstandene Teilsystem sollte ein Integrationstest durchgeführt werden. Wie die Komponenten zu Teilsystemen zusammengesetzt werden, wird durch die Integrationsstrategie definiert. Häufig wird hier in der Reihenfolge der Fertigstellung der Komponenten vorgegangen. Dies ermöglicht eine Integration ohne hohen Aufwand und hohe Kosten. Dieses Vorgehen nennt sich **Ad-hoc-Integration**. Es gibt weitere Integrationsstrategien wie z.B. **Top-down-Integration** und **Bottom-up-Integration**. Bei letzterem beginnt man bei den Komponenten, die keine weiteren mehr aufrufen. Diese werden dann weiter zusammengesetzt, bis das System vollständig ist. Bei der Top-down-Integration geschieht dies umgekehrt.

### Systemtests

Die nächsthöhere Stufe ist der **Systemtest**. Hierbei wird das gesamte System getestet, um zu überprüfen, ob es den Spezifikationen entspricht. Der Systemtest soll möglichst auf einer Hardware und unter Bedingungen, die dem Endbenutzer-System sehr ähnlich sind, durchgeführt werden. Man testet das System für gewöhnlich nicht direkt in der Kundenumbgebung, da durch Fehlerwirkungen auch Datenverluste oder ähnliches entstehen können.

[16] Beispiel: Funktion 1 liefert 3 Ausgabewerte, Funktion 2, die mit diesen Werten arbeiten soll, benötigt allerdings 4 Eingabewerte.

### Abnahmetests

Abschließend erfolgt der **Abnahmetest**. Diesen findet man selten in wissenschaftlichen Umgebungen, da Softwareprodukte hier im Allgemeinen nicht für einen Kunden entwickelt werden. Auf dieser Ebene ist es wichtig, dass der Kunde beim Test anwesend ist und die Software bewerten kann.
Soll die Software auf vielen verschiedenen Systemen ausgeführt werden, besteht außerdem die Möglichkeit, im Rahmen des Abnahmetests einen *Feldtest*[17] durchzuführen. Bei diesem wird das Produkt an eine ausgewählte Menge von Benutzern ausgeliefert. Diese sollen die Software möglichst alltagsgetreu nutzen, Fehlerwirkungen und gegebenenfalls Verbesserungsvorschläge dokumentieren und den Entwicklern mitteilen. Dadurch bekommen die Entwickler die Möglichkeit, die Software nach Wünschen der Benutzer anzupassen.

### Testen nach Änderungen

Wird eine Software verändert, weil Bugs behoben, die Funktionalität erweitert oder der Code lediglich zur besseren Übersicht umstrukturiert wurde, müssen auch die zugehörigen Tests gegebenenfalls überarbeitet und erneut ausgeführt werden.

## 2.2.4 Arten von Tests

Die verschiedenen Arten von Softwaretests lassen sich grob in zwei Gruppen unterteilen: die **statischen** und die **dynamischen** Tests. Diese zwei Arten und deren Unterarten werden in diesem Abschnitt beschrieben.

[17] auch Alpha- oder Beta-Test genannt

### Statische Tests

Bei statischen Tests wird im Gegensatz zu den dynamischen Tests das Programm für die Tests nicht zur Ausführung gebracht. Bei statischen Tests handelt es sich um **Reviews** oder **Quellcodeanalysen**, die üblicherweise mittels Werkzeugen durchgeführt werden[Maj12]. Ziel der statischen Analyse ist immer das Aufdecken von Verstößen gegen die Spezifikationen[SL12]. Diese Art von Tests ermöglicht es, viele Fehler bereits vor der Ausführung der dynamischen Tests aufzudecken und zu beheben, wodurch der Aufwand und die Kosten, die für diese Tests entstehen, erheblich verringert werden.
Bei **Reviews** oder **strukturierten Gruppenanalysen** sollen mehrere Mitarbeiter die Software gemeinsam analysieren. Dabei sollen Fehlerzustände entdeckt und möglicherweise behoben werden. Bei Reviews sind sowohl die Tester, als auch die Entwickler vertreten und tauschen sich über Probleme im Softwareprodukt aus. Dabei ist es wichtig, dass die geäußerte Kritik sich nicht an die jeweilige Person, sondern ausschließlich an die Software richtet[SL12]. Durch ein derart gestaltetes Review kann sogar das Arbeitsklima und die Zusammenarbeit der Mitarbeiter verbessert werden[Maj12]. Jede Art von Review sollte sich an den *Standard for Software Reviews* der IEEE[Soc08], in der die wichtigen Schritte für ein erfolgreiches Review beschrieben werden, halten. In dieser Norm wird außerdem zwischen verschiedenen Arten von Reviews unterschieden, die in Tabelle 1 aufgelistet sind.
Neben den Reviews gibt es außerdem die **statische Analyse**. Hierbei wird der Programmcode auf verschiedene Kriterien hin untersucht. Das bekannteste und am meisten genutzte Werkzeug zur statischen Analyse ist der *Compiler*. Dieser kann syntaktisch inkorrekte Anweisungen nicht übersetzen und vermerkt diese in einer Liste von Fehlern. Diese Liste wird im Regelfall bereits vom Entwickler ausgewertet und Fehlerzustände können behoben werden.
Verschiedene Datentypen, wie z.B. XML (E**x**tensible **M**arkup **L**anguage) und HTML (**H**yper**t**ext **M**arkup **L**anguage), können unter Beachtung verschiedener Standards auto-

| Art des Reviews | Ziele | Vorbereitung | Formalität | Zusatz |
|---|---|---|---|---|
| Walkthrough | • Fehler, Defekte, Unklarheiten identifizieren<br>• Verteilung von Wissen<br>• Diskussion von Lösungsalternativen | • geringe bis keine Vorbereitung | • informell | • geeignet für kleine Entwicklungsteams |
| Inspektion | • Fehler, Defekte, Unklarheiten identifizieren<br>• Verbesserung der Qualität von Prüf- und Entwicklungsprozess | • recht hoher Vorbereitungsaufwand<br>• Checklisten erstellen<br>• Eintritts- und Austrittskriterien für Prüfschritte festlegen | • (sehr) formal | • sehr umfangreich<br>• strenger Ablauf |
| Technisches Review | • Übereinstimmung von Dokument und Spezifikation<br>• Eignung für den Einsatz untersuchen | • viel Vorbereitung<br>• Objekte im Vorhinein überprüfen | • unterschiedlich | • Fachexperten als Gutachter |
| Informelles Review | • Fehler, Defekte, Unklarheiten identifizieren | • kaum Vorbereitung<br>• Auswahl der Gutachter<br>• festlegen der Abgabetermine | • informell | • "buddy testing"<br>• "pair programming" |

**Tabelle 1:** Arten von Reviews (nach [SL12] und [Soc08])

matisiert validiert werden. Mittels Kontroll- und Datenflussanalyse können ungenutzte oder nicht initialisierte Variablen bzw. so genannter "toter Code"[18] lokalisiert werden.

### Dynamische Tests

Dynamische Tests beschreiben diejenige Art von Tests, bei denen das jeweilige Testobjekt zur Ausführung gebracht wird. Auch diese lassen sich weiter unterteilen. Die zwei Hauptkategorien sind die **Blackbox**- und die **Whitebox-Tests**.
Beim Blackbox-Test wird das Programm bzw. das jeweilige Programmstück nur von außen betrachtet, d.h. der innere Ablauf ist nicht bekannt. Lediglich Ein- und Ausgabewerte werden überprüft. Die Testfälle können demnach nur aus der Spezifikation abgeleitet werden oder sind in dieser bereits beschrieben. Um konkrete Testfälle zu erstellen nutzt man zum Beispiel **Äquivalenzklassentests** oder die **Grenzwertanalyse**. Bei der ersten Variante werden zu jedem Eingabeparameter verschiedene (gültige und ungültige) **Äquivalenzklassen** gebildet. Diese werden jeweils so gewählt, dass alle Werte aus der selben Äquivalenzklasse zum gleichen Testergebnis führen. Nun werden jeweils gültige und ungültige Testfälle erstellt. Dabei gilt zu beachten, dass bei gültigen Testfällen alle Eingabewerte gültig sein müssen und bei ungültigen nur genau ein Eingabewert ungültig sein darf.
Die Grenzwertanalyse verfolgt ein ähnliches Prinzip. Hierbei werden die Randwerte der Äquivalenzklassen überprüft, da bei diesen Werten häufig Fehlerwirkungen auftreten. In der Praxis werden häufig beide Verfahren genutzt, um die jeweiligen Testfälle für das Blackbox-Testen zu erstellen[SL12]. Es existieren noch weitere Blackbox-Verfahren. Diese werden an dieser Stelle aber nicht vorgestellt.
Die Whitebox-Tests unterscheiden sich im Wesentlichen dadurch, dass die Testfälle aus dem Code abgeleitet werden. Der Vorteil daran ist, dass dadurch auch Programmteile getestet werden, die nicht in der Spezifikation beschrieben sind, aber die Funktionalität der

[18] Code, der z.B. aufgrund einer Verzweigung nie zur Ausführung kommt

Software erweiteren. Bei den Whitebox-Tests gibt es ebenfalls verschiedene Verfahren, die in der Praxis angewendet werden. Von diesen werden im Folgenden die wichtigsten drei vorgestellt. Für alle drei Verfahren muss ein *Kontrollflussgraph* erstellt werden. Deshalb existiert für sie in der Literatur auch die Bezeichnung *kontrollflussorientierte Testverfahren*[Roi05]. Die einzelnen Knoten in diesem Graphen entsprechen Schleifen oder Verzweigungen im Programmcode. Ein beispielhafter Kontrollflussgraph ist in *Abbildung 4* zu sehen.

Eines dieser Verfahren nennt sich **Anweisungsüberdeckungstest**. Verfolgt man den Ansatz der Anweisungsüberdeckung, so müssen möglichst alle Anweisungen (*Knoten* im Graphen) ein Mal erreicht werden. Im Beispiel müssten die Anweisungen in folgender Reihenfolge ausgeführt werden:

$a \rightarrow c \rightarrow e \rightarrow f \rightarrow d \rightarrow g$

Die Anweisungsüberdeckung beträgt dann 100%[19]. Dieser Wert[20] gilt als nicht sehr aussagekräftig.

Ein weiteres Maß für Whitebox-Tests ist die **Zweigüberdeckung**, bei der, anders als bei der Anweisungsüberdeckung, jede *Kante* im Graphen mindestens ein Mal durchlaufen werden muss, um eine 100%ige Zweigüberdeckung zu erreichen. Dieses Maß[21] gilt als aussagekräftiger, da 100% Zweigüberdeckung eine volle Anweisungsüberdeckung implizieren aber nicht umgekehrt. Im Beispiel wird Zweig *b* bei der Anwendungsüberdeckung nicht weiter beachtet. Um eine volle Zweigüberdeckung zu gewährleisten, muss dieser Zweig ebenfalls zur Ausführung kommen. Es ergibt sich als weiterer Testfall also folgender Pfad:

$a \rightarrow b \rightarrow g$

[19] Anweisungsüberdeckung = Anzahl besuchter Knoten / Gesamtzahl der Knoten[Kle12]
[20] auch C0-Maß genannt[SL12]
[21] auch C1-Maß

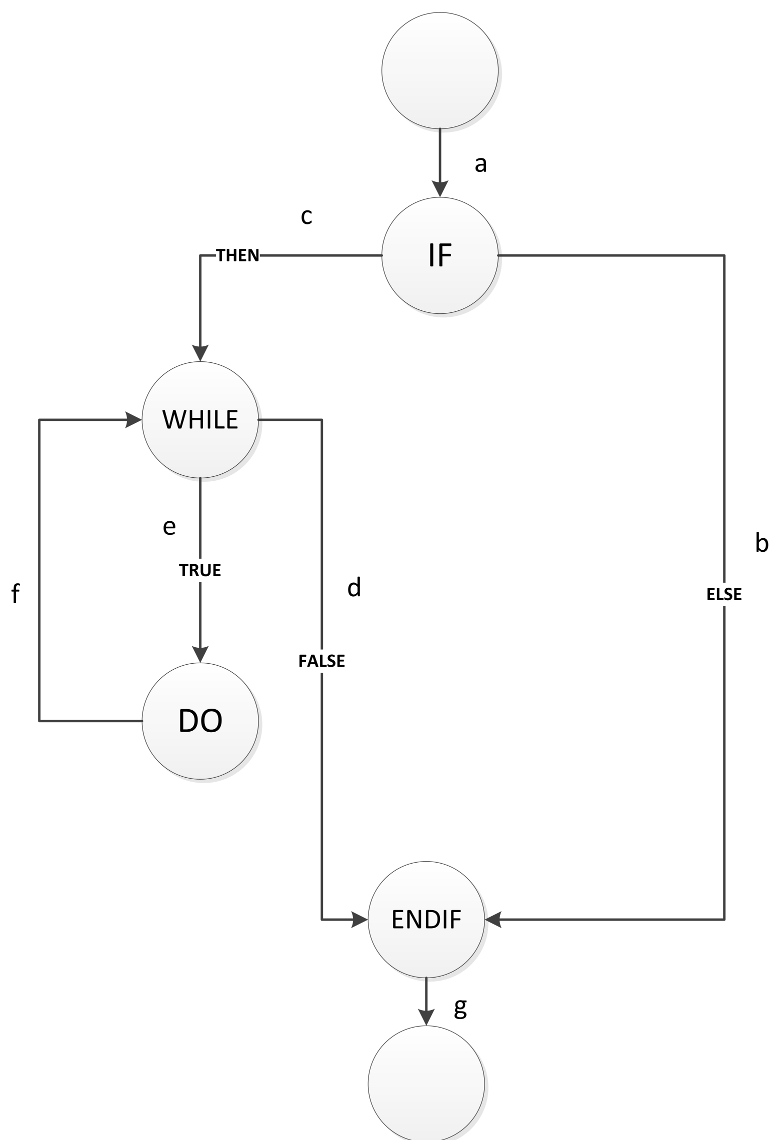



**Abbildung 4:** Vereinfachtes Beispiel eines Kontrollflussgraphen

Beide Verfahren eignen sich nur bedingt für objektorientierte Systeme, da Anweisungen innerhalb einer Klasse häufig nicht sehr umfangreich sind und die wichtigere Kommunikation zwischen den Objekten stattfindet. Außerdem können sie nur bei *atomaren*[22] Bedingungen eingesetzt werden.

Existieren im Programmcode kombinierte Bedingungen, so muss die **Bedingungsüber-**

[22]Das bedeutet, dass eine Bedingung nicht aus mehreren Bedingungen zusammengesetzt ist.

**deckung** betrachtet werden. An dieser Stelle wird nur das wichtigste und häufigste Verfahren, die **minimale Mehrfachbedingungsüberdeckung** erläutert. Dabei handelt es sich um einen Kompromiss zwischen der *einfachen Bedingungsüberdeckung*, die ein sehr schwaches Kriterium darstellt, und der *Mehrfachbedingungsüberdeckung*, aus der zwar viele Testfälle abgeleitet werden können, die allerdings sehr aufwendig ist, da die Zahl der Kombinationen der Parameterwerte exponentiell ansteigt[SL12].
Bei der minimalen Mehrfachbedingungsüberdeckung müssen nicht alle Kombinationen der Parameter überprüft werden, sondern nur diejenigen, bei denen eine Änderung der atomaren Einzelbedingung auch eine Änderung des Gesamtausdrucks zur Folge hat.
Auch bei den Whitebox-Tests gibt es noch weitere Verfahren, die aufgrund ihrer fehlenden Relevanz für diese Arbeit nicht genauer erläutert werden.
Für die Whitebox-Tests wird zusätzlich noch ein **Testendekriterium** benötigt. Dabei handelt es sich um eine für das Projekt einheitlich festgelegte Abdeckungsquote. Diese ist unter Umständen sogar vertraglich geregelt und wird vom Kunden vorgeschrieben. Als Beispiel könnte eine Zweigabdeckung von 80% als ein angemessenes Testendekriterium dienen.
Zusätzlich zu den Blackbox- und Whitebox-Verfahren gibt es noch das **intuitive** oder **erfahrungsbasierte Testen**, bei dem die Kenntnisse des Testers eine entscheidende Rolle spielen. Bei diesen Verfahren soll der Tester Testfälle, die seiner Meinung nach eine Relevanz haben, ausarbeiten. Weiß er, dass ein Programmcode an einer bestimmten Stelle sehr fehleranfällig sein kann (zum Beispiel aufgrund der verwendeten Programmiersprache), oder hat er Erfahrungen mit ähnlichen Projekten, kann er dies nutzen, um die Qualität der Software mittels weiterer Tests zu erhöhen. Außerdem sind Fehler in der Software im Regelfall nicht gleichverteilt. Die Wahrscheinlichkeit, dass dort, wo bereits ein Fehler gefunden wurde, weitere Fehler sind, ist erhöht. Dieses Wissen kann der Tester ebenfalls nutzen, um seine Testfälle an kritischen Stellen zu erweitern.

## 2.3 Vorgehensweise beim Testen

Üblicherweise wird der Testvorgang in zwei Schritte unterteilt, die im Folgenden beschrieben werden. Insbesondere wenn die Performance eines Programms eine große Rolle spielt, ist es sinnvoll, diese Unterscheidung vorzunehmen.

### 2.3.1 Proof of Concept

Als erster Schritt wird ein so genanntes *Proof of Concept* durchgeführt. Dabei soll hauptsächlich gezeigt werden, dass die Softwarefunktionen die erwarteten Ergebnisse liefern. Dazu zählt, dass die Software mit jeder Art von Eingabe umgehen kann und eine angemessene Ausgabe liefert. Wichtig ist auch, dass die entwickelte Software den Spezifikationen (des Kunden) gerecht wird.
Ein *Proof of Concept* wird mittels der in Kapitel 2.2.4 vorgestellten Methoden durchgeführt. In der Praxis reicht es hier allerdings häufig nicht aus, sich auf eine Vorgehensweise zu beschränken. Es sollten möglichst sowohl statische als auch dynamische Testverfahren miteinbezogen werden. Bei letzteren sollten ebenfalls Blackbox- und Whitebox-Testverfahren genutzt werden. Unter Umständen sollten weitere Testfälle mit Hilfe des intuitiven oder erfahrungsbasierten Testens erstellt werden. Die Kombination dieser verschiedenen Testverfahren, ermöglicht es, ein Softwareprodukt zu erstellen, welches den Kriterien der ISO Normen entspricht[iso13b].

### 2.3.2 Proof of Performance

Für Softwareprodukte, bei denen die Performance ebenfalls eine große Rolle spielt, sollte zusätzlich zum *Proof of Concept* ein *Proof of Performance*, mit dessen Hilfe man verschiedene Performance-Aspekte testet, erstellt werden. Eine simple Form des *Proof of Performance* ist die Auswertung der Laufzeit des Programms. Diese sollte unterhalb bestimmter Schwellenwerte liegen, die ebenfalls spezifiziert werden können.

Beim *Proof of Performance* können außerdem Kriterien wie die effiziente Speichernutzung und Skalierbarkeit der Software überprüft werden. Man spricht in diesem Zusammenhang auch von *nicht funktionalen Tests*. Häufig unterscheiden sich diese Kriterien in der Entwicklungsumgebung von denen in der späteren Anwendungsumgebung. Daher wird häufig eine Trennung vorgenommen und die Performance wird auf beiden Systemen unabhängig voneinander getestet. Beim *Proof of Performance* werden ebenfalls Flaschenhälse[23] des Programms aufgedeckt und können so behoben werden. Durch eine Analyse und Auswertung der erhaltenen Testdaten können dann, falls nötig, verschiedene Verfahren zur Optimierung der Performance angewandt werden.
Ein *Proof of Performance* wird für gewöhnlich mittels verschiedener Werkzeuge, von denen einige in Kapitel 3.2 vorgestellt werden, durchgeführt.

### 2.3.3 Testautomatisierung

Die manuelle Durchführung von Tests ist gegebenenfalls sehr umständlich. Vor allem bei häufiger Ausführung des Tests, weil beispielsweise der Code verändert wurde (vgl. Kapitel 2.2.3 *Testen nach Änderungen*), müssen Tests automatisiert werden. Dazu ist es notwendig, dass die Ausführung der Tests durch ein Skript oder ein Testwerkzeug geschieht. Automatisierte Tests erleichtern den Prozess des Testens erheblich. Insbesondere nach Code-Änderungen oder Erweiterungen ermöglicht eine Automatisierung der Softwaretests eine schnelle Identifikation von Fehlerwirkungen. Das führt dazu, dass der Testprozess möglichst kosteneffizient durchgeführt wird.
Ein weiterer Vorteil von automatisiertem Testen ist, dass die Tests in ein Continuous Integration-System (kontinuierliche Integration) eingebunden werden können. Dies beschreibt den automatischen build-Prozesses und die Ausführung von Tests, die bei

[23]Teile eines Systems, die die Performance maßgeblich beeinträchtigen. Bei einem Computer zum Beispiel handelt es sich bei der Festplatte (HDD) häufig um einen Flaschenhals, da sie eine deutlich langsamere Lese- und Schreibgeschwindigkeit hat, als alle anderen Bauteile verarbeiten können.

jedem commit[24] in das Versionsverwaltungssystem, ausgelöst werden. So kann man direkt feststellen, ob alles erfolgreich abgeschlossen wurde und ob alle Tests fehlerfrei waren. Bei SC wird dafür ein Jenkins-Server[jen13] verwendet. Mit diesem erweiterbaren Open-Source-Server, lässt sich der Build-Status optimal überwachen. Ergebnisse von ausgeführten Tests, auch bezüglich der Performance, lassen sich im Jenkins-Web-Interface graphisch darstellen.
Wichtig bei der Automatisierung von Testprozessen ist, dass genug Kenntnisse in der manuellen Umsetzung dieser Tests gegeben ist. Werkzeuge, die automatisiert eingesetzt werden sollen, müssen geeignet und in der Praxis bewährt sein. Sind die Testprozesse nicht ausgereift oder wurden manuelle Tests nicht im Vorhinein gut durchdacht, ist es sinnvoller zunächst das Konzept zu überdenken bzw. zu ändern. Erst dann sollte über eine Automatisierung nachgedacht werden. *"It is far better to improve the effectiveness of testing first than to improve the efficiency of poor testing. Automating chaos just gives faster chaos"*[FG99].

## 2.4 Besonderheiten bei Tests

Im Projekt ESSEX werden numerische, parallele Codes entwickelt. Bei diesen können verschiedene Fehler entstehen, die bei einer "herkömmlichen", sequentiellen Programmierung nicht auftreten.

### 2.4.1 Testen von numerischen Codes

Eine der Besonderheiten bei den numerischen Berechnungen ist, dass es sich häufig um sehr große Dimensionen handelt. Im Projekt ESSEX sind Vektoren mit einer Größe von bis zu $10^{18}$ angedacht. Rechnet man mit Vektoren oder Matrizen dieser Größe,

[24] Ein commit beschreibt bei einem Versionsverwaltungssystem das Hochladen und die Synchronisierung des eigenen Codes mit dem im Repository

ist eine sehr hohe Genauigkeit bei Gleitkommawerten notwendig. Außerdem ist es typisch für numerische Anwendungen, dass mehrere Datentypen verwendet werden. Im Projekt ESSEX gibt es beispielsweise vier verschiedene Datentypen, die beim Testen alle betrachtet werden müssen. Dabei handelt es sich um reelle und komplexe Zahlen in jeweils einfacher und doppelter Genauigkeit.

### 2.4.2 Testen von parallelen Anwendungen

Weitere Probleme können beim parallelen Testen auftreten. Durch die Parallelisierung gibt es verschiedene Phänomene, die im sequentiellen Code nicht auftreten. Dazu zählen zum Beispiel **Race Conditions, Deadlocks** oder **Messeffekte**. Von einer Race Condition spricht man, wenn mindestens zwei Prozesse gleichzeitig auf eine Variable bzw. Speicherstelle zugreifen und mindestens eine davon diese Stelle beschreiben will[RC02]. Der Begriff Deadlock bezeichnet die Verklemmung zweier Prozesse. Dieses ist leicht mittels einer Grafik zu erläutern (siehe *Abbildung 5*). Dadurch, dass die Prozesse ihre zugewiesene Ressource nicht freigeben, bevor sie die neue Ressource belegen können, kommt es an dieser Stelle zur Blockade. Das Programm kann nicht weiter ausgeführt werden.

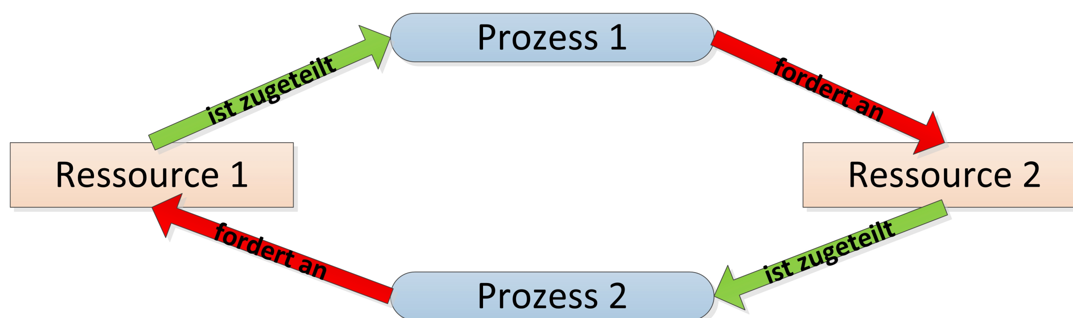



**Abbildung 5:** Einfachster Fall eines Deadlocks

Zusätzlich zu diesen spezifischen Problemen, können bei parallelen Programmen so genannte Messeffekte auftreten[DP02]. Fügt man im parallelen Code Anweisungen

ein, die das Debugging erleichtern sollen (z.B. eine Konsolenausgabe), verändert sich häufig das Timingverhalten[25] des Programms dahingehend, dass die zuvor aufgetretenen Fehler nun nicht mehr beobachtet werden können. Aufgrund der parallelen zu der *Heisenbergschen Unschärferelation*[26] wird dieses Verhalten auch als **Heisenbug** bezeichnet.
Tritt einer der genannten Probleme auf, kommt es zu falschen Ergebnissen oder im schlimmsten Fall zu einem Systemabsturz und zu Datenverlusten.

[25]Timingverhalten bedeutet hier, dass Befehle durch Änderungen (z.B. eine Konsolenausgabe) zu einem (wenn auch nur leicht) späteren Zeitpunkt ausgeführt werden als ohne. Dies allein kann schon dazu führen, dass der Fehler, der zuvor aufgetreten ist, nicht mehr auftritt.

[26]nach der man niemals zur gleichen Zeit zwei Messgrößen eines Quantenteilchens genau bestimmen kann[Gro]

# 3 Entwicklung eines Testkonzepts

In diesem Kapitel soll die Entwicklung des Testkonzepts beschrieben werden. Dabei wird zunächst eine Anforderungsanalyse durchgeführt. Darauf folgend werden die Werkzeuge, die den Testprozess unterstützen bzw. automatisieren sollen, vorgestellt. Im Anschluss werden die Überlegungen für Komponenten-, Integrations- und Systemtests erläutert. Ein Abnahmetest entfällt hier vollständig, da es sich um ein Forschungsprojekt handelt und daher kein Kunde im klassischen Sinne existiert. Zuletzt wird beschrieben, wie die Software in die Continuous Integration eingebunden wird. Das vorläufige Testkonzept, welches an die Standards des IEEE 829 angelehnt ist, befindet sich im Anhang.

## 3.1 Anforderungen an das Testkonzept

In diesem Kapitel werden die Anforderungen an das Testkonzept erläutert. Dabei müssen die verschiedenen Besonderheiten aus Kapitel 2.4 beachtet und auf das Projekt ESSEX übertragen werden. Die praktische Anwendung des Konzepts im Projekt steht hier im Vordergrund.
Vier Eigenschaften für die Software aus ESSEX müssen besonders betrachtet werden:

- Nebenläufigkeit
- Numerische Codes
- Fehlertoleranz
- Performance

### 3.1.1 Nebenläufigkeit

Ein besonders wichtiger Aspekt ist die Überprüfung der Nebenläufigkeit. Hierbei werden besonders die Phänomene *Deadlocks* und *Race Conditions* betrachtet (vgl. Kapitel 2.4.2). Beim Auftreten einer dieser beiden Phänomene, besteht eine hohe Wahrscheinlichkeit, dass das Ergebnis nicht mehr korrekt ist. Eine besondere Herausforderung dabei ist, dass es sich um Algorithmen handelt, die gleichzeitig in mehreren Tausend bis einer Million Prozessen berechnet werden sollen. Bei einer Software, die nicht aus dem Bereich des High Performance Computing stammt, werden Berechnungen heutzutage auch häufig in mehreren Prozessen beziehungsweise Threads durchgeführt, allerdings handelt es sich dabei um eine deutlich geringere Anzahl von nebenläufigen Ausführungen. Das Problem beim Testen dieser hoch parallelen Softwareprodukte ist, dass die Ausführung nie deterministisch sein kann. Durch verschiedene Faktoren ist es möglich, dass Befehle, die für einen geregelten Softwareablauf später ausgeführt werden müssten bereits früher zur Ausführung kommen. *Race Conditions* und *Deadlocks* sind in der Realität häufig nicht reproduzierbar. Verschiedene Tools sollen dabei helfen, kritische Programmteile zu identifizieren. Dabei werden häufig statische Code-Analysen durchgeführt, um zu erkennen, ob an einer bestimmten Stelle ein *Deadlock* entstehen könnte.
Durch diese Analysen lassen sich niemals alle Probleme der Nebenläufigkeit erkennen und beheben, allerdings sollen diese im Testkonzept beachtet werden, damit grobe und häufige Fehler vermieden werden können.
Es muss zum einen die Parallelisierung auf Node-Level und zum anderen die korrekte MPI-Funktionalität (vgl. Kapitel 2.1.2) getestet werden. Beim einsetzen dieser Schnittstellen zur parallelen Programmierung ist es ebenfalls sinnvoll, ein Werkzeug, welches die korrekte Verwendung anzeigt, zu nutzen.

### 3.1.2 Numerische Codes

Die Probleme bei den numerischen Codes, die im Testkonzept beachtet werden müssen, sind zum einen Ungenauigkeiten bei Gleitkommaoperationen[27] und zum anderen die verschiedenen Datentypen, die in der Software verwendet werden.
Das Testen der einzelnen Datentypen ist keine Herausforderung, allerdings erhöht sich der Aufwand zur Testerstellung durch die Verwendung dieser enorm. Da die erstellten Algorithmen jeweils vier Datentypen unterstützen sollen, muss der Testentwickler für jeden dieser Datentypen einen einzelnen Test entwickeln. Abhilfe kann hier ein Werkzeug schaffen, welches es erlaubt, alle Testfälle für jeden einzelnen Datentyp automatisch durchzuführen. Dies verringert zwar nicht die Kosten für die Ausführung der Tests, die Kosten für die Testerstellung werden jedoch erheblich reduziert. Um ein aussagekräftiges Testergebnis zu erhalten, muss ein Schwellenwert für Rundungsungenauigkeiten festgelegt werden. Dieser darf nicht zu klein sein, da ansonsten Fehlerwirkungen identifiziert werden, die lediglich durch die Verwendung von Gleitkommawerten entstehen. Allerdings darf dieser Schwellenwert auch nicht zu groß sein, da ansonsten Fehlerwirkungen nicht erkannt werden. Der Wert muss an den Bedarf der Wissenschaftler, die dieseAlgorithmen nutzen sollen, angepasst sein.

### 3.1.3 Fehlertoleranz

Im Projekt ESSEX soll ein System zur Fehlertoleranz (engl. *fault tolerance*) zusätzlich die verwendeten Ressourcen entlasten. Dieses Fehlertoleranz-System wird im Zuge der Bibliothek Ghost an der Universität Erlangen entwickelt. Dieses soll die Robustheit des Systems erhöhen, weil es dadurch ermöglicht wird, korrekte Rechenergebnisse trotz des Ausfalls eines Prozesses zu erhalten.
Kommt es während der Ausführung beispielsweise zu einem Ausfall eines Knotens[28],

[27] z.B. durch Rundungen
[28] z.B. durch einen Stromausfall

kann die Berechnung nicht weitergeführt werden. Die Ausführung hält an und das Ergebnis bzw. Zwischenergebnis wird unbrauchbar. Um dies zu vermeiden soll ein *Checkpoint*-System implementiert werden. Dabei sollen Zwischenergebnisse temporär gesichert werden, um die Berechnung nach dem Auftreten eines solchen Fehlers fortsetzen zu können. Dadurch muss die Berechnung nicht vollständig neu gestartet werden und die Ressourcen werden schneller wieder für andere Operationen frei.
Dieses Verfahren ist jedoch eher unüblich im HPC-Bereich und wird im Rahmen des Projekts von den Entwicklern selbst implementiert. Deshalb ist es wichtig, dieses System ebenfalls ausgiebig zu testen. Zu diesem Zweck muss eine Testumgebung geschaffen werden, mit der es möglich ist, einzelne Nodes bzw. Prozesse zwischenzeitlich abzuschalten. Im Falle, dass das System funktioniert, müsste mit einer kurzen Verzögerung dennoch das korrekte Ergebnis berechnet werden. Funktioniert es nicht kommt es zu Fehlermeldungen, enormen Performance-Einbußen oder falschen Ergebnissen.
Parallelen zu diesem Anwendungsfall gibt es zum Testen von verteilten Systemen. Für Client-Server-Architekturen oder verteilte Datenbanken beispielsweise existieren verschiedene Testumgebungen, mit denen sich das Verhalten beim Ausfall einer Hardwarekomponente simulieren lässt[Kro13]. Allerdings konnte bei einer Recherche festgestellt werden, dass kein ausgereiftes Framework entwickelt wurde, welches diese Funktionen für MPI-basierte Applikationen liefert.
Es gibt drei Möglichkeiten einen Hardwareausfall manuell zu simulieren beziehungsweise zu erzeugen. Bei der ersten wird die Software auf einem System ähnlich dem Zielsystem, also einem Rechner mit mehreren Nodes, ausgeführt. Während eines umfangreichen Berechnungsprozesses wird die Stromversorgung einer oder mehrerer Nodes unterbrochen. Bei einem Vergleich der Ergebnisse mit denen aus einem störfallfreien Testlauf lassen sich Aussagen über die Korrektheit des Fehlertoleranzsystem und über die Performance im Falle einer Störung tätigen. Der Vorteil dieser Methode ist, dass das hier analysierte Verhalten nahe dem auf dem "realen" Zielsystem ist. Eine Automatisierung wäre beispielsweise mit Hilfe von elektronischen Automatisierungstechniken möglich,

jedoch sehr aufwendig und wartungsintensiv. Für eine stichprobenhafte Überprüfung reicht dieses Verfahren dennoch aus.
Die zweite Möglichkeit ist, die Nodes mittels virtueller Maschinen (VM) zu simulieren. Dabei werden auf einem Computer mehrere dieser virtuellen Maschinen, die jeweils eine Node repräsentieren, ausgeführt. Die MPI-Kommunikation geschieht zwischen diesen "Nodes". Analog zum ersten Verfahren wird nun die Ausführung einer oder mehrerer VMs plötzlich abgebrochen. Obwohl dieses Verfahren nicht so nah am Zielsystem ist, wie das vorherige, sollten dennoch ähnliche Ergebnisse erzielt werden. Die Automatisierung ist in diesem Fall erheblich einfacher und lässt sich beispielsweise durch ein Shell-Script durchführen. Auch eine eigene Testumgebung ließe sich so erzeugen, allerdings muss hier der Kosten-Nutzen-Faktor beachtet werden. Bei diesem Verfahren besteht sogar die Möglichkeit der Einbindung in die Continuous Integration.
Abschließend lässt sich sagen, dass beide Varianten im Testkonzept Beachtung finden sollten. Das Verfahren mit den virtuellen Maschinen ist nach einer angemessenen Automatisierung sogar für den häufigen Gebrauch geeignet. Zusätzlich sollte nach großen Änderungen oder stichprobenhaft das erste Verfahren durchgeführt werden, um das Vertrauen in die Software weiter zu erhöhen.
Die dritte Möglichkeit ist angelehnt an die erwähnten Parallelen zum Testen von Client-Server-Architekturen. Im Rahmen einer Masterarbeit[Kro13] wurde hier im DLR eine Test-Infrastruktur zum Testen dieser erstellt. Dazu wird ein Cloud-Computer-Cluster benötigt. Dieses wird beispielsweise von Amazon[ama13b] bereitgestellt. Die *Amazon Elastic Compute Cloud*[ama13a] (**Amazon EC2!**) soll Entwicklern die Cloud-Programmierung erleichtern. Die Rechenressourcen lassen sich je nach Bedarf anpassen. Auf diesem Computer-Cluster lässt sich dann die MPI-Applikation installieren. Mittels eines Skripts lässt sich dann wiederum ein Hardwareausfall simulieren. Die Auswertung der Ergebnisse erfolgt analog zu den vorher erläuterten Verfahren.
Vorteil hier ist, dass interne Ressourcen gespart werden. Beim zweiten Verfahren wäre ein leistungsstarker Computer bzw. ein eigenes Computer-Cluster nötig, um die VMs zu

betreiben. Nachteil ist zum einen, dass zusätzliche Kosten entstehen können. Erst bei einem Testlauf könnte man diese abschätzen, da die Abrechnung einer On-Demand-Instanz stundenweise geschieht. Zum anderen ist nicht klar, ob eine ausreichende Datensicherheit gewährleistet ist.

### 3.1.4 Performance

Eine weitere wichtige nicht funktionale Anforderung ist die Performance, wie in Kapitel 2.3.2 beschrieben. Diese muss ausreichend überprüft werden, da es wichtig ist, dass die Ressourcen schnell freigegeben werden können. Viele Betreiber von Supercomputern verlangen einen Beweis der Performance, sodass sichergestellt ist, dass die Ressourcen effizient genutzt werden.

Zunächst war für diese Arbeit angedacht, dass verschiedene Tools zu Performance-Test-Zwecken überprüft und ausgewertet werden sollen. In der Aufgabenstellung vorgegeben waren hier Tools wie VAMPIR[Gmb], SCALASCA[Jül] oder Likwid[lik13]. Nach einer eingehenden Diskussion waren sich die Entwickler des DLR und des Rechenzentrums Erlangen einig, dass eine Automatisierung des Performance-Test unter Zuhilfenahme dieser Tools nicht besonders sinnvoll ist, da die Verwendung dieser stark von der Expertise des jeweiligen Entwicklers abhängt. Häufig verwenden diese Tools eine graphische Oberfläche und stellen somit ihre Ergebnisse in einem für Computer nur schwer leserlichem Format dar. Die Automatisierung dieser Tools führt zwangsläufig zu einer Filterung der Informationen, die von diesen erzeugt werden.

Des Weiteren muss die Ausführung der Performance-Tests von einem Entwickler bzw. Tester durchgeführt werden, der mit der Hardware des Systems und den Implementationsdetails vertraut ist. Teile der Software können je nach Anwendungsfall sehr unterschiedliche Ergebnisse auf verschiedenen Rechenplattformen liefern.

Entwickler des DLR und der Universität Erlangen waren sich bei einem Testplanungs-Gespräch einig, dass es sinnvoll ist, den automatisierten Performance-Test auf ein

Minimum zu reduzieren. Dazu soll lediglich die Ausführungszeit von performancekritischen Funktionalitäten auf einer einzelnen Node gemessen und analysiert werden. Gibt es bei diesem Wert nach einer Änderung im Programmcode erhebliche Schwankungen, weiß der Entwickler, dass irgendeine dieser Änderungen zu Performance-Einbußen geführt hat. Dieser hat dann immer noch die Möglichkeit das Tool seiner Wahl manuell zu benutzen, um beispielsweise Flaschenhälse zu identifizieren. Sichert man die gemessene Performance und stellt diese graphisch dar, kann man auch leicht feststellen, wie sie sich im Laufe der Entwicklung verändert hat. Auf dem Jenkins-Server soll dieser Graph dargestellt werden.
Es ist wichtig, die Performancetests mit verschiedenen Konfigurationen (mit einer verschiedenen Anzahl von Prozessen) durchzuführen. Vergleicht man diese verschiedenen Konfigurationen hinsichtlich ihrer Laufzeiten, können Aussagen über die Skalierbarkeit der Software getätigt werden. Sollte diese nicht angemessen sein, d.h. wächst die Performance nicht in Relation zur Anzahl der Prozesse[29], muss die Software überarbeitet werden. Falls es bei der Skalierbarkeit schon auf kleinen Systemen zu Problemen kommt, werden diese garantiert auf dem Zielsystem ebenfalls auftreten. Eine schlechte Skalierbarkeit ist für einen Algorithmus aus dem HPC-Bereich nicht akzeptabel.

## 3.2 Verwendung von Tools

Um den Testprozess zu vereinfachen und übersichtlicher zu gestalten, sollen verschiedene Tools zur Unterstützung genutzt werden. Daraus ergeben sich mehrere Vorteile. Zum einen erleichtern die Werkzeuge das Erstellen und die Wartbarkeit von Testfällen. Zum anderen wird die Auswertung der Tests erheblich erleichtert. Es werden Tools zur Testfallerstellung, zur dynamischen Analyse und für die Performanceanalyse genutzt.

[29] Im Idealfall verdoppelt sich die Performance bei einer doppelten Anzahl von parallelen Prozessen.

### 3.2.1 Google C++ Testing Framework

Als fundamentales Werkzeug zur Testerstellung wird das *Google C++ Testing Framework*[goo] genutzt. Dieses Framework abstrahiert die Testerstellung auf ein höheres Niveau und ermöglicht dadurch eine schnellere und übersichtlichere Implementierung von Tests. Des Weiteren liefert Google Test viele Möglichkeiten, Tests individuell anzupassen. So lassen sich zum Beispiel eigene Fehlermeldungen bei einem fehlgeschlagenen Test einbauen.
Verschiedene Tests lassen sich übersichtlich in Testfälle gliedern. Dadurch repräsentieren die Testfälle das Aussehen des eigentlichen Programmcodes. Dies hilft sowohl den Testern als auch den Entwicklern bei der Arbeit. Die Ausführung ist außerdem schnell und eine große Community nutzt das Google C++ Testing Framework, sodass bei auftretenden Problemen Unterstützung garantiert werden kann. Ferner besitzt das Framework eine gute Dokumentation, die sowohl für Anfänger als auch für fortgeschrittene Nutzer Hilfestellungen bietet. Außerdem werden in dieser Dokumentation die Befehle, die zum Erstellen von Tests benötigt werden, ausführlich und anhand von Beispielen erklärt. Google Test ist auf allen gängigen Betriebssystemen lauffähig. Die Ausgabe, die Google Test liefert, ist sehr übersichtlich und leicht verständlich. Da die Benutzung sehr einfach ist, lassen sich die Tests auch automatisieren und in die Continuous Integration eingliedern. Dieses Werkzeug lässt sich einsetzen, um Tests auf Komponenten- und Integrationsebene zu schreiben.

### 3.2.2 Valgrind

Ein Werkzeug, das zur Analyse der Nebenläufigkeit der parallelen Software auf Node-Ebene ausgewählt wurde, ist *Valgrind*[vala]. Dabei handelt es sich um ein Framework zur dynamischen Analyse von Software. Valgrind ist modular aufgebaut, besteht also aus einer Sammlung von Tools. In dieser Arbeit wird *Helgrind* genutzt. Mit Helgrind ist

es möglich, Race Conditions und Deadlocks (vgl. Kapitel 2.4.2) zu erkennen. Helgrind überprüft dabei, ob Speicherstellen im gemeinsamen Speicher von mehreren Threads gleichzeitig angesprochen werden[valb] (siehe *Abbildung 6*).

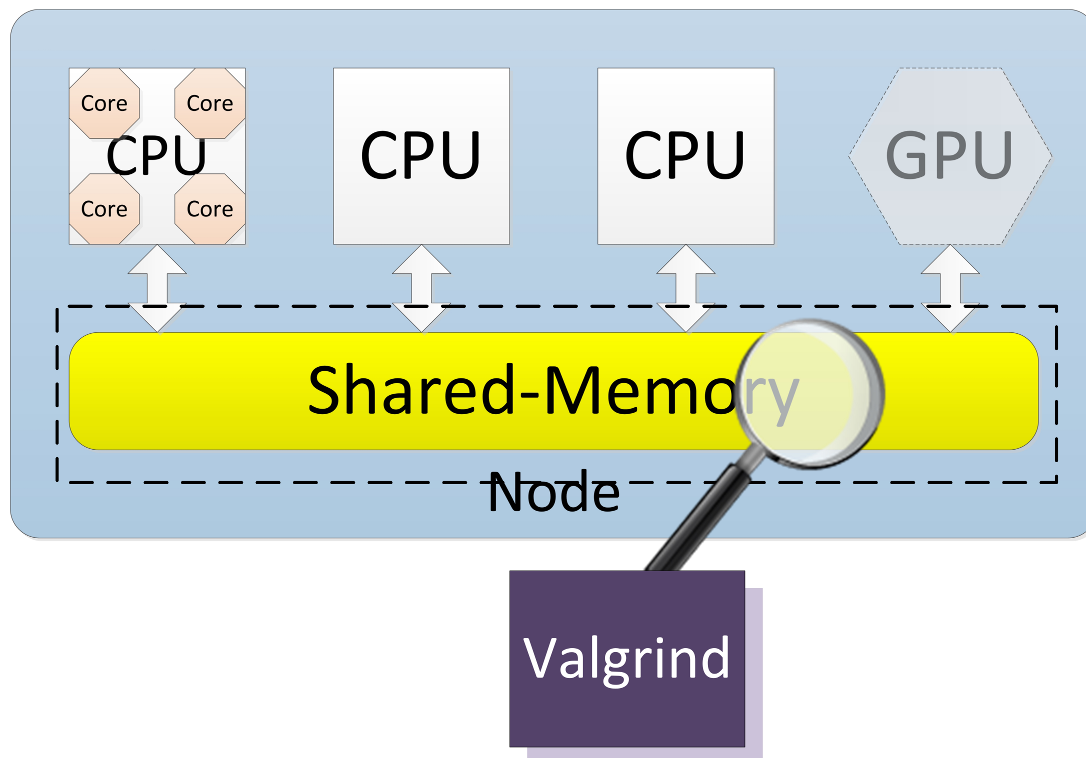



**Abbildung 6:** Valgrind überwacht den gemeinsamen Speicher einer Node

Zusätzlich überprüft Helgrind, ob die Lock[30]-Reihenfolge im gesamten Programm konsistent ist. Ist dies nicht der Fall, können dadurch Deadlocks entstehen. Das Auftreten eines Deadlocks ist stark abhängig vom Timingverhalten des Programms und nicht reproduzierbar. Dadurch, dass nicht speziell Deadlocks, sondern Inkonsistenzen beim Locking erkannt werden, können alle Stellen, an deinen ein Deadlock theoretisch entstehen könnte, identifiziert werden.

Ein weiterer Vorteil ist, dass eine falsche Verwendung der Threading-Befehle ebenfalls erkannt wird. Helgrind ist ein experimentelles Tool, welches jedoch brauchbare Ergebnisse

[30] Das Sperren einer Ressource

liefert. Diese sind in Kapitel 4.3.1 näher beschrieben. Die Valgrind-Tool-Suite lässt sich ebenfalls ohne Probleme automatisieren.

### 3.2.3 Tool zum Testen von MPI-Anwendungen

Um die korrekte Verwendung von MPI zu überprüfen, sollen ebenfalls Werkzeuge eingesetzt werden. Ein solches Werkzeug soll es ermöglichen, eine inkorrekte Nutzung von MPI und Deadlocks zu identifizieren. Ein mögliches Tool um dies zu realisieren ist MUST[Dre]. Es wird seit einigen Jahren von der Technischen Universität Dresden entwickelt. Es kann verschiedene Arten von inkorrekter MPI-Nutzung und auch mögliche Deadlocks, die durch die Verwendung von MPI entstehen, erkennen. Um MUST nutzen zu können müssen zunächst zwei weitere Tools, die die Basis-Infrastruktur für die Verwendung von MUST bereitstellen, installiert werden.
Die Benutzung des Tools ist sehr einfach. Anstelle des normalerweise verwendeten Befehls *mpirun* bzw. *mpiexec* wird hier der Befehl *mustrun* ausgeführt. *mustrun* stellt dabei die selben Optionen zur Verfügung wie *mpiexec*. So kann man mit der Option *-np* die Anzahl der MPI-Prozesse angeben. Ausgeführt wird jedoch immer ein Prozess mehr als angegeben. Dieser wird von MUST benötigt, um die MPI-Kommunikation abzufangen und auszuwerten (vgl. *Abbildung 7*).
Das Ergebnis, das MUST liefert ist im HTML-Format. Ein Vorteil von diesem Format ist, dass es statisch analysierbar ist. Dadurch lässt sich nicht nur die Ausführung des Werkzeug, sondern auch die Auswertung der Ausgabe automatisieren. Bedarfsweise kann die Ausgabedatei auch mit Hilfe eines Browsers visualisiert werden. Ein Beispiel für diese Ausgabe befindet sich in *Abbildung 8*. In diesem wurde ein möglicher Deadlock durch eine inkonsistente Nutzung von MPI gefunden und dargestellt.
Das Problem bei der Verwendung von MUST ist, dass eine Installation aufgrund der Abhängigkeiten sehr umständlich ist. Es ist schwer vorherzusagen, ob sich dies auf jedem System realisieren lässt. Zumindest im DLR müsste es für eine regelmäßige

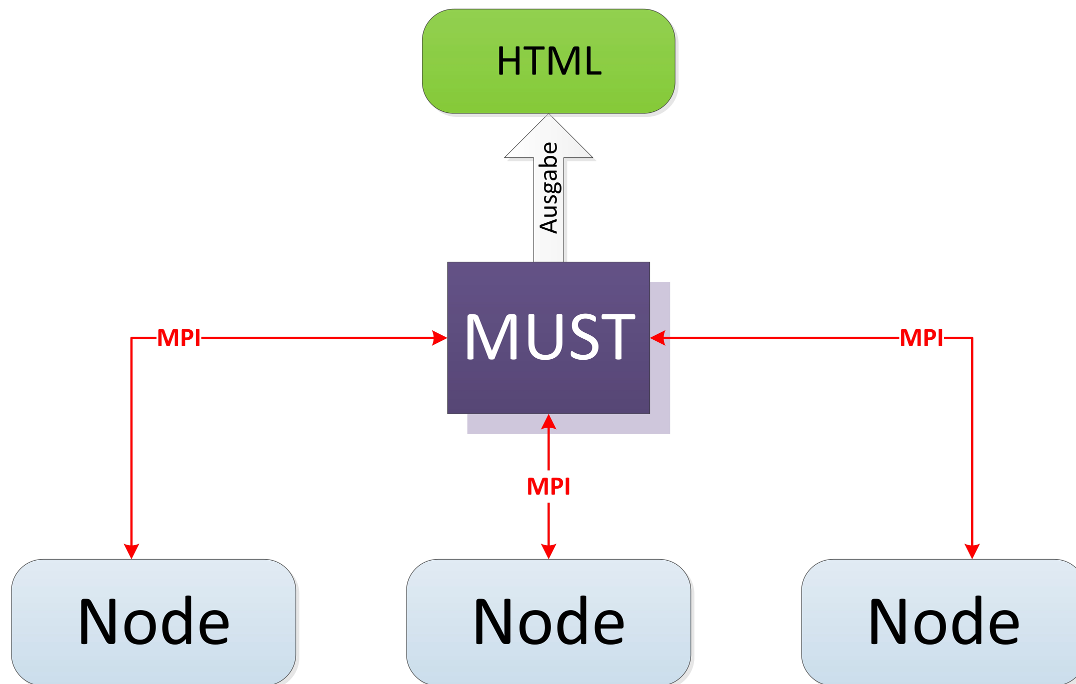



**Abbildung 7:** MUST analysiert die MPI-Kommunikation

und effiziente Anwendung im Modulsystem aufgenommen werden, sodass das Tool von jedem Entwickler ohne viel Aufwand genutzt werden kann. Durch die zeitliche Beschränkung dieser Arbeit, konnte das Tool nicht mehr exemplarisch installiert und überprüft werden. Die weitere Evaluation steht also noch aus. Dennoch ist es theoretisch ein brauchbares Tool und sollte im Testprozess genutzt werden.

## 3.3 Grundsätzliche Vorgaben im Testkonzept

Dieses Kapitel erläutert die Grundlagen des Testkonzepts. Dabei handelt es sich um testebenenübergreifende Methoden, die zur Qualitätssicherung des Softwareprodukts beitragen sollen.

Zunächst wird hier beschrieben, wie und welche Arten von statischen Tests den Software-Entwicklungsprozess unterstützen sollen.

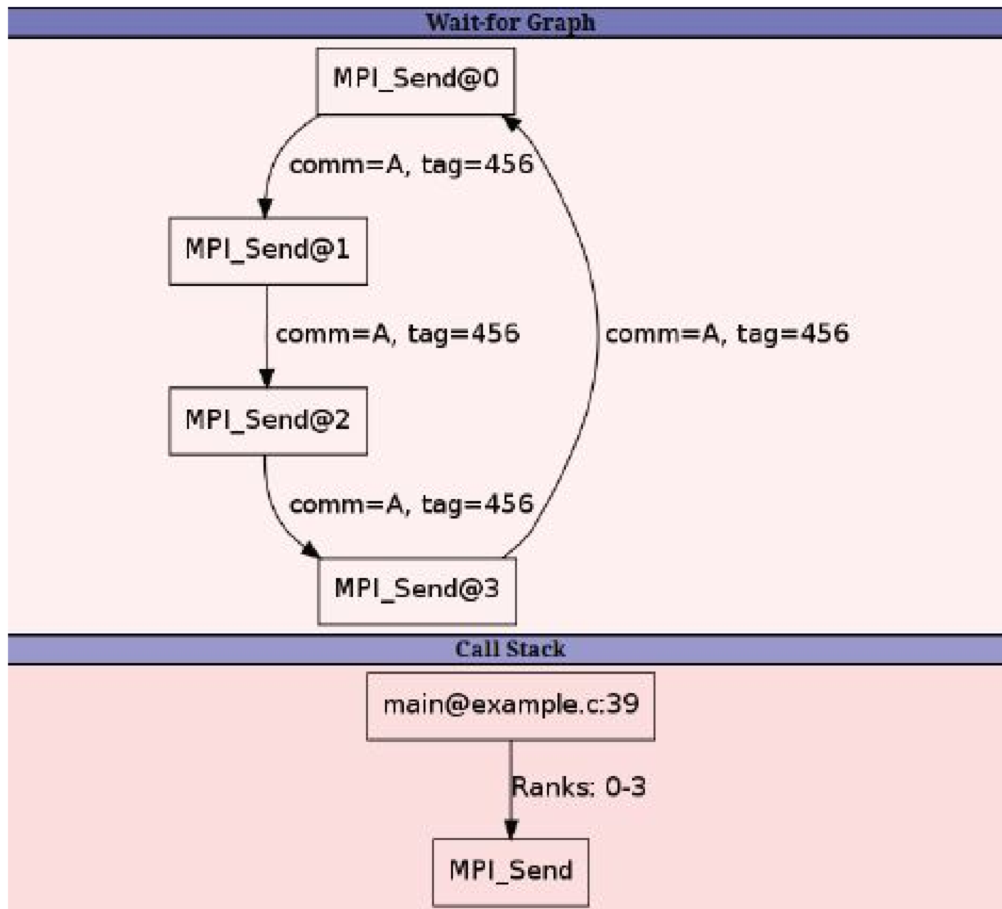



**Abbildung 8:** Beispiel: Ausgabedatei von MUST

Damit der Programmcode ausreichend überprüft wird, genügt es im Abstand von einigen Wochen Code-Reviews durchzuführen. Dazu müssen sich die Entwickler der verschiedenen Forschungseinrichtungen, die am Projekt beteiligt sind, an einem Ort einfinden. Treffen dieser Art haben bereits in der Vergangenheit und auch während der Erstellung dieses Testkonzepts stattgefunden. Bei der relativ geringen Anzahl der Entwickler, genügen informelle Review-Arten (Kapitel 2.2.4). Da in der Einrichtung für Simulations- und Softwaretechnik viele Studenten und Praktikanten beschäftigt werden, die teilweise nur wenige Wochen oder Monate am Projekt beteiligt sind, bietet es sich ebenfalls an, die Methoden "buddy testing" oder "pair programming" zu nutzen. Letzteres Verfahren wurde ebenfalls während der Bearbeitung dieser Bachelorarbeit

von zwei Praktikantinnen durchgeführt. Dadurch, dass zwei Entwickler am gleichen Programmcode arbeiten, werden Fehler schneller erkannt. Um eine statische Codeanalyse durchzuführen, wird lediglich der Compiler genutzt. Damit werden syntaktische Fehler vermieden.
Diese Verfahren lassen sich nicht den einzelnen Testebenen zuordnen, sondern müssen für den gesamten Programmcode durchgeführt werden.
Um zunächst zu zeigen, dass die jeweilige Komponente, der Algorithmus oder das System funktionieren, wird zunächst mit relativ kleinen Matrizen und Vektoren gearbeitet. Diese ermöglichen es, Ergebnisse besser zu validieren. Dies ist Teil des Proof of Concept (vgl. Kapitel 2.3.1). Zusätzlich müssen große Matrizen und Vektoren benutzt werden, damit weitere Fehlerzustände entdeckt und behoben werden können. Erst durch die großen Eingabeparameter werden numerische Fehler und Fehler, die durch die Parallelisierung entstehen, provoziert.

## 3.4 Komponententests

Ein gutes Konzept für Komponenten ist die Grundlage für ein erfolgreiches Softwareprojekt. Die höheren Testebenen bauen jeweils aufeinander auf. Beim Unit-Test werden einzelne Funktionen oder Klassen so getestet, dass möglichst alle Fehler erkannt und behoben werden können. Im Projekt ESSEX werden die Grundfunktionen wie z.B. Matrix-Vektor-Multiplikationen auf dieser Ebene getestet. Es bietet sich an, an dieser Stelle eine Kombination aus Blackbox- und Whitebox-Verfahren (siehe Kapitel 2.2.4) zu wählen.
Da im Projekt nur Algorithmen zur Lösung mathematischer Probleme entwickelt werden und diese eindeutige Ein- und Ausgabewerte besitzen, bietet es sich ebenfalls an, Äquivalenzklassentests durchzuführen. Es müssen genügend Äquivalenzklassen abgedeckt werden, damit früh erkannt wird, wo sich Fehlerzustände befinden. Hierbei

treten allerdings zwei Probleme auf: Zum einen handelt es sich bei den Eingabewerten beispielsweise um sehr große Vektoren und Matrizen. Bei diesen ist es nicht möglich, aus den Eingabeparametern das erwartete Ergebnis im Vorhinein zu berechnen. Daher muss das Ergebnis auf einem anderen Wege verifiziert werden. Dies geschieht mit Hilfe der Umkehrrechnung der jeweiligen Berechnung. Dadurch kann ebenfalls gezeigt werden, dass das Programm das korrekte Verhalten zeigt. Zum anderen muss hier auf die Besonderheiten der numerischen Programmierung (Kapitel 2.4.1) geachtet werden. Rundungsungenauigkeiten können hier Fehlerwirkungen erzeugen, die im Code nicht begründet sind. Hier müssen angemessene Toleranzwerte beachtet werden (Kapitel 3.1.2).
Des Weiteren müssen bereits hier die verschiedenen Datentypen beachtet werden.
Es ist an dieser Stelle besonders sinnvoll, die genannten Blackbox-Testverfahren durch weitere Whitebox-Verfahren zu ergänzen. Als Kriterium bietet sich die Zweigüberdeckung (vgl. Kapitel 2.2.4) an. Da es sich um relativ kleine Funktionen handelt, die auch wenig Verzweigungen aufweisen, ist es hier nicht schwer eine Zweigüberdeckung von 100% zu erreichen.

## 3.5 Integrationstests

Bei den Integrationstests sollen nun die Algorithmen getestet werden, welche die Funktionen auf Komponentenebene benutzen. Einer dieser Algorithmen ist beispielsweise das **Gram-Schmidt-Orthogonalisierungsverfahren**. Bei diesem Verfahren wird aus einer Basismatrix und einer Menge vorgegebener Vektoren ein Orthogonalsystem erzeugt, das den selben Untervektorraum erzeugt. Das bedeutet, dass alle Ausgabevektoren zueinander orthogonal sind.
Auch hier besteht wieder nicht die Möglichkeit, die Ausgabewerte direkt auf Korrektheit zu überprüfen. Allerdings lässt sich das Problem umgehen, indem man die

Ausgabevektoren mit den Eingabevektoren bzw. mit der transponierten Ausgabematrix multipliziert. Bei den im Projekt verwendeten Multivektoren ergibt sich in dem Fall, dass der Algorithmus korrekt ist, eine Einheitsmatrix. Unterscheidet sich die Ausgabe von dieser, lässt sich daraus ableiten, dass sich in der Implementation des Algortihmus ein Fehlerzustand befindet. An dieser Stelle wird ebenfalls deutlich, wie wichtig es ist, dass die Komponenten ausreichend getestet werden. Ist dies nicht der Fall, ist nicht ersichtlich, ob der Fehler auf Integrations- oder auf Komponentenebene zu suchen ist. Für die Integrationstests gilt ebenfalls, dass Äquivalenzklassentests hier besonders sinnvoll sind, es ist allerdings schwierig bei den großen Matrizen bzw. Vektoren sinnvolle Äquivalenzklassen abzuleiten.

Im Falle des Projekts ESSEX ist es besonders wichtig, dass zusätzlich immer das erfahrungsbasierte Testen (vgl. Kapitel 2.2.4) angewandt wird. Der Tester muss eine gute Kenntnis über den jeweiligen Algorithmus haben, damit er weitere geeignete Testfälle konstruieren kann. Bei diesen Testfällen handelt es sich zum Beispiel um Matrizen, bei denen Besonderheiten auftreten können, die allein durch einen Äquivalenzklassentest nicht abgedeckt werden können.

Bei den Integrationstests werden ebenfalls Blackbox- und Whitebox-Verfahren verwendet. Auch hier ist eine Zweigüberdeckung von 100% anzustreben. Bei weniger komplexen Algorithmen sollte dies realistisch sein. Bei den umfangreicheren Algorithmen zum Beispiel zur Eigenwertberechnung, ist es sehr schwierig diesen Wert zu erreichen. Wichtiger als die Teststrategie ist an dieser Stelle die Integrationsstrategie. Da die Komponenten an sehr unterschiedlichen Stellen bearbeitet werden und zu unterschiedlichen Zeiten fertiggestellt werden, bietet es sich an, die **Ad-hoc-Integrationsstrategie** zu verwenden. Dadurch wird Zeit gewonnen, da die fertiggestellte Komponente so früh wie möglich in die passende Umgebung integriert wird.

## 3.6 Systemtests

Besonders bei Systemtests unterscheidet man funktionale und nicht funktionale Tests (vgl. Kapitel 2.2.3). Bei den Ersteren ist es wichtig, dass die korrekte Ausführung des Gesamtsystems ausreichend getestet wird. Sind die Integrationstests abgeschlossen, kann mit den Systemtests begonnen werden. Bei Systemtests sind Whitebox-Verfahren nicht mehr möglich. Die Tests beschränken sich lediglich auf das System als Blackbox. Testfälle können nur noch aus den Systemanforderungen abgeleitet werden.
Die Fertigstellung des Gesamtsystems liegt zur Zeit in ferner Zukunft. Es können an dieser Stelle nicht alle Eventualitäten beachtet werden. Sollten im Laufe des Entwicklungsprozesses Abweichungen bezüglich der Software geben, muss das Testkonzept und insbesondere das Konzept für Systemtests angepasst werden.
Zu den funktionalen Anforderungen zählen außerdem das Testen der Fehlertoleranz und der Nebenläufigkeit. Ein nicht funktionaler Systemtest ist beispielsweise der Performancetest. Wie diese drei Kriterien betrachtet und getestet werden sollen, wurde bereits in den Kapiteln 3.1.1, 3.1.3 und 3.1.4 erläutert.

## 3.7 Continuous Integration

Der gesamte build- und Testprozess soll nach Möglichkeit in die Continuous Integration (vgl. Kapitel 2.3.3) eingegliedert werden. Wichtig hierbei ist, dass die Tests im Vorhinein genügend automatisiert werden. Dies lässt sich durch die Verwendung von CMake realisieren. Im CMake file können die Tests angegeben werden, die beim Aufrufen des Befehls *make test* ausgeführt werden sollen. Die Eingliederung soll analog zum DLR-Projekt FreeWake geschehen. Dieses Projekt ist bereits erfolgreich eingegliedert und lässt sich auf dem Jenkins-Server verfolgen. Neben dem build-Status können die Testabdeckung und die Performance visualisiert werden. Letzteres ist in diesem Projekt besonders nützlich, da leicht Performance-Einbrüche bei einer Code-Änderung entdeckt

werden können und der Fehlerzustand schneller lokalisiert werden kann. Ein Beispiel für die Visualisierung der Performance auf dem Jenkins Server befindet sich in *Abbildung 9*. Im Projekt FreeWake wird die Automatisierung des build- und Testprozesses ebenfalls

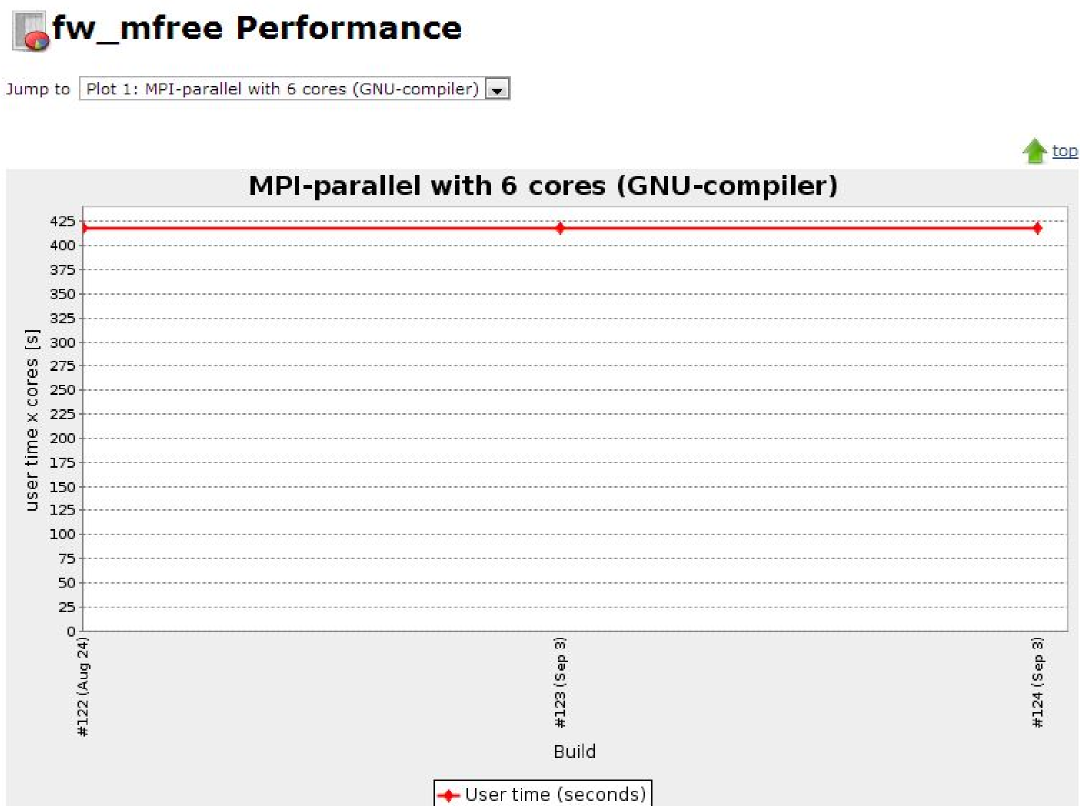



**Abbildung 9:** Performance-Überblick des Projekts FreeWake in der Jenkins Web-API

mit CMake durchgeführt. Daher sollte eine Einbindung vom Projekt ESSEX in die Continuous Integration analog funktionieren und kein Problem darstellen.

# 4 Exemplarische Umsetzung einiger Tests

In diesem Kapitel wird die Umsetzung einiger Tests beschrieben. Dabei handelt es sich zum einen Komponenten- und Integrationstests und zum anderen um Tests, die die Nebenläufigkeit überprüfen sollen.
Es ist unmöglich an dieser Stelle die Tests auf den Systemen durchzuführen, auf denen die Software später ausgeführt werden soll, weil es schwierig ist, diese Ressourcen bereitzustellen. Außerdem existiert kein vorgegebenes Zielsystem. Das Projekt dient dazu, verschiedene Algorithmen für Exascale-Computer zusammenzustellen. Ein einheitliches Zielsystem ist nicht definiert. Daher werden die folgenden Tests zunächst nur auf handelsüblichen Computern ausgeführt. Die beispielhafte Umsetzung gilt hier nur als Machbarkeitsstudie und soll zeigen, dass die verwendeten Tools sich für das Projekt eignen.

## 4.1 Erstellung von Tests mit Hilfe des Google C++ Testing Frameworks

Das Google C++ Testing Framework vereinfacht die Erstellung von Tests für die Programmiersprachen C und C++ (Kapitel 3.2). Ein Beispiel für einen simplen Testfall findet sich in Listing 2.

```
1 TEST_F(CLASSNAME, dot_mvec)
2 {
3   if (typeImplemented_)
4     {
5     for (int j=0;j<nvec_;j++)
6       for (int i=0;i<nloc_*stride_;i+=stride_)
7       {
8       vec2_vp_[j*lda_+i]=mt::one()/st::conj(vec1_vp_[j*lda_+i]);
9       }
10    _ST_* dots = new ST[nPvec_];
11    SUBR(mvec_dot_mvec)(vec1_,vec2_,dots,&ierr_);
12    ASSERT_EQ(0,ierr_);
13
14    _ST_ val = st::one() * (ST)nglob_;
15    ASSERTREALEQ(mt::one(),ArrayEqual(dots,nvec_,1,nvec_,1,val));
16    delete [] dots;,
17    }
18 }
```

**Listing 2:** Beispiel für einen Testfall in Google Test

Es handelt sich dabei um den Test der Funktion `dot_mvec`, die ein Matrix-Vektor-Produkt errechnen soll. Um zu überprüfen, ob der Algorithmus fehlerfrei durchgelaufen ist, wird der Zustand von `ierr` überprüft (vgl. Kapitel 3.4). Daraufhin wird überprüft, ob auch das Ergebnis der Berechnung korrekt ist. Dazu wird der Vektor `vec2_` so konstruiert, dass das Skalarprodukt vom zufälligen Vektor `vec1_` und `vec2_` jeweils die Länge des Vektors ergibt. Daraufhin wird der maximale Fehler dieser Berechnung überprüft. Dies geschieht, indem der Fehler mit 1 addiert wird und das Ergebnis dieser Berechnung mit 1 verglichen wird. Ist der Fehler größer als $10^{-16}$, ist der Test fehlgeschlagen. Anderenfalls ist der Testdurchlauf korrekt.

Jeder Test der mit `TEST_F` bzw. `TEST_P` beginnt, wird automatisch bei der Ausführung

von Google Test erkannt und ausgeführt. Bevor die Tests ausgeführt werden, muss für jede Klasse ein Objekt dieser Klasse in der Methode `setUp()` initialisiert werden. Analog dazu existiert für jedes Testobjekt eine Funktion `tearDown()`, der Destruktor. Wird der Test nun ausgeführt, werden alle Tests, die in dem Testfall beschrieben sind, nacheinander ausgeführt. Wird wie im Beispiel `ASSERT_EQ` statt `EXPECT_EQ` genutzt, wird der Testfall beim Auftreten des ersten Fehlers abgebrochen. Ein Beispiel für die Ausgabe, die Google Test bei einem korrekten Durchlauf für einen Test und für das gesamte Testszenario liefert, findet sich in Listing 3. Dort kann man die einzelnen Funktionen sehen, die getestet werden. Außerdem wird am Ende eines Testszenarios angezeigt, dass Fehler gefunden wurden oder alle Tests fehlerfrei sind.

```
1  [----------] 1 test from ZSdMatTest_6_8
2  [ RUN      ] ZSdMatTest_6_8.get_attributes
3  *** Test ZSdMatTest_6_8.get_attributes starting.
4  data type:  complex double
5  *** Test ZSdMatTest_6_8.get_attributes ending.
6  [       OK ] ZSdMatTest_6_8.get_attributes (0 ms)
7  [----------] 1 test from ZSdMatTest_6_8 (0 ms total)
8
9  [----------] Global test environment tear-down
10 [==========] 132 tests from 51 test cases ran. (149 ms total)
11 [  PASSED  ] 132 tests.
```

**Listing 3:** Besipiel für die Ausgabe von Google Test bei einem fehlerfreien Testdurchlauf

Des Weiteren ist es mit Google Test möglich, die Laufzeit der Funktionen zu überprüfen. Grundsätzlich wird bei jedem Testlauf mit Google Test die Laufzeit gemessen. Um die Performance der Funktionen zu messen wurden weitere Tests erstellt. Exemplarisch wurde dazu eine Funktion zur Matrix-Vektor-Multiplikation überprüft. Der Performance-Test wird mit verschiedenen Größen von Matrizen und Vektoren automatisiert durchgeführt. Die dabei gemessene Laufzeit kann daraufhin auf dem Jenkins-Server automatisiert

ausgewertet und visualisiert werden. Zunächst wurden die Benchmark-Tests in einer Debug-Version durchgeführt. Daher ergeben sich relativ lange Berechnungszeiten. Um ein realistisches Ergebnis zu erhalten, müssen diese Tests zukünftig in einer Release-Version durchgeführt werden. Dabei sollten die gemessenen Laufzeiten deutlich geringer sein. Ein Beispiel für die Ausgabe der gemessenen Laufzeit findet sich in Listing 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="8" failures="0"
  disabled="0" errors="0" time="7.923" name="AllTests">
  <testsuite name="C_BENCH_MHD1280B_4" tests="2" failures="0"
      disabled="0" errors="0" time="3.726">
      <testcase name="read_matrix" status="run" time="0.084"
        classname="C_BENCH_MHD1280B_4" />
      <testcase name="times_mvec" status="run" time="3.642"
        classname="C_BENCH_MHD1280B_4" />
  </testsuite>
[...]
```

**Listing 4:** Google Test Ausgabe für einen Performancetest

## 4.2 Testfallerstellung für die Haupt-Orthogonalisierung-Routine im Projekt

An dieser Stelle wird beschrieben, wie sich Testfälle aus den Anforderungen an eine Funktion ableiten lassen. Als Beispiel wird dazu die Haupt-Orthogonalisierungs-Routine des Projekts verwendet. Für die Erstellung von Testfällen ist es für den Tester zunächst nicht notwendig zu wissen, welcher Algorithmus genutzt wird. Für die Erstellung von erfahrungsbasierten Testfällen ist es allerdings sinnvoll, dass der Tester diese Information bekommt. Im Beispiel wird der Gram-Schmidt-Algorithmus zur Orthogonalisierung verwendet. Die Tests fallen in die Kategorie der Integrationstests (vgl. Kapitel 3.5).

Um geeignete Testfälle zu erstellen, werden zunächst Äquivalenzklassen (vgl. Kapitel 2.2.4) aus den Anforderungen an den Algorithmus abgeleitet. Diese ergeben sich aus den Anforderungen an den Algorithmus. In diesem Fall ist die Erstellung von geeigneten Testfällen nicht sehr umfangreich, da bis auf eine Ausnahme jeweils nur eine gültige Äquivalenzklasse für die jeweiligen Eingabeparameter existieren. Zusätzlich gibt es für jede Eingabe keine bis zwei ungültige Äquivalenzklassen. Aus diesen Werten ergeben sich zunächst zwei gültige und acht ungültige Testfälle, da für die gültigen Testfälle für jeden Eingabeparameter nur die gültigen Äquivalenzklassen entscheidend sind und für die ungültigen Testfälle jeweils nur ein Parameter mit einem Repräsentanten aus einer ungültigen Äquivalenzklasse mit gültigen Werten für die anderen Parameter kombiniert wird. Je ein Beispiel für einen gültigen und einen ungültigen Testfall befinden sich in Tabelle 2 und 3. Die Komponentenspezifikation und alle mit dem Äquivalenzklassenverfahren aufgestellten Testfälle befinden sich in Anhang A.2.

| **Nr.** | **V** | **W** | **num_sweeps** | **ierr** | **R1/R2** |
|---|---|---|---|---|---|
| **1** | $V^{100\times 20} \wedge V^T V = E$ | $W^{100\times 4} \wedge l.u.$ | 3 | $\surd$ | $\surd$ |

**Tabelle 2:** Beispiel eines gültigen Testfalls für das Gram-Schmidtsche Orthogonalisierungsverfahren
l.u. bedeutet linear unabhängig

| **Nr.** | **V** | **W** | **num_sweeps** | **ierr** | **R1/R2** |
|---|---|---|---|---|---|
| **3** | $V^{100\times 20} \wedge V^T V \neq E$ | $W^{100\times 4} \wedge l.u.$ | 3 | $\surd$ | $\surd$ |

**Tabelle 3:** Beispiel eines ungültigen Testfalls für das Gram-Schmidtsche Orthogonalisierungsverfahren

Es handelt sich bei diesen Testfällen nur bedingt um konkrete Testfälle, da die Matrizen zunächst noch konstruiert bzw. generiert werden müssen. Außerdem sind diese großen Matrizen hier nicht darstellbar, daher werden sie durch die Platzhalter repräsentiert. Diese Testfälle sollten aber noch durch erfahrungsbasiertes Testen erweitert werden. In diesem konkreten Fall, werden dabei Testmatrizen konstruiert, bei denen der Test fehlschlagen kann. Im Falle dieses Algorithmus wurde bereits eine Matrix gefunden, bei

der es zu Schwierigkeiten in der Berechnung kommt. Diese Matrix wurde so konstruiert, dass die Singulärwerte immer weiter gegen 0 gehen. Durch Rundungsungenauigkeiten entstehen dann während des Gram-Schmidt-Prozesses lineare Abhängigkeiten, wodurch die Berechnung nicht möglich ist. Dies lässt sich gut als Testfall umsetzen.
Des Weiteren sollten die Testfälle durch Whitebox-Tests ergänzt werden. Es soll eine Zweigüberdeckung von 100% angestrebt werden.

## 4.3 Nebenläufigkeitstests

In diesem Kapitel wird die exemplarische Durchführung der Nebenläufigkeitstests beschrieben. Zunächst wird die Parallelisierung auf Node-Ebene mittels Valgrind getestet. Daraufhin wird getestet, ob die MPI-Funktionen fehlerfrei sind, oder MPI nicht korrekt verwendet wurde bzw. Deadlocks durch die Verwendung von MPI entstehen. Dies soll durch das Tool MUST vereinfacht werden.

### 4.3.1 Auf Node-Ebene mittels Valgrind

Der Test auf Nebenläufigkeit wurde beispielhaft mit der Valgrind-Tool-Suite durchgeführt. Verwendet wurde die Version 3.5.0, die auf dem Modulsystem für Linuxrechner im DLR bereits vorhanden war.
Um zu überprüfen, ob Valgrind sich für die Verwendung im Projekt ESSEX eignet, wurde ein Beispielprogramm (`main_task_model`) verwendet. In diesem Beispiel wird zunächst ein zweidimensionales Array mit **n** Zeilen und **k** Spalten erzeugt und mit Nullen gefüllt. Daraufhin werden **k** Threads erzeugt von denen jeder eine Spalte zuge-

wiesen bekommt. Mit folgendem Algorithmus werden nun die einzelnen Zeilen berechnet:

$$x_{i+1} = \begin{cases} x_i + 1 & für \quad x \quad ungerade \land x > 1 \\ \frac{1}{2} x_i & für \quad x \quad gerade \land x > 1 \\ rand(0, 10000) & für \quad x = 0 \lor x = 1 \end{cases}$$

Die Bearbeitung einer Spalte wird abgebrochen, sobald für $x_i$ der Wert 42 auftritt. Der zuständige Thread meldet dann an die anderen, dass er die Abarbeitung gestoppt hat. Dieses Beispiel soll die Verwendung des *task buffers* und der *queue* (vgl. Kapitel 2.1.2) demonstrieren. Die Threads kommunizieren miteinander, um die eingesetzten Grundoperationen zu bündeln und die Performance deutlich zu erhöhen. Im Beispiel sind die Grundoperationen sehr einfach gehalten, in der Realität handelt es sich dabei unter anderem um Matrix-Vektor-Operationen.

Um dieses Programm nun mit Helgrind auf Race Conditions und Deadlocks zu überprüfen, wird folgender Befehl verwendet:

```
valgrind --tool=helgrind --log-file=helgrind.log ./main_task_model
```

Das Ergebnis wird in der Datei *helgrind.log* gespeichert. Die Auswertung der log-Datei ergab jeweils eine unterschiedliche Anzahl von Fehlern. Alle Testläufe hatten allerdings gemeinsam, dass mehrere hundert Fehler erkannt wurden. Diese können allerdings nicht nur vom getesteten Programm selbst stammen, da dieses relativ klein ist. Eine genauere Analyse der log-Datei und eine Recherche ergab, dass das Tool, viele so genannter *false positives*[31] ausgibt. Dies ist bei der Verwendung der Valgrind-Tool-Suite ein bekanntes Problem. Diese *false positives* entstehen, weil Helgrind nicht nur das ausgewählte Programm auf Wettlaufsituationen überprüft, sondern auch jede Bibliothek, zu der eine Abhängigkeit im Programm besteht. Dadurch werden Wettläufe in Standard-C und C++ -Bibliotheken ebenfalls erkannt. In einem Testlauf wurden einige der Meldungen

[31] false positive bedeutet, dass kein Fehler vorhanden ist, aber dennoch einer erkannt wird.

auch durch die OpenMP- und Google Test-Bibliotheken erzeugt. Diese sind allerdings nicht relevant für die Entwicklung der Software. Um dies zu verhindern, gibt es die Möglichkeit, Fehlerbenachrichtigungen zu unterdrücken. Dies geschieht mittels des so genannten *suppression file*, welches mittels des Zusatzes

```
--suppression-file=<filename>
```

genutzt werden kann. Wiederholt man diesen Befehl, können bis zu 100 suppression files gleichzeitig genutzt werden. In dieser Datei können mehrere Regeln für die Unterdrückung von Meldungen definiert werden. Eine Regel befindet sich dabei immer zwischen zwei geschweiften Klammern. In der ersten Zeile steht der Name der suppression. In der folgenden Zeile werden die Tools angegeben, für die diese suppression gültig ist. Außerdem wird die Art des Fehlers angegeben. Alle nachfolgenden Zeilen enthalten Funktions- oder Objektnamen, in der Reihenfolge, in der sie aufgerufen werden. Nur eine Meldung, die diese Kriterien erfüllt, wird unterdrückt. Zusätzlich kann eine suppression mittels *Wildcards* ergänzt werden. Die so genannte *frame-level wildcard* ('...') ermöglicht es, Bibliotheken, in denen häufig Race Conditions erkannt werden, in einer suppression zusammenzufassen.

```
1 {
2   <insert_a_suppression_name_here>
3   Helgrind:Race
4   ...
5   fun:gomp_barrier_wait_end
6   ...
7   fun:gomp_team_start
8   fun:main
9 }
```

**Listing 5:** Beispiel einer suppression

In Quellcode 5 gibt die frame-level wildcard an, dass zwischen der Funktion *gomp_team_start* und *gomp_barrier_wait_end* weitere Funktionsaufrufe vorhanden sein können. Mit dieser

suppression wird eine Meldung unterdrückt, die durch die Verwendung von OpenMP entsteht, für die Entwicklung der Software jedoch keine weitere Bedeutung hat.
Suppressions können durch Valgrind selbst generiert werden. Mittels der Option

```
--gen-suppressions=all
```

wird zu jeder Meldung eine gültige suppression generiert. Diese sind sehr spezifisch und enthalten nur die exakte Abfolge von Funktionsaufrufen. Mittels der Wildcards lassen sie sich aber generalisieren. So wird das suppression file vereinfacht und übersichtlicher. Dadurch, dass unerwünschte Meldungen unterdrückt werden, bleiben nur noch diejenigen übrig, die für die Entwickler interessant sind. Mit Hilfe dieser Informationen ist es ihm nun möglich, seinen Code so zu verändern, dass Race Conditions vermieden werden.
Nachdem Valgrind bzw. Helgrind jetzt ausreichend vorbereitet ist, kann ein erster Testlauf ausgeführt werden. Zunächst wird nur das Beispielprogramm `main_task_model` auf Race Conditions überprüft. Dadurch, dass alle irrelevanten Meldungen herausgefiltert sind, weist die Ausgabe von Helgrind nur noch auf true positives hin. Bei einem ersten Testlauf konnte bereits ein Fehler in der von der Universität Erlangen entwickelten Bibliothek Ghost ein Fehler identifiziert werden. Dieser äußerte sich in einer möglichen Race Condition. Da die Entwickler des DLR und der Universität Erlangen in täglichem Kontakt stehen, konnten diese über den Fehler informiert werden und diesen schnell beheben. Dieses Beispiel zeigt, dass der Nebenläufigkeitstest mit Helgrind grundsätzlich funktioniert und eingesetzt werden kann.
Um diese Ergebnisse weiter zu untermauern und um zu überprüfen, ob nicht auch *true positives*[32] durch die suppressions herausgefiltert werden, wurde das Programm *main_task_model* auf zwei Weisen abgeändert. Die beiden entstandenen Programme *main_racecond* und *main_deadlock* sollen jeweils Race Conditions bzw. einen Deadlocks beinhalten.

[32] also Fehler, die auch wirklich welche sind

Der Testlauf hat ergeben, dass Valgrind ein brauchbares Ergebnis liefert. Während beim Programm *main_task_model* keine Fehlermeldungen mehr erzeugt wurden, wurden in den beiden anderen Beispielprogrammen jeweils Fehler gefunden. Wichtig ist dennoch, dass der Entwickler sich grundlegend mit dem Valgrind-Tool auskennt. Die Fehlermeldung wirken für einen Laien im ersten Moment ein wenig kryptisch. Jeder Entwickler, der mit dem Werkzeug arbeitet, sollte eine Fehlermeldung identifizieren können, damit der Fehler behoben werden kann.

Um den Test möglichst komfortabel zu gestalten, soll die Ausführung von Valgrind möglichst automatisiert werden. Dazu wird ein bash[33]-script[34] erstellt, welches die multiple Ausführung von Valgrind ermöglicht. Außerdem wird nicht in jedem Durchlauf ein neues log file erstellt, sondern lediglich eins an welches die Ausgabe jeweils angehängt wird. Dieses log file lässt sich dann automatisiert auf Fehlermeldungen überprüfen. Findet man keine Fehlermeldung im log file besteht eine sehr hohe Wahrscheinlichkeit, dass keine Fehlerzustände im Programmcode vorhanden sind.

Des Weiteren muss dieses Skript in den Testprozess von CMake (vgl. Kapitel 2.1.2) integriert werden. Im CMake-File sind bereits die verschiedenen Testszenarien, die mit Google Test entwickelt wurden, integriert. Mittels des Befehls `add_test()` lässt sich ein weiterer Test in den automatisierten Testprozess aufnehmen. An dieser Stelle ist es möglich, das Skript einzubinden. Das Skript gibt nach einem fehlerfreien Durchlauf eine 0 als Status zurück. Tritt irgendein Fehler auf, liefert es einen anderen Wert. Eine 0 führt bei der Ausführung des automatisierten Testprozesses dazu, dass der Test als fehlerfrei angezeigt wird. Wird ein anderer Wert zurückgegeben, wird dies als Fehler registriert und eine dementsprechende Ausgabe findet statt. Wo genau der Fehler liegt, lässt sich allerdings nur mit Hilfe des log files identifizieren.

[33] Bourne-again-shell, eine häufig verwendete shell unter Linux-Betriebssystemen

[34] Ein Skript ermöglicht die Hintereinanderausführung mehrerer shell-Befehle. In diesem shell-Skript lassen sich auch Kontrollstukturen einbinden.

Mittels des Befehls *make test* werden nun zunächst die mit Google Test erstellten Komponenten- und Integrationstests ausgeführt. Daraufhin wird das Skript ausgeführt, welches das Valgrind-Tool mehrfach aufruft, um die Nebenläufigkeit zu testen. In der Praxis funktioniert dieses Verfahren sehr gut und macht die Ausführung der Tests deutlich komfortabler. Dadurch wird auf einen Blick sichtbar, ob es Fehler im parallelen Programmcode gibt.

Das Ergebnis, welches durch den CMake-Testprozess erstellt wird, lässt sich ebenfalls sehr gut in die weitere Continuous Integration eingliedern. Dadurch, dass ein eindeutiges Ergebnis geliefert wird, können auf dem Jenkins-Server die einzelnen Status der Tests leicht abgelesen werden.

```
Running tests...
Test project /home/tisc_ti/dfg-essex/build
    Start 1: kernels-tests
1/6 Test #1: kernels-tests .................... Passed 4.93 sec
    Start 2: core-tests
2/6 Test #2: core-tests ....................... Passed 3.02 sec
    Start 3: sched-tests
3/6 Test #3: sched-tests ...................... Passed 5.53 sec
    Start 4: krylov-tests
4/6 Test #4: krylov-tests ..................... Passed 3.36 sec
    Start 5: bench-tests
5/6 Test #5: bench-tests ...................... Passed 10.82 sec
    Start 6: concurrency-test
6/6 Test #6: concurrency-test ................. Passed 35.24 sec

100% tests passed, 0 tests failed out of 6

Total Test time (real) = 63.17 sec
```

**Listing 6:** CMake-Test-Ausgabe für einen fehlerfreien Testdurchlauf

Problem bei Valgrind ist, dass das jeweilige suppression file sehr systemabhängig ist. Das heißt, sollten beispielsweise andere Versionen von OpenMP genutzt werden, können wieder false positives auftreten, die auf dem Testsystem nicht festgestellt wurden. Daher benötigt Valgrind auf jedem System eine individuelle Einrichtung und wahrscheinlich jeweils unterschiedliche suppression files. Diese sollten sich zwar ohne Probleme generieren lassen, dennoch erhöht dieses Problem den Aufwand, zur Einrichtung des Tools. Allerdings kann dieses Problem bei anderen Werkzeugen ebenfalls auftreten. Die Verwendung von Valgrind bietet sich daher dennoch an.

### 4.3.2 Testen der MPI-Funktionen

Die Tests, die mit dem Google C++ Testing Framework entwickelt wurden, lassen sich bereits parallelisiert ausführen. Dazu wird folgender Befehl verwendet:

```
mpirun -np <NP> ./phist-0.2.0-dev-core-test
```

Mit der Option `-np` lässt sich angeben, wie viele MPI-Prozesse gestartet werden sollen. Tritt bei diesem Testlauf eine Fehlerwirkung auf, die durch einen Testdurchlauf ohne MPI nicht auftreten, ist es wahrscheinlich, dass die Fehlerursache in der Verwendung von MPI liegt. Dieses Testverfahren lässt sich mittels eines bash-scripts ebenfalls automatisieren. Dazu wird für jeden ausführbaren Test ein Skript geschrieben, welches die Tests mehrfach durchlaufen lässt und jeweils die Anzahl der verwendeten MPI-Prozesse schrittweise erhöht. So können automatisiert verschiedene Konfigurationen (z.B. 1, 2, 4 oder 8 Prozesse) getestet werden. Dieses Skript lässt sich ebenfalls in den CMake-Testprozess integrieren. Dieses Verfahren allein genügt allerdings nicht, um Fehler in der Verwendung von MPI auszuschließen. Des Weiteren wird dadurch nicht ersichtlich, an welcher Stelle der Fehler auftritt. Die Fehlersuche wäre somit sehr aufwendig. Es eignet sich dennoch, um eine realistische Einschätzung darüber zu bekommen, ob MPI korrekt verwendet wurde oder ob es zu Fehlern führt.

Bei diesem Verfahren kam es zu dem Problem, dass jeder MPI-Prozess eine eigene Ausgabe hatte. So wurde für jeden Test mehrfach angezeigt und in die Ausgabedatei geschrieben, dass er korrekt lief, bzw. fehlgeschlagen ist. Um dies zu verhindern, musste ein eigener *Event-Listener* erstellt werden. Mit diesem kann dann die Ausgabe so modifiziert werden, dass nur der root[35]-Prozess den jeweiligen Ausgabekanal nutzen kann. Der Standard-Event-Listener musste zusätzlich aus der Liste der Event-Listener entfernt werden. So konnte die mehrfache Ausgabe verhindert werden.

[35]Wurzel

# 5 Fazit und Ausblick

In diesem Kapitel werden kurz die Ergebnisse der Arbeit zusammengefasst und ausgewertet. Im Anschluss wird ein Ausblick in die Zukunft gegeben, der angeben soll, ob sich das Testkonzept im Projekt etablieren kann.

## 5.1 Fazit

Im Rahmen dieser Bachelorarbeit wurde ein für das Projekt ESSEX angemessenes Testkonzept erstellt, da die besonderen Gegebenheiten im Projekt, die in Kapitel 3.1 beschrieben wurden, hinreichend betrachtet und eine Lösung für die jeweilige Problemstellung zunächst theoretisch erarbeitet. Dies wurde mittels verschiedener Tools vereinfacht und automatisiert. Beispiele für die Erstellung von Tests und die Verwendung der Tools wurden in Kapitel 4 beschrieben und die Tools evaluiert. Schwerpunkt der Arbeit lag deutlich beim Testen der Nebenläufigkeit und nahm den Großteil der Zeit in Anspruch. Dieses ist insbesondere bei der hohen Anzahl von Prozessen, die im Projekt gefordert sind, noch nicht weit erforscht. Daher war es hier besonders wichtig, ein angemessenes Konzept zu entwickeln. Einige Teile des Testkonzepts müssen noch praktisch umgesetzt werden, um zu beweisen, dass die Annahmen, die gemacht wurden, sich bestätigen. Dies sollte allerdings keine Probleme bereiten.
Das eigentliche Testkonzept ist realistisch gestaltet und sollte umsetzbar sein. Änderungen am Testkonzept im Laufe des weiteren Software-Entwicklungs-Prozesses sind

üblich und auch erwünscht. Die Entwickler vom DLR sind bereit, dieses Testkonzept in den Entwicklungsprozess miteinzubeziehen und es anzuwenden.
Obwohl es sich um ein durchaus sehr umfangreiches und komplexes Thema handelt und der Bearbeitungszeitraum von 12 Wochen relativ kurz war, wurden alle Aspekte zunächst theoretisch behandelt und weitgehend mit praktischen Beispielen belegt. Die Funktionalität des Tools MUST (vgl. Kapitel 3.2.3) konnte nicht mehr exemplarisch nachgewiesen werden, da der Zeitaufwand zu hoch gewesen wäre. Dies soll jedoch in naher Zukunft geschehen und sollte keine Probleme bereiten.
Ein Großteil der Arbeit beinhaltete die Literaturrecherche zu den Themen Testen und parallele Algorithmen. Eine umfassende Recherche war hier nötig, um ein fundiertes Fachwissen im Bereich Testen zu erlangen. Des Weiteren war es ebenfalls wichtig, die Grundlagen des parallelen Programmierens zu beherrschen. Ein Testkonzept in diesem Projekt wäre sonst nicht realisierbar. Dieses gesammelte Wissen kann durch diese Bachelorarbeit an zukünftige Mitarbeiter weitergegeben werden. Dies ist besonders wichtig, da zu dem Thema nicht viel Literatur zu finden ist. Die vorhandene Literatur eignet sich nur bedingt um ein Testkonzept zu entwickeln, da es sich um eine sehr spezielle Software handelt.
An einer Stelle konnte deutlich gezeigt werden, dass das Testkonzept greift. Dies hat sich darin geäußert, dass bei der Ausführung von Valgrind (vgl. Kapitel 4.3.1) nach Änderungen am Programmcode häufig Race Conditions gefunden wurden und daraufhin auch schnell behoben werden konnten. Es beweist, dass für die Nebenläufigkeitstests Valgrind eine gute Wahl war. Dies sollte Fehlerwirkungen auch in Zukunft schnell aufdecken. Teilweise ist es allerdings problematisch, da nicht gezeigt werden konnte, ob das suppression file portierbar ist. Daher kann nicht genau gesagt werden, ob es sich 1:1 auf andere Systeme übertragen lässt.

## 5.2 Ausblick

Zusammenfassend kann gesagt werden, dass die Entwickler mit dem Testkonzept sehr zufrieden sind. Sie sind bereit das Testkonzept in dieser Form zu übernehmen und weiterzuführen. Es ist wichtig, dass es in Zukunft weiterentwickelt wird und an die Änderungen im Projekt angepasst wird. Es ist wahrscheinlich, dass nicht alles so umgesetzt wird, wie es in der Theorie gewünscht wird. Dennoch kann das Testkonzept sehr wohl als Grundlage für die Testprozesse genutzt werden. Für neue Mitarbeiter und studentische Hilfskräfte, die im Bereich Testen tätig sind, ist es eine sehr gute Hilfestellung.
Des Weiteren ist ein wichtiger Schritt die Eingliederung in die Continuous Integration. Dies sollte möglichst für das im DLR entwickelte Softwarepaket und zusätzlich für die Bibliotheken, die an den anderen Forschungseinrichtungen entwickelt werden, durchgeführt werden. Dadurch können alle Entwickler den aktuellen build- und Teststatus überwachen. Dies ermöglicht eine deutlich komfortablere Entwicklung. Außerdem können Performance-Schwierigkeiten dadurch schnell erkannt und behoben werden.
Wichtig ist, dass in naher Zukunft die Funktionalität von MUST in der Praxis nachgewiesen werden muss. Ein Problem dabei stellt nur der hohe Installationsaufwand dar. Die eigentliche Evaluation des Tools und die Auswertung der Ergebnisse sollte unproblematisch sein.
Als Fernziel soll das Testkonzept dazu beitragen, die Softwarequalität zu erhöhen. Ebenfalls sinnvoll könnte es sein, einen durchgängigen Qualitätsmanagement-Prozess in das Projekt ESSEX einzuführen, in den das Testkonzept integriert werden kann. Dieses würde die Softwarequalität weiter erhöhen und einen Qualitätsstandard garantieren. Wünschenswert wäre es auch, wenn dieses Testkonzept Entwickler in anderen Projekten dazu anregen würde, dem Thema Testen mehr Beachtung zu schenken.

# A Testfälle für den Gram-Schmidt-Algorithmus

Aus der folgenden Spezifikation für den Algorithmus ergeben sich die unten stehenden Testfälle.

## A.1 Spezifikation des Algorithmus

```
1 // This is the main orthogonalization routine in PHIST.
2 // It takes an orthogonal basis V and a set of vectors W,
3 // and computes [Q,R1,R2] such that Q*R1 = W-V*R2, Q'Q=I.
4 // (Q overwrites W here).
5 // The matrices R1 and R2 must be pre-allocated by the caller.
6 //
7 // The algorithm used is up to numSweeps steps of classical
8 // block Gram-Schmidt, alternated with QR factorizations to
9 // normalize W. The method stops if the reduction in norm of W
10 // by a CGS step is less than approx. a factor sqrt(2).
11 //
12 // If we find that W does not have full column rank,
13 // the matrix Q is augmented with random vectors which are made
14 // mutually orthogonal and orthogonal against V. The original
15 // rank of W is returned in ierr at the end of the routine. If
16 // it happens somewhere during the process, we return an error
```

```
17 // code ierr=-7.
18 //
19 // If a breakdown occurs, indicating that one of the columns of
20 // W lives in the space spanned by the columns of V, ierr=-8 is
21 // returned. A more convenient behavior may be added later, like
22 // randomizing the column(s) as before.
23 //
24 // If the decrease in norm in one of the columns
25 // in the last CGS sweep indicates that the algorithm has not
26 // yet converged, we return ierr=-9 to indicate that more steps
27 // may be advisable. This should not happen in practice if
28 // numSweeps>=2 ('twice is enough').
```

## A.2 Testfälle

| **Nr.** | **V** | **W** | **num_sweeps** | **ierr** | **R1/R2** |
|---|---|---|---|---|---|
| **1** | $V^{100\times 20} \wedge V^TV = E$ | $W^{100\times 4} \wedge l.u.$ | 3 | $\surd$ | $\surd$ |
| **2** | $V^{100\times 20} \wedge V^TV = E$ | $W^{100\times 4} \wedge l.a.$ | 3 | $\surd$ | $\surd$ |
| **3** | $V^{100\times 20} \wedge V^TV \neq E$ | $W^{100\times 4} \wedge l.u.$ | 3 | $\surd$ | $\surd$ |
| **4** | $V^{25\times 20} \wedge V^TV = E$ | $W^{25\times 6} \wedge l.u.$ | 3 | $\surd$ | $\surd$ |
| **5** | $V^{100\times 20} \wedge V^TV = E$ | $W^{90\times 4} \wedge l.u.$ | 3 | $\surd$ | $\surd$ |
| **6** | $V^{100\times 20} \wedge V^TV = E$ | $W^{100\times 4} \wedge l.u.$ | $-1$ | $\surd$ | $\surd$ |
| **7** | $V^{100\times 20} \wedge V^TV = E$ | $W^{100\times 4} \wedge l.u.$ | 3 | $\times$ | $\surd$ |
| **8** | $V^{100\times 20} \wedge V^TV = E$ | $W^{100\times 4} \wedge l.u.$ | 3 | $\surd$ | $\times$ |

l.u. = linear unabhängig

l.a. = linear abhängig

# B Testkonzept nach IEEE 829

Es folgt ein Testkonzept, das an die IEEE Norm 829 angelehnt ist. Hier ist es sehr allgemein gehalten. Testobjekte, Leistungsmerkmale und vor allem die Teile, die das Personal betreffen, müssen noch konkretisiert werden.

**Testkonzept für das Projekt ESSEX, Ver. 1.0, Freigabe: nein**

**Einführung**

Im Projekt ESSEX (**E**quipping **S**parse **S**olvers for **Ex**ascale) sollen Algorithmen zur Eigenwertlösung großer dünnbesetzter Matrizen entwickelt werden. Diese sollen Exascale-Computer-fähig und hoch parallel sein.
Grundlage des Testkonzepts liefert das ESSEX-Proposal[WBF+12] und die Anforderungen der Komponenten und Algorithmen. Die Softwarequalität soll nach ISO 25010[iso13c] gewährleistet sein.
Beteiligte Unternehmen sind das Deutsche Zentrum für Luft- und Raumfahrt und die Universitäten Erlangen, Greifswald und Wuppertal.

**Testobjekte**

Alle im Rahmen des Projekts entwickelten Komponenten und Algorithmen sollen getestet werden. Dazu zählen die in Erlangen entwickelten verschiedene Funktionen z.B. zur Matrix-Matrix- oder Matrix-Vektor-Multiplikation, verschiedene Algorithmen z.B.

Orthogonalisierungsverfahren und Eigenwertlöser. Weitere Bibliotheken, Algorithmen und übrige Softwarekomponenten, die an den übrigen Forschungseinrichtungen entwickelt werden, sollen ebenfalls getestet werden.
Im DLR beispielsweise handelt es sich um alle *core*- und *kernel*-Komponenten, sowie um Krylov-, Jacobi-Davidson- und weitere Verfahren.

**Zu testende Leistungsmerkmale**

Zum einen soll sichergestellt sein, dass alle Komponenten und Algorithmen getestet und abgenommen sind. Des Weiteren ist es in diesem Projekt besonders wichtig, die korrekte Verhaltensweise der parallelen und nebenläufigen Programmteile zu überprüfen. Das Fehlertoleranzsystem, das an der Universität Erlangen entwickelt wurde, muss ebenfalls mitgetestet werden. Die Performancemessung beschränkt sich auf die Messung von Laufzeiten. Profiling-Tools (siehe unten) sollen nicht automatisiert verwendet werden. Jedem Entwickler sei überlassen, welches Tool er favorisiert. Diese Werkezuge können dann zur Unterstützung des Debugging-Prozesses genutzt werden, um beispielsweise Flaschenhälse im Programm zu entdecken und beheben. Eine Liste von möglichen Tools befindet sich ebenfalls weiter unten.

**Leistungsmerkmale, die nicht getestet werden**

**Teststrategie**

Da es sich lediglich um ein Forschungsprojekt handelt und keine Gefahr für Mensch oder System bestehen, wird keine explizite Risikoanalyse durchgeführt. Gefahren, die die Sicherheit des Systems betreffen, existieren ebenfalls nicht. Außerdem wird die Software nicht von einem Kunden im eigentlichen Sinne abgenommen, sondern dient lediglich Forschungszwecken. Dennoch ist es wichtig, dass alle Softwareteile funktionieren, damit Ressourcen nicht unnötig belastet werden. Außerdem müssen die Rechenergebnisse

zuverlässig sein.
Es wird keine formale Teststrategie verfolgt, da die Einführung des Testkonzepts relativ spät war, da im Projekt bereits mehrere Komponenten vorhanden waren. Der Entwickler ist zunächst dazu angehalten, nach der Erstellung beziehungsweise Änderung einer Komponente das dazugehörige Testszenario anzupassen. Er muss gewährleisten, dass nach der Änderung Komponenten-, Integrations- und Systemtests weiterhin erfolgreich waren.
Wichtig sind vor allem die Komponententests, da die hier entwickelten Funktionen besonders häufig verwendet werden.

### Abnahmekriterien

Das jeweilige Testobjekt kann freigegeben werden, sobald mindestens 80% der geplanten Tests implementiert und 100% fehlerfrei sind. Es muss gewährleistet sein, dass kein Fehler der Klasse-1 (Systemabsturz, möglicherweise mit Datenverlust) gefunden wird. Des Weiteren kann das Testobjekt nicht freigegeben werden, falls es im Vergleich zu einer vorhergehenden Version zu enormen Performance-Einbußen oder bei neu entwickelter Software nicht zur erwarteten Performance kommt.

### Testdokumentation

Da die Entwickler zum großen Teil für ihre eigene Software verantwortlich sind, ist eine Testdokumentation nicht zwingend notwendig. Falls es Kommunikationsbedarf zwischen den Entwicklern gibt, kann jederzeit der Kontakt hergestellt und das Problem geklärt werden. Das Vorgehen im Testprozess wird in diesem Dokument beschrieben.

**Testinfrastruktur**

Da die Entwickler auch die Testaufgaben übernehmen, sind sie bereits mit Arbeitsplätzen, die zur Testerstellung genügen, ausgestattet. Dadurch ist auch das Betriebssystem und die Entwicklungsumgebung nach belieben des Entwicklers gewählt. Im Regelfall handelt es sich um eine Linuxdistribution. Die Testinfrastruktur des Gesamtsystems ist noch nicht definiert, da es für die Verwendung auf unterschiedlichen Endsystemen gedacht ist. Gemeinsam haben diese, dass sie die Ausführung von paralleler Software mit MPI und einer weiteren Programmierschnittstelle für die parallele Programmierung auf Node-Ebene ermöglichen. Des Weiteren handelt es sich dabei um Supercomputer. Die Tests sollten sowohl auf dem Rechner des jeweiligen Entwicklers, als auch auf einem oder mehreren verfügbaren Supercomputern ausgeführt werden.

**Verantwortlichkeiten**

Verantwortlich für den geregelten Ablauf des Testprozesses sind die Entwickler selbst. Ein Testexperte der jeweiligen Einrichtung sollte beratend zur Verfügung stehen, falls Probleme auftreten. Ein Mitarbeiter von der jeweiligen Forschungseinrichtung ist verantwortlich für die Umsetzung von Tests und die Einhaltung des in diesem Testkonzept festgelegten Testbedingungen.

**Personal**

siehe *Verantwortlichkeiten*
Entwickler sollten sich miteinander austauschen und Probleme klären, sowie informelle Reviews durchführen. Des Weiteren ist es sinnvoll, studentische Hilfskräfte mit dem Projekt zu betrauen und in den Entwicklungs- bzw. Testprozess einzubinden.

### Zeitplan

Die Projektlaufzeit beträgt 36 Monate. Über den gleichen Zeitraum erstreckt sich auch die Entwicklung der Tests, da an der Software ebenfalls andauernd gearbeitet wird. Da kein einheitliches Entwicklungsmodell vorliegt, können die Testaktivitäten nicht auf dieses abgestimmt werden.

### Genehmigung/Freigabe

Da es sich hier nicht um das endgültige Testkonzept handelt ist eine Freigabe in dieser Form nicht erforderlich. Ansonsten geschieht die Freigabe von den Entwicklern in Absprache mit den Projektleitern.

# Literaturverzeichnis

[ama13a] *Amazon Elastic Compute Cloude (EC2)*. http://aws.amazon.com/de/ec2, 2013. – Abruf: 11. September 2013

[ama13b] *amazon web services*. http://aws.amazon.com/de/, 2013. – Abruf: 11. September 2013

[ari13] *Launch Vehicles - Ariane 5*. http://www.esa.int/Our_Activities/Launchers/Launch_vehicles/Ariane_5_ECA, 2013. – Abruf: 10. Juli 2013

[aup12] *The Agile Unified Process (AUP)*. http://www.ambysoft.com/unifiedprocess/agileUP.html, 2012. – Abruf: 18. Juli 2013

[BDD+00] BAI, Z. ; DEMMEL, J. ; DONGARRA, J. ; RUHE, A. ; VORST, H. van d.: *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, 2000 (Software, Environments and Tools). – ISBN 9780898714715

[Ber13] BERLIK, S.: *Softwaretechnik II - Grundlagen des Softwaretestens*. http://pi.informatik.uni-siegen.de/berlik/swt/swt.pdf, 2013. – Abruf: 11. Juli 2013

[BKL+09] BALZERT, H. ; KOSCHKE, R. ; LÄMMEL, U. ; BALZERT, H. ; LIGGESMEYER, P. ; QUANTE, J.: *Lehrbuch Der Softwaretechnik: Basiskonzepte Und Requirements Engineering*. Spektrum Akademischer Verlag GmbH,

2009. – ISBN 9783827422477

[CL12] Choi, W. ; Lee, J.: *Graphene: Synthesis and Applications*. Taylor & Francis, 2012 (Nanomaterials and Their Applications). – ISBN 9781439861875

[cma] *CMake - Cross-platform Make*. http://www.cmake.org/, . – Abruf: 16. August 2013

[Com90] Committee, IEEE Computer Society. Standards C.: *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, 610*. IEEE, 1990 (ANSI / IEEE Std). – ISBN 9781559370790

[dfg13] *DFG - Deutsche Forschungsgemeinschaft*. http://www.dfg.de/, 2013. – Abruf: 15. Juli 2013

[DLR13a] *DLR Portal*. http://www.dlr.de/dlr, 2013. – Abruf: 11. Juli 2013

[DLR13b] *Simulations- und Softwaretechnik (SC)*. http://www.dlr.de/sc/, 2013. – Abruf: 09. Juli 2013

[DP02] Dr. Pankratius, V.: *Software Engineering für moderne, parallele Plattformen - 12. Testen und Fehlerfindung in parallelen Programmen*. http://www.ipd.uka.de/Tichy/uploads/folien/157/SEPP10-TestDebugging.pdf, 2002. – Abruf: 18. Juli 2013

[DR06] Desikan, S. ; Ramesh, G.: *Software Testing: Principles and Practice*. Pearson Education Canada, 2006. – ISBN 9788177581218

[Dre] Dresden, Technische U.: *MUST*. http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/must, . – Abruf: 31. August 2013

[ESA13] *European Space Agency*. http://www.esa.int/ger/ESA_in_your_country/Germany, 2013. – Abruf: 15. Juli 2013

[FG99] Fewster, M. ; Graham, D.: *Software test automation: effective use of test execution tools*. Addison-Wesley, 1999 (ACM Press books). – ISBN

9780201331400

[GHJV94] GAMMA, E. ; HELM, R. ; JOHNSON, R. ; VLISSIDES, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. – ISBN 9780321700698

[Gmb] GMBH, GWT-TUD: *Vampir - Performance Optimization*. http://www.vampir.eu/, . – Abruf: 05. September 2013

[goo] *googletest - Google C++ Testing Framework*. https://code.google.com/p/googletest/, . – Abruf: 2. August 2013

[gra12] *Graphen*. http://www.nanopartikel.info/cms/Wissensbasis/Graphen, 2012. – Abruf: 10. Juli 2013

[Gro] GROSSMANN, S.: *Heisenbergsche Unschärferelation*. http://www.weltderphysik.de/fileadmin/user_upload/Redaktion/Gebiete/theorien/Grundzuege_der_QM/201011_unschaerfe.pdf, . – Abruf: 18. Juli 2013

[ibm12] *IBM - Deutschland*. http://www.ibm.com/de/de/, 2012. – Abruf: 11. Juli 2013

[IEC10] IEC: *International Electrotechnical Commission*. http://www.iec.ch/, 2010. – Abruf: 12. Juli 2013

[iee13] *IEEE*. http://www.ieee.org/index.html, 2013. – Abruf: 11. Juli 2013

[ISO13a] *ISO - International Organization for Standardization*. http://www.iso.org/iso/home.html, 2013. – Abruf: 09. Juli 2013

[iso13b] *ISO/IEC 25000:2005 Software Engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE*. http://www.iso.org/iso/catalogue_detail.htm?csnumber=35683, 2013. – Abruf: 12. Juli 2013

[iso13c] *ISO/IEC 25010:2011 Systems and software engineering - Systems and*

*software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733, 2013. – Abruf: 12. Juli 2013

[iso13d] *ISO/IEC 9126-1:2001 Software engineering - Product quality - Part 1: Quality model*. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749, 2013. – Abruf: 12. Juli 2013

[ist13] *International Software Testing Qualification Board*. http://www.istqb.org/, 2013. – Abruf: 12. Juli 2013

[jen13] *Jenkins*. http://jenkins-ci.org/, 2013. – Abruf: 19. Juli 2013

[Jül] Jülich, Forschungszentrum: *scalasca*. http://www.scalasca.org/, . – Abruf: 05. September 2013

[Kle12] Kleuker, S.: *Qualitätssicherung Durch Softwaretests: Vorgehensweisen und Werkzeuge Zum Test Von Java-Programmen*. Vieweg Verlag, Friedr, & Sohn Verlagsgesellschaft mbH, 2012. – ISBN 9783834820686

[Kro13] Kroll, P.: *Automated testing of distributed systems using on-demand virtual infrastructure*. 2013

[Lan96] Lann, Gérard L.: *The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems*. http://hal.inria.fr/docs/00/07/36/13/PDF/RR-3079.pdf, 1996. – Abruf: 10. Juli 2013

[lik13] *likwid - Lightweight performance tools*. https://code.google.com/p/likwid/, 2013. – Abruf: 05. September 2013

[Lio96] Lions, J. L.: *Ariane 5 Flight 501 Failure - Report by the Inquiry Board*. http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf, 1996. – Abruf: 10. Juli 2013

[Maj12] Majchrzak, T.A.: *Improving Software Testing: Technical and Organiza-*

*tional Developments*. Springer Berlin Heidelberg, 2012 (SpringerBriefs in information systems). – ISBN 9783642274640

[mpi] *The Message*

[MSB11] Myers, G.J. ; Sandler, C. ; Badgett, T.: *The Art of Software Testing*. Wiley, 2011 (ITPro collection). – ISBN 9781118133156

[Mye79] Myers, G.J.: *The Art of Software Testing*. Wiley, 1979 (A Wiley-Interscience publication). – ISBN 9780471043287

[nVI] nVIDIA: *Parallele Berechnungen mit CUDA*. http://www.nvidia.de/object/cuda-parallel-computing-de.html, . – Abruf: 18. August 2013

[ope] *OpenMP*. http://openmp.org/wp/, . – Abruf: 22. August 2013

[PKS02] Pol, M. ; Koomen, T. ; Spillner, A.: *Management und Optimierung des Testprozesses: ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap*. dpunkt-Verlag, 2002. – ISBN 9783898641562

[RC02] Rubini, A. ; Corbert, J.: *Race Conditions*. http://www.oreilly.de/german/freebooks/linuxdrive2ger/charrace.html, 2002. – Abruf: 18. Juli 2013

[Roi05] Roitzsch, E.H.P.: *Analytische Softwarequalitätssicherung in Theorie und Praxis*. Verlag-Haus Monsenstein und Vannerdat, 2005 (Edition Octopus). – ISBN 9783865822024

[scr13] *Scrum*. http://www.scrum.org/, 2013. – Abruf: 18. Juli 2013

[scW13] *SC - Wiki - Software Project Mannual*. https://wiki.sistec.dlr.de/SoftwareProjectManual, 2013. – internes Dokument. Abruf: 18. Juli 2013

[SL12] Spillner, A. ; Linz, T.: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*.

Dpunkt.Verlag GmbH, 2012 (ISQL-Reihe). – ISBN 9783864900242

[Soc08] SOCIETY, IEEE C.: *IEEE Standard for Software Reviews and Audits*. http://www.baskent.edu.tr/~zaktas/courses/Bil573/IEEE_Standards/1028_2008.pdf, 2008. – Abruf: 15. Juli 2013

[Soc10] SOCIETY, IEEE C.: *IEEE Standard Classification for Software Anomalies*. http://www.baskent.edu.tr/~zaktas/courses/Bil573/IEEE_Standards/1044_2009.pdf, 2010. – Abruf: 11. Juli 2013

[spa12] *Sparse Matrix Data Structures*. http://www.cs.indiana.edu/classes/p573/notes/sparse/sparsemat.html, 2012. – Abruf: 10. Juli 2013

[vala] *Valgrind*. http://valgrind.org/, . – Abruf: 8. August 2013

[valb] *Valgrind - Helgrind*. http://valgrind.org/info/tools.html#helgrind, . – Abruf: 8. August 2013

[WBF+12] WELLHEIN, G. ; BASERMANN, A. ; FEHSKE, H. ; HAGER, G. ; LANG, B.: *ESSEX-Proposal*. 2012