Implementierung von Performance Assertions für MPI Metriken

Bachelor-Thesis von Simon Reuß
Dezember 2013

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Scientific Computing

Implementierung von Performance Assertions für MPI Metriken

Vorgelegte Bachelor-Thesis von Simon Reuß

1. Gutachten: Prof. Dr. Christian Bischof
2. Gutachten: Dipl.-Inform. Christian Iwainsky

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 18. Dezember 2013

(Simon Reuß)

Zusammenfassung

Performance Analysen rechenintensiver Programme sind aufgrund der benötigten Kenntnisse und Ressourcen eine Herausforderung, weswegen Werkzeuge zur Unterstützung des Benutzers entwickelt und eingesetzt werden. Einige dieser Analyseprogramme versuchen den Prozess zu automatisieren und die Eingriffe des Benutzers zu minimieren.

Performance Assertions sind ein Konzept zur automatischen Validierung von benutzerdefinierten Aussagen über das Laufzeitverhalten eines Programmes. Die zu überprüfenden Aussagen werden in einer definierten Sprache formuliert und ohne direkte Beteiligung des Benutzers anhand der Daten einer Programmausführung kontrolliert.

Im Rahmen dieser Arbeit wird dieses Konzept mit Fokus auf parallele Anwendungen des Message Passing Interface (MPI) implementiert. Eine zur Validierung der Umsetzung durchgeführte Evaluation zeigt, dass Performance Assertions für die Überprüfung von Performance Modellen geeignet sind und der Overhead im Vergleich zu Profiling und Tracing geringer ist.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Performance-Analyse rechenintensiver Programme stellt ein Feld von besonderem Interesse für Forschung und Effizienzbetrachtungen im wirtschaftlichen Umfeld des Hochleistungsrechnens dar. Eine Möglichkeit hierbei ist der Einsatz von Analyse-Tools, um anhand von Event Traces, Profiling und Aufzeichnung weiterer Metriken, wie Speicherverbrauch, Netzwerkauslastung oder spezieller Hardware Counter, einen Eindruck über das Laufzeitverhalten der Applikation zu gewinnen. Die Untersuchung der sich ergebenden Datenmengen ist aufgrund des Umfangs und der Deutung dieser Daten eine Herausforderung. Es existieren jedoch Analysewerkzeuge, die gegebenenfalls in der Lage sind, die gesammelten Daten automatisch auf häufig auftretende Performanceprobleme zu überprüfen, beziehungsweise die Daten zur erleichternden Deutung entsprechend aufzubereiten und zu visualisieren.

Dabei ist die Analyse des Leistungsverhaltens keinesfalls ein einmaliges Ereignis innerhalb des Lebenszyklus eines Programmes, welcher durchaus mehrere Jahre oder Jahrzehnte umfassen kann: Während der fortlaufenden Entwicklung oder Wartung kann es durch kleine Änderungen zu unerwarteten Leistungsabweichungen kommen, deren Bestehen erhebliche Ressourcen kosten kann. Folglich müsste die Leistungsanalyse in regelmäßigen Abständen wiederholt durchgeführt werden, was jedoch mit erheblichem Aufwand verbunden ist.

Eine ähnliche Herausforderung entsteht durch die rasante Entwicklung der Hardware: Waren zum Beispiel frühere Hochleistungsrechner noch mit wenigen Prozessorkernen ausgestattet, so sind heute massiv parallele Architekturen mit mehreren tausend Prozessorkernen keine Seltenheit mehr. Hinzu kommt die steigende Verbreitung von Hardwarebeschleunigung, wie GPGPU[1], sowie die ständige Erweiterung der bestehenden Prozessorarchitekturen, wie beispielsweise die fortlaufende Ergänzung der x86 Architektur um neue Vektorinstruktionen.

In Anbetracht der sich ständig ändernden Hardware, Auswirkungen von Codeänderungen und Datensatzänderungen ist eine automatische Performance Analyse empfehlenswert. Eine automatisierte Möglichkeit hierzu bieten sogenannte Performance Assertions. Die leistungsbezogenen Erwartungen an einen bestimmten Programmabschnitt werden mithilfe einer definierten Sprache formuliert und entweder durch direkte Annotation des Quellcodes oder Speicherung in einer separaten Datei fixiert. Die Validierung der Erwartungen kann entweder post mortem (nach der Programmterminierung) anhand von Daten, die bei einer Ausführung des Programmes generiert und gesammelt wurden oder aber unmittelbar während der Ausführung erfolgen.

Durch die Verwendung von Performance Assertions können somit potentiell problematische Codestellen überwacht werden. Zur Lokalisierung dieser Stellen ist gegebenenfalls eine vorherige Analyse notwendig. Die Überprüfung der formulierten Erwartungen kann dabei während jeder Ausführung ohne Eingreifen des Benutzers erfolgen. Dies erlaubt unter anderem die Realisierung von Regressionstests, um zu gewährleisten, dass durch eine Codeänderung die Performance eines Programmes nicht negativ beeinflusst wird. Eine entsprechend angepasste Implementierung könnte zudem eingesetzt werden, um das Verhalten von kritischen Softwaresystemen zu überwachen und im Falle einer nicht zulässigen Überschreitung gegebenenfalls automatisch Schritte zur Konfliktauflösung einzuleiten. Im Falle von langlebigem Code, beziehungsweise Implementierungen von Algorithmen, kann das Leistungsverhalten durch das Einsetzen der Assertions stetig kontrolliert werden. Auf diese Weise kann ebenfalls erreicht werden, dass die (allgemeinen) Leistungserwartungen nach einer Portierung des Algorithmus auf eine neue Technologie oder ein anderes Programmierparadigma, beispielsweise durch Parallelisierung, noch gelten. Voraussetzung hierfür ist, dass die Assertions unabhängig von der verwendeten Technologie formuliert sind.

Aus wissenschaftlicher Sicht sind Performance Assertions interessant, da sie die zielgerichtete Validierung aufgestellter Performance Modelle erlauben. Ein Performance Modell versucht das Laufzeitverhalten einer Anwendung anhand von System- und Eingabeparametern auszudrücken.

[1] **G**eneral **P**urpose Computation on **G**raphics **P**rocessing **U**nit

1.2 Problemstellung

Das Ziel dieser Arbeit ist die Implementierung des Performance Assertion Konzepts mit Metriken für das Message Passing Interface (MPI) [7] anhand des Compilerframeworks ROSE[2] [20]. Eine Performance Assertion soll als Pragma-Anweisung innerhalb des C/C++ Quellcodes formuliert werden. Ein spezieller Compiler liest diese Anweisungen ein und instrumentiert das Programm entsprechend durch Aufrufe einer Laufzeitumgebung, welche während der Programmausführung die für eine Assertion benötigten Metriken erfasst und auswertet.

Dementsprechend umfasst die Umsetzung ein breites Themenspektrum, angefangen bei der Definition der Assertion-Sprache anhand bestehender Konzepte sowie der Festlegung der Metriken, die für die Betrachtung des Zeitverhaltens einer MPI Applikation entscheidend sind. Anschließend gilt es einen Compiler zu erstellen, der durch Manipulation und Analyse des Abstrakten Syntaxbaumes (AST) ein modifiziertes Programm erzeugt, das die im Quelltext formulierten Assertions validiert. Hierbei wird eine entsprechende Laufzeitumgebung benötigt, die in der Lage ist, die anfallenden Metriken zu erfassen und die Auswertungsergebnisse der Assertions zu verwalten.

1.3 Gliederung

Zu Beginn der Arbeit werden in Kapitel 2 relevante Untersuchungen und Veröffentlichungen der Forschung zusammengefasst. Im Anschluss wird in Kapitel 3 die Assertion-Sprache und die verfügbaren Metriken vorgestellt. Darauf aufbauend wird in Kapitel 4 die eigentliche Implementierung erläutert und in Kapitel 5 anhand der `su3_rmd` Applikation der MIMD Lattice Computation (MILC) Collaboration evaluiert. Abschließend erfolgt eine Zusammenfassung der Arbeit und Ausblick auf weitere Möglichkeiten in Kapitel 6.

[2] Homepage `http://rosecompiler.org/`, zuletzt geprüft am 4.11.13

2 Stand der Forschung

Innerhalb dieses Kapitels wird ein Überblick über verschiedene Verfahren zur automatischen Validierung des Laufzeitverhaltens gegeben. Darunter sind Performance Assertions sowie Ansätze, die das Programmverhalten auf entsprechende Muster untersuchen oder die Leistung eines Programmes auf eine Metrik reduzieren.

Da die Erfassung entsprechender Metriken zur Auswertung der formulierten Erwartungen ein wichtiger Aspekt ist, wird im zweiten Abschnitt des Kapitels eine bestehende Messinfrastruktur exemplarisch auf ihre Anwendbarkeit für diese Performance Assertion Implementierung untersucht.

2.1 Automatische Validierung der Performance

Performance Assertion Checking

Die Idee, die Performance eines Programmes automatisch zu validieren, wurde unter anderem bereits von Perl und Weihl [19] aufgegriffen und führte zu PSpec, einer Sprache und Sammlung von Werkzeugen zum Schreiben und Auswerten von Erwartungen an die Performance-Eigenschaften einer Anwendung anhand von Messprotokollen und Event Logs. Es handelt sich hierbei um einen postmortem Ansatz, da die Auswertung nach der Ausführung des Programmes erfolgt. Der Benutzer ist dafür verantwortlich, die zu untersuchende Anwendung so zu erweitern, dass sie für jede Ausführung die zu untersuchenden Ereignisse in einem Protokoll festhält. Sofern dieses entsprechende Informationen, wie zum Beispiel Thread-IDs enthält, sind auch Aussagen über nebenläufige Systeme möglich.

Die eigentlichen Erwartungen werden in der deskriptiven Sprache PSpec formuliert. Hierzu muss eine Spezifikation geschrieben werden, in der zunächst die vorhandenen Ereignisse mit ihren möglichen Attributen deklariert werden. Anhand mehrerer Ereignisse wird ein Intervalltyp deklariert, der von den Ereignissen und ihren Attributen Metriken ableitet. Bei den Erwartungen handelt es sich um Prädikate, beziehungsweise boolesche Ausdrücke, die mit Hilfe der Definitionen eine Aussage über eine Menge von Intervallen treffen. Für die Faltung von Mengen werden verschiedene Operatoren bereitgestellt. Darunter sind unter anderem: Logisches Und, Addition und Durchschnitt.

Die bereitgestellten Werkzeuge umfassen eine Prüfungskomponente, die ermittelt, ob ein Messprotokoll eine Spezifikation erfüllt, einen interaktiven Auswerter von PSpec Ausdrücken und einen Parameterschätzer, der mithilfe eines Messprotokolls versucht, die Werte für Konstanten in einer Spezifikation zu ermitteln.

Asserting Performance Expectations

Einen wichtigen Beitrag für das in dieser Bachelor-Thesis umgesetzte Verfahren liefert die Arbeit von Vetter und Worley [25], die das prinzipielle Konzept von Performance Assertions und eine entsprechende Laufzeitumgebung vorgestellt haben. Ihre Untersuchung berücksichtigt allerdings nur serielle Programme und deren Leistungsmetriken; parallele Programme und die für deren Charakterisierung notwendigen Metriken werden nicht behandelt.

In dem vorgestellten Ansatz wird eine Performance Assertion durch einen Funktionsaufruf zu Beginn und am Ende des zu überwachenden Bereichs innerhalb des Quellcodes realisiert. Mit dem ersten Aufruf werden notwendige Parameter, wie der zu validierende Ausdruck als Zeichenkette und eventuelle Adressen von auszulesenden Programmvariablen, an die Laufzeitumgebung übergeben, welche die zu überwachenden Metriken ermittelt und deren Messung initiiert. Am Ende des Abschnittes erfolgt durch den zweiten Funktionsaufruf die Auswertung der Assertion, indem die gesammelten Werte herangezogen werden. Falls die Aussage nicht zutrifft, kann zusätzlich zur Protokollierung der Anzahl der Fehlschläge eine benutzerdefinierte Funktion ausgeführt werden.

Außerdem ist das vorgestellte Konzept in der Lage, aufgrund von Messungen zur Laufzeit zwischen alternativen Codeabschnitten denjenigen auszuwählen, der einen Ausdruck über mehrere Aufrufe hinweg minimiert [25, S. 9f.]. Das System ermittelt weiterhin für die im Rahmen einer Aussertion beobachteten Werte Statistiken, darunter Minimum und Maximum. Im Gegensatz zu PSpec findet die Validierung während der Laufzeit des Programmes statt. Durch das Parsing des Ausdrucks zur Laufzeit kommt es zu einem Overhead.

Für die Autoren stellt die Integration der Performance Assertions in einen Compiler aufgrund der sich bietenden Möglichkeiten eine interessante Alternative zu ihrer bisherigen Herangehensweise dar. So könnte der Auswertungsausdruck der Assertion von Optimierungen profitieren und durch das vom Compiler ermittelte semantische Wissen des Quellcodes könnten automatisch Assertions erstellt werden [25, S. 5,7].

Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments

Eine alternative Vorgehensweise zeigen Hoffmann et al. [9] auf, indem sie das Leistungsverhalten einer Anwendung durch eine Herzschlagrate abstrahieren. Der Benutzer instrumentiert sein Programm, indem er an ausgewählten Stellen im Programmcode eine Funktion aufruft, die zur Darstellung des Fortschritts einen Herzschlag simuliert. Eine Laufzeitumgebung protokolliert Zeitpunkt, Thread-ID und einen optional vom Benutzer wählbaren Tag, der die Unterscheidung von Ereignissen erlaubt. Zu Beginn des Programmes muss die Laufzeitumgebung über einen Funktionsaufruf initialisiert werden, dabei wird die Anzahl an Herzschlägen festgelegt, die zwischengespeichert und für die Berechnung einer Durchschnittsfrequenz verwendet wird. Durch eine weitere Funktion kann ein minimaler und maximaler Erwartungswert für die Frequenz festgelegt werden. Ein externer Prozess kann diese Werte und die Herzschlagrate abfragen und sich oder den ausführenden Rechner an das ermittelte Leistungsverhalten der Anwendung anpassen.

Die Laufzeitumgebung wurde zweimal auf unterschiedliche Arten implementiert: Einmal basierend auf Dateien, sodass auf die Informationen von mehreren Endgeräten über ein Netzwerk zugegriffen werden kann und ein weiteres Mal unter der Verwendung von Shared Memory, auf den mehrere Prozesse auf einem Rechner zugreifen können.

Bei dieser Reduktion des Laufzeitverhaltens auf eine abstrakte Metrik werden andere charakteristische Eigenschaften, wie das Cacheverhalten oder der Synchronisations- beziehungsweise Kommunikationsoverhead, nicht direkt betrachtet.

Capturing Performance Knowledge for Automated Analysis

Um die automatische Performance-Analyse von durch Instrumentierung gesammelten Daten zu verbessern, integrierten Huck et al. [10] die quelloffene Compiler-Infrastruktur OpenUH [16] mit PerfExplorer [11], einem Framework zur Performance-Analyse von Informationen anhand von Data Mining Techniken. Die für die Analyse benötigten Informationen werden durch andere Werkzeuge gesammelt. Das Framework wurde dabei um eine Skriptschnittstelle und das Inferenzsystem von JBoss Rules [2] zur Verarbeitung von Regeln erweitert. Die Skriptschnittstelle erlaubt unter anderem das Ableiten von neuen Metriken und die Automatisierung der Performance-Analyse sowie der Datenverarbeitung. Weiterhin wurde OpenUH um ein Modul zur compilerbasierten Instrumentierung erweitert. Die Instrumentierung von MPI Prozeduren erfolgt über das Profiling Interface von MPI (PMPI).

Im Unterschied zu Performance Assertions formuliert der Benutzer selbst keine Erwartungen an einzelne Codestellen, die es zu validieren gilt. Stattdessen versucht PerfExplorer automatisiert durch ein Skript und Regeln bestimmte Performance-Eigenschaften des Programmes abzuleiten. Während der Ausführung eines Programmes, welches mit OpenUH instrumentiert und kompiliert wurde, werden Daten gesammelt, die post mortem automatisch mit PerfExplorer anhand eines vom Benutzer geschriebenen Skripts und entsprechender Regeln analysiert werden. Als Ergebnis erhält der Benutzer Verbesserungsvorschläge; Codeänderungen müssen manuell durchgeführt werden.

Das Forschungsziel von Huck et al. ist die Integration der von PerfExplorer gewonnenen Erkenntnisse in das kostengesteuerte Optimierungssystem von OpenUH, um die Performance der Applikation zu verbessern.

2.2 Messsystem

Für die Validierung formulierter Performance Assertions wird ein Messsystem benötigt, welches in der Lage ist, die erforderlichen Metriken für die einzelnen unabhängigen Assertions mit möglichst geringem Overhead zur Laufzeit des Programmes zu erfassen, damit Verzerrungen des ursprünglichen Verhaltens gering ausfallen. Weiterhin muss sich das System in die Laufzeitumgebung integrieren lassen, um eine effiziente Auswertung der einzelnen Assertions zu ermöglichen.

In diesem Zusammenhang ist Score-P [18, 14] zu nennen, ein vereinheitlichtes Performance-Messsystem, das als gemeinsame Grundlage für die Datenerfassung mehrerer Optimierungstools entwickelt wurde und somit den aktuellen Stand der Technik darstellt. Daten können sowohl mittels Tracing als auch Profiling erfasst werden. Unterstützt wird neben dem parallelen Entwicklungsparadigma des Nachrichtenaustauschs über MPI auch Thread-basierte Parallelisierung über OpenMP. Die Instrumentierung eines Programmes erfolgt durch einen Compiler oder durch manuell platzierte Messpunkte sowie im Falle von MPI durch das PMPI Interface. Die Daten können für eine post mortem Analyse gesammelt werden oder aber über die „Online Access“ Schnittstelle zur Laufzeit mittels TCP/IP an ein externes Programm übertragen werden. In diesem Modus ist allerdings nur ein Zugriff auf die gemessenen Profiling-Daten möglich.

TAU (Tuning and Analysis Utilities) [22] ist eine Sammlung von Werkzeugen für die Analyse von parallelen Programmen durch Profiling und Tracing, erlaubt im Unterschied zu Score-P aber auch die Instrumentierung der Java Virtuellen Maschine, des Python Interpreters sowie kompilierter oder laufender Programme. Die Schnittstelle für die manuelle Platzierung der Messpunkte unterstützt ebenfalls Java und Python. Score-P ist in der Lage, TAU für die Instrumentierung des Programmes zu nutzen, sodass die eigene Messinfrastruktur verwendet wird [18]. Gesammelte Profiling Daten kann das Programm über eine Schnittstelle abfragen, auf Tracing Daten kann „online“ zugegriffen werden. Zu den Analysewerkzeugen von TAU gehört unter anderem das von Huck et al. entwickelte und erweiterte PerfExplorer.

3 Die Assertion-Sprache

Die Sprache, in der die Erwartungen an das Laufzeitverhalten formuliert werden, ist eine zentrale Komponente innerhalb des Systems, da durch ihre Syntax und Semantik die verfügbaren Features festlegt werden und damit entschieden wird, ob und auf welche Art und Weise Ausdrücke zur Charakterisierung der Laufzeit überprüfbar sind. Hierzu gehört ebenfalls die Definition der Metriken, auf die der Benutzer zurückgreifen kann. Basierend sowohl auf dem Sprachvorschlag der Aufgabenstellung dieser Arbeit, als auch auf Konzepten von Vetter und Worley [25], wie beispielsweise dem Zugriff auf Variablen der Applikation, wurde die im folgenden vorgestellte Performance Assertion Sprache entwickelt.

3.1 Eigenschaften

Eine wichtige Eigenschaft einer Programmiersprache ist ihre Verständlichkeit und Erlernbarkeit für den Benutzer, da auf diese Weise die Produktivität erhöht und die Wartung bestehender Assertions vereinfacht wird. Um dies zu erreichen, sollte die Syntax entsprechend gestaltet werden und Ähnlichkeiten zu bestehenden Programmiersprachen, wie C oder Java aufweisen. Damit geht die Existenz der standardmäßigen logischen, relationalen und arithmetischen Operatoren in Infixnotation einher, sodass Aussagen verknüpft, Werte verglichen sowie einfache Berechnungen durchgeführt werden können. Da die Assertion-Sprache durch den Compiler in C Code überführt wird, ist eine Nähe zur Zielsprache aus Entwicklersicht vorteilhaft.

Für die Formulierung von Performance Modellen werden grundlegende mathematische Funktionen zur Umsetzung von Quadratwurzel, Betrag und vergleichbaren Operationen benötigt. Da für eine Modellierung der Performance der Zustand des Programmes, beispielsweise die Problemgrößen, entscheidend sein kann, sollte die Sprache den Zugriff auf Programmvariablen erlauben. Die Verwendung von Platzhaltern, deren Wert beim Programmstart dynamisch festgelegt werden kann, ohne dass eine erneute Kompilierung erfolgen muss, ermöglicht unter anderem die flexible Verwaltung architekturspezifischer Werte zwischen verschiedenen Systemen oder Ausführungen.

3.2 Syntax

Als Grundlage für die Ausdrücke der Assertion-Sprache dient die Syntax der Programmiersprache C. Diese wurde allerdings angepasst, um den obigen Eigenschaften und dem allgemeinem Verwendungszweck gerecht zu werden. Die Form der Grammatik basiert auf den Parser Bausteinen des AstFromString[1] Namespace der ROSE Compiler-Infrastruktur und ist nachfolgend in Erweiterter Backus-Naur-Form (EBNF) angegeben:

[1] Dokumentation `http://rosecompiler.org/ROSE_HTML_Reference/namespaceAstFromString.html`, zuletzt geprüft am 7.10.13

```
PA_Pragma                = "perfAssertion", PA_Assumptions ;
PA_Assumptions           = PA_LogicalExp0 ;
PA_LogicalExp0           = PA_LogicalExp1, { "|", PA_LogicalExp1 }
                         | PA_LogicalExp1, { "->", PA_LogicalExp2 } ;
PA_LogicalExp1           = PA_Assumption, { "&", PA_Assumption } ;
PA_Assumption            = ["!"], "(", PA_Assumptions, ")" | PA_RelationalExp ;
PA_RelationalExp         = PA_MathExp, PA_RelationalOperator, PA_MathExp ;
PA_RelationalOperator    = "<=" | "<" | ">=" | ">" | "!=" | "==" ;
PA_MathExp               = PA_AdditiveExp ;
PA_AdditiveExp           = PA_MultiplicativeExp, { ("+" | "-"), PA_MultiplicativeExp } ;
PA_MultiplicativeExp     = PA_UnaryMinusExp, { ("*" | "/"), PA_UnaryMinusExp } ;
PA_UnaryMinusExp         = ["-"], PA_Atom ;
PA_Atom                  = PA_Number | PA_Identifier
                         | PA_ConfigValueReference | PA_VariableReference
                         | PA_BuiltinFunction
                         | "(", PA_MathExp, ")" ;
PA_Number                = ((PA_Digit - "0"), { PA_Digit } | "0"), [ ".", PA_Digit, { PA_Digit }] ;
PA_Digit                 = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "8" | "9" ;
PA_Identifier            = "MPITime" | "MPIPointToPointTime" | "MPICollectiveTime"
                         | "MPIWaitTime" | "MPITransferTime"
                         | "WallTime"
                         | "seconds" | "milliseconds" | "microseconds" ;
PA_ConfigValueReference  = "${", CONFIG_VALUE_NAME, "}" ;
PA_VariableReference     = "$", VARIABLE_NAME ;
PA_BuiltinFunction       = ("exp" | "log" | "sqrt" | "abs"), "(", PA_MathExp, ")"
                         | "pow", "(", PA_MathExp, ",", PA_MathExp, ")"
                         | "nMPIProcesses", "(", PA_VariableReference, ")" ;
```

Aus der Grammatik geht hervor, dass jede Performance Assertion mit dem Schlüsselwort „`perfAssertion`“ beginnt. Weiterhin steigt die Operatorpriorität vom Anfang bis zum Ende der Definition, sodass die logischen Operatoren die niedrigste Priorität besitzen und das unäre Minus die höchste. Die Klammerung von Ausdrücken erlaubt eine manuelle Anpassung der Rangfolge. Der logische Operator „&“ besitzt eine höhere Priorität als „|“ oder „->“, genauso wie die arithmetischen Operatoren „*“ und „/“ gegenüber „+“ und „-“. Das Format der Zahlen orientiert sich an einfachen positiven Dezimalzahlen. Die Nichtterminale `CONFIG_VALUE_NAME` und `VARIABLE_NAME` bleiben formal unterspezifiziert, beide entsprechen den gültigen Variablennamen innerhalb der Programmiersprache C. Die Beschränkung für die Namen der Konfigurationswerte wurde eingeführt, um das Parsing in der Implementierung zu vereinfachen.

3.3 Semantik

Die Semantik der Assertion-Sprache sei wie folgt definiert:

Platzierung und Gültigkeitsbereich: Eine Performance Assertion wird als C/C++ Compiler-Pragma realisiert und bezieht sich auf einen direkt folgenden Basic Block oder eine Schleife. Bei einem Basic Block handelt es sich um einen mit „`{ ... }`“ definierten Sichtbarkeitsbereich. Die Erfassung der zur Auswertung notwendigen Daten beginnt und endet mit diesem Bereich. Im Falle einer direkt folgenden Schleife bezieht sich die Assertion auf alle Iterationen, die ausgeführt werden. Um eine Erwartung für jede Schleifeniteration validieren zu lassen, muss der Schleifenkörper als Basic Block annotiert werden. Sofern der Gültigkeitsbereich einer Assertion ein `continue` oder `break` Statement aufweist, wird eine auf diese Art und Weise unterbrochene Schleifeniteration mit ausgewertet. Weiterhin ist das Verhalten beim Annotieren des äußeren Basic Block einer Funktion nicht definiert.

Kontrollflussänderungen durch `goto` Statements sowie C++ Exceptions werden nicht unterstützt und können zu nicht definiertem Verhalten der Implementierung führen, sofern durch ihre Verwendung der Gültigkeitsbereich der Assertion verlassen wird. Hierbei handelt es sich um eine Vereinfachung für die Implementierung, da beim Verlassen des Bereichs die benötigten Messungen beendet werden und die Auswertung der Assertion erfolgt. Ein Überspringen dieser Phase kann die Laufzeitumgebung in einen ungültigen Zustand versetzen, wenn gestartete Messungen nicht beendet werden. Die Unterstützung der Kontrollflussanweisungen erfordert daher ein gesondertes Vorgehen, auf das im Rahmen dieser Arbeit aus folgenden Gründen verzichtet wurde:

- Eine `goto` Anweisung kann unter Umständen durch eine Schleife ersetzt oder das Problem durch die Wahl eines anderen Gültigkeitsbereich für die Assertion umgangen werden, so dass der Sprung den Bereich nicht verlässt.

- Eine Unterstützung von C++ Exceptions erfordert, dass die Implementierung aktiv zwischen C und C++ Programmen unterscheidet, da in einem normalen C Programm die hierfür benötigten C++ Sprachmechanismen nicht verwendet werden können, sofern die Kompilierung nicht mit einem entsprechenden Compiler erfolgt.
- Eine Exception stellt eine Ausnahme im Programmfluss dar und ist als solches kein elementares Konzept bei der Durchführung von Berechnungen.

Logische Operatoren: Die logischen Operatoren „&“ und „|“ entsprechen dem logischem Und (Konjunktion) sowie logischem Oder (Disjunktion). „->“ beschreibt die aussagenlogische Implikation, wobei $a \rightarrow b$ zu $\neg a \vee b$ äquivalent ist.

Relationale und arithmetische Operatoren: Die Bedeutung der Operatoren „!“, „<=“, „<“, „>=“, „>“, „!=“, „==“, „+“, „-“ und „*“ ist durch die gleichnamigen Operatoren in C/C++ und deren Semantik gegeben. Der Divisionsoperator „/“ führt eine Fließkommadivision durch, selbst wenn beide Operanden ganzzahlig sind.

Zahlen: Eine Zahl ohne dezimalen Anteil, das heißt ohne „.“, wird als vorzeichenbehafteter, ganzzahliger Wert interpretiert. Dezimalzahlen werden als Fließkommazahlen doppelter Genauigkeit realisiert. Falls die Operanden einer binären Operation einen unterschiedlichen Typ aufweisen, so ist das Ergebnis eine Fließkommazahl. Negative Zahlen werden durch das unäre Minus gebildet.

Variablenzugriff: Durch „`$VARIABLE_NAME`“ kann der Wert der Variable `VARIABLE_NAME` ausgelesen werden. Die Auswertung erfolgt am Ende des Gültigkeitsbereichs der Performance Assertion. Folglich beeinflussen Schreibvorgänge auf die Variable innerhalb des zu erfassenden Codebereichs den ausgelesenen Wert. Voraussetzung ist, dass `VARIABLE_NAME` innerhalb des Scopes der Assertion, der durch das definierende Compiler-Pragma gegeben ist, einen validen Variablenbezeichner oder ein Präprozessor-Makro darstellt. Letzteres erlaubt die Verwendung von MPI Konstanten, wie beispielsweise `MPI_COMM_WORLD`. Der Benutzer ist dafür verantwortlich, dass die Variablentypen implizit in sinnvolle Zahlen konvertiert werden können.

Zugriff auf Konfigurationswerte: Durch „`${CONFIG_VALUE_NAME}`“ kann auf den Wert zugegriffen werden, der unter dem Namen `CONFIG_VALUE_NAME` in einer Konfigurationsdatei abgelegt wurde. Die Konfigurationsdatei wird beim Programmstart eingelesen, wobei alle Werte als Fließkommazahl doppelter Genauigkeit interpretiert werden. Siehe 4.1.2 für eine genauere Beschreibung dieses Features. Ein nicht vorhandener Eintrag wird zur IEEE-754 Konstante `NaN` ausgewertet.

eingebettete Funktionen: Um die Formulierung von erweiterten Performance Modellen zu ermöglichen, erlaubt die Assertion-Sprache die Verwendung einiger vordefinierter Funktionen. Die mathematischen Funktionen „`exp`“ für die natürliche Exponentialfunktion, „`log`“ für den natürlichen Logarithmus, „`sqrt`“ für die Quadratwurzelfunktion, „`abs`“ für den Betrag und “`pow`“ für das Potenzieren liefern als Ergebnis Fließkommazahlen doppelter Genauigkeit.

Eine Besonderheit stellt die Funktion „`nMPIProcesses`“ dar, die für einen gegebenen MPI Kommunikator die Anzahl der Prozesse zurückgibt. Zu beachten ist, dass für diese Funktion kein allgemeiner mathematischer Ausdruck, sondern eine Variablenreferenz vom Typ `MPI_Comm` erwartet wird.

Konstanten und Metriken: Bei einem Identifier handelt es sich um einen Bezeichner für eine Konstante oder Metrik. Die Assertion-Sprache stellt die Konstanten „`seconds`“, „`milliseconds`“ und „`microseconds`“ bereit, die in Verbindung mit den Metriken genutzt werden können, um die Einheit einer Zeitdauer von Sekunden, Millisekunden oder Mikrosekunden zu verdeutlichen und die Zeitdauer unabhängig von der Auflösung des internen Zeitgebers des Messsystems zu formulieren. Da alle Metriken als vorzeichenlose 64 Bit Ganzzahlen verarbeitet werden und der Zeitgeber der Implementierung eine Auflösung von Nanosekunden besitzt (siehe 4.1.1), expandiert beispielsweise „`milliseconds`“ zu 10^6, da eine Sekunde 10^9ns oder 1000ms entspricht.

Die anderen Bezeichner repräsentieren die in dieser Arbeit verfügbaren Metriken. Bei „`WallTime`“ handelt es sich um die real verstrichene Zeit, die für die Bearbeitung des Gültigkeitsbereich der Assertion benötigt wurde. „`MPITime`“ steht für die akkumulierte Zeit aller MPI Funktionen. Mit „`MPIPointToPointTime`“, „`MPICollectiveTime`“ und „`MPIWaitTime`“ existieren jeweils Metriken, die sich nur auf eine bestimmte Teilmenge der MPI Funktionen beziehen. „`MPIPointToPointTime`“ bezieht sich auf alle Funktionen der Punkt-zu-Punkt Kommunikation, „`MPICollectiveTime`“ auf alle kollektiven Operationen und „`MPIWaitTime`“ auf alle Wartefunktionen für nichtblockierende Kommunikation.

MPI bietet keinen Mechanismus, um die Übertragungszeit der Punkt-zu-Punkt Kommunikation zu bestimmen. Für blockierende Kommunikation entspricht die Zeit des Prozeduraufrufs nicht der Übertragungszeit, da gegebenenfalls auf eine passende Empfangsoperation gewartet werden muss oder im Standard-Modus die Nachricht

intern gepuffert werden kann [7]. Im Falle von nichtblockierender Kommunikation ist es zudem nicht ohne größeren Aufwand möglich, den Zeitpunkt zu bestimmen, zu dem die Operation abgeschlossen ist. Eine Möglichkeit hierzu ist die wiederholte Abfrage (Polling) des Kommunikationsstatus in einem separaten Thread über `MPI_Request_get_status` oder der Nachrichtenwarteschlangen der MPI Implementierung über die von Cownie und Gropp [6] entwickelte Debugger Schnittstelle. Dabei ist die Übertragungszeit für nichtblockierende Kommunikation eine interessante Kenngröße, um den „Overlap“, die Überlappung von Kommunikation und Berechnung, zu quantifizieren. Eine solche Überlappung kann die Laufzeit eines Programmes verbessern [4]. „`MPITransferTime`“ ist eine geschätzte Metrik, die versucht die Zeit, welche für die Übertragung der Punkt-zu-Punkt Kommunikation anfällt, durch eine lineare Funktion der Nachrichtengröße zu approximieren. Die lineare Funktion ergibt sich aus einer Latenz in Mikrosekunden und einer Datenübertragungsrate in MBit/s. Beide Werte können durch Einträge in einer Konfigurationsdatei angepasst werden, standardmäßig werden $1\mu s$ Latenz und eine Übertragungsrate von 100 MBit/s verwendet.

Die `MPIWaitTime` Metrik berücksichtigt die Funktionen `MPI_Wait`, `MPI_Waitall`, `MPI_Waitany` und `MPI_Waitsome`. Die Zugehörigkeit der MPI Funktionen zu den `MPIPointToPointTime` und `MPICollectiveTime` Metriken folgt der Gruppierung der Funktionen für die C Sprachanbindung im MPI Standard 3.0 [7, S. 685ff. und S. 689ff.]. Daraus folgt, dass die `MPIWaitTime` Metrik eine Teilmenge der `MPIPointToPointTime` Metrik ist. Eine denkbare Alternative wäre gewesen, den beiden Metriken der Kommunikationstypen nur die Funktionen zuzuweisen, die eine Kommunikation initiieren, sodass Funktionen wie `MPI_Get_count` nicht miteinbezogen werden. Eine solche Entscheidung ist jedoch abhängig von der Intention des Benutzers. Im Falle der kollektiven Funktion `MPI_Reduce_local`, die eine lokale Reduktion auf den Argumenten ausführt, wird dies deutlich: Der Benutzer kann sowohl an der eigentlichen Kommunikationszeit interessiert sein, die für diese Funktion 0 ist oder aber an der Zeit der MPI Implementierung, wenn er erwägt, eine effizientere Implementierung der Reduktion einzusetzen, beispielsweise durch die Verwendung von Threads. Aus diesem Grund wird die formale Aufteilung des Standards verwendet. Um zusätzlichen Verwaltungsaufwand zur Laufzeit zu vermeiden, erfolgt die Gruppierung der MPI Funktionen während des Buildvorgangs (siehe 4.2) und ist daher nicht einstellbar.

4 Implementierung

Die Implementierung der Performance Assertions erfolgt maßgeblich unter Verwendung des Compilerframeworks ROSE [20], das die Programmierung benutzerdefinierter Compiler erlaubt. Hierzu wird der zu kompilierende Quellcode durch das Frontend in einen Abstrakten Syntaxbaum (AST) überführt und Funktionalitäten zu dessen Analyse und Modifikation bereitgestellt. Der transformierte AST wird am Ende wieder in Quelltext umgewandelt und mit dem Compiler des Systems übersetzt.

Weiterhin werden die Boost[1] Bibliotheken sowie die MPI Implementierungen MPICH[2] und OpenMPI[3] eingesetzt.

Die Umsetzung ist in drei Projekte unterteilt, den Compiler, die Laufzeitumgebung und den Generator:

- Der Compiler erstellt anhand der als Pragma formulierten Performance Assertions eine instrumentierte Variante des Eingabeprogrammes, die bei ihrer Ausführung unter Verwendung der Messbibliothek die Assertions auf ihre Gültigkeit hin überwacht.
- Die Aufgabe der Laufzeitumgebung ist die Erfassung der Metriken sowie die Verwaltung der Auswertungsergebnisse der Assertions.
- Der Generator generiert für die nicht explizit implementierten MPI Funktionen die zur Messung benötigten Methoden automatisch.

Abbildung 4.1 zeigt die Interaktion der einzelnen Komponenten der Implementierung.

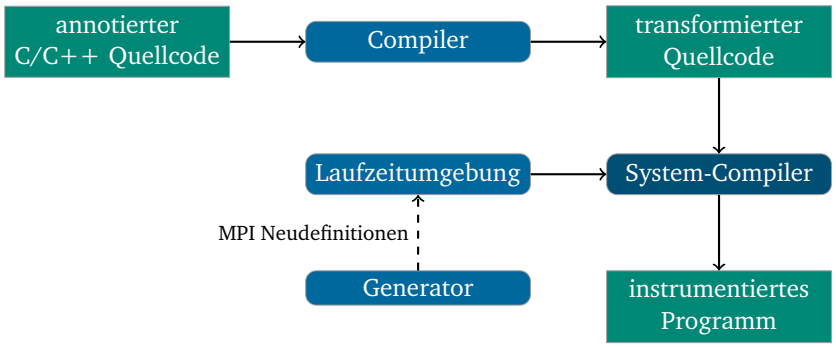

Abbildung 4.1: Interaktion der Komponenten

Um den Kompilierungsvorgang möglichst einfach zu gestalten, wird als Buildsystem CMake[4] genutzt. Auf diese Weise wird während dem Buildvorgang der Generator ausgeführt und dessen Ausgabe für die Kompilierung der Laufzeitumgebung verwendet.

Aufgrund der oben genannten Abhängigkeiten und der Verwendung von Funktionen aus dem POSIX Standard [12], ist diese Implementierung gegebenenfalls nicht auf allen Systemen lauffähig.

4.1 Laufzeitumgebung

Bei der Laufzeitumgebung handelt es sich um eine statische Bibliothek, gegen die alle vom Compiler instrumentierten Programme gelinkt werden müssen. Sie stellt das Messsystem zur Erfassung der für die Validierung der Assertions benötigten Daten bereit und verwaltet die Auswertungsergebnisse. Die Deklarationen der Funktionen, die als Interface den externen Zugriff auf die Funktionalitäten der Laufzeitumgebung erlauben, finden sich zentral im Header `pa_runtime.h`. Die Umgebung selbst gliedert sich in das Konfigurationssystem, den `MPIController` und den `AssertionCollector`, welche zur Strukturierung und Vermeidung von Namenskonflikten als Namespaces realisiert sind.

[1] Homepage `http://www.boost.org/`, zuletzt geprüft am 11.10.13
[2] Homepage `http://www.mpich.org/`, zuletzt geprüft am 11.10.13
[3] Homepage `http://www.open-mpi.org/`, zuletzt geprüft am 11.10.13
[4] Homepage `http://www.cmake.org/`, zuletzt geprüft am 11.10.13

Die in 2.2 vorgestellten Messsysteme unterstützen Profiling und Tracing zur Erfassung der benötigten Daten. Eine Auswertung der Performance Assertions zur Laufzeit des Programmes kann im Falle von Score-P nur über die Online Access Schnittstelle erfolgen. Dies setzt jedoch die Auslagerung der Auswertung in einen separaten Prozess oder Thread voraus, was mit zusätzlichem Kommunikationsaufwand verbunden ist und die übrige MPI Kommunikation beeinflussen kann. Weiterhin ist nicht sichergestellt, dass die benötigten Daten für den Gültigkeitsbereich einer Assertion aus den Profiling-Ergebnissen extrahiert werden können. Bei einer Verwendung von Score-P oder TAU ist zudem davon auszugehen, dass das Potential für eigene Messungen und Metriken gering ist: Eine Anpassung oder Erweiterung der auf Profiling und Tracing spezialisierten Systeme ist als komplex einzustufen, da sie nicht für diesen Verwendungszweck entworfen wurden. Aus diesen Gründen wird in dieser Arbeit keines der genannten Messsysteme verwendet, sondern die Messung der Metriken eigenhändig implementiert.

4.1.1 Zeitgeber

Für alle Zeitmessungen innerhalb der gesamten Laufzeitumgebung wird die POSIX Funktion `clock_gettime` [12, S. 667] mit dem Taktgeber `CLOCK_MONOTONIC` verwendet. Diese konsistente Nutzung verringert den Wartungsaufwand und verbessert die Vergleichbarkeit der gewonnenen Werte, da bei unterschiedlichen Verfahren zu erwarten ist, dass der Overhead für die Bestimmung stärker voneinander abweicht. `CLOCK_MONOTONIC` liefert die seit einem unspezifizierten Zeitpunkt in der Vergangenheit verstrichene monotone Zeit in Sekunden und Nanosekunden und ist daher nur minimalen Änderungen zur Laufzeit unterworfen.

Um die für eine Operation benötigte Zeit zu erfassen, wird `clock_gettime` vor und nach dieser Operation aufgerufen und die Differenz der Werte gebildet. Die Speicherung der ermittelten Zeitwerte erfolgt in einer vorzeichenlosen 64 Bit breiten Ganzzahl in Nanosekunden. Der POSIX Standard garantiert für `CLOCK_MONOTONIC` lediglich eine Auflösung von 20ms [12, S. 273], der exakte Wert und die erreichbare Genauigkeit ist von der Implementierung im Kernel des Betriebssystem und gegebenenfalls dem Vorhandensein eines High Precision Event Timer (HPET) abhängig. Bei einem HPET handelt es sich um einen hochpräzisen Taktgeber.

Eine alternative Methode der Zeiterfassung besteht im Auslesen des Time Stamp Counter Registers der CPU. Dieses Register der x86 Architektur zählt die Anzahl der Taktzyklen seit dem letzten Reset. Kennt man die Taktfrequenz, so lässt sich aus der Differenz der Taktzyklen die verstrichene Zeit berechnen. Sofern es sich um einen Mehrkernprozessor handelt und die einzelnen Register nicht miteinander synchronisiert werden oder die Taktrate aufgrund von Stromspar- oder Turbooptionen variabel ist, kann dieser Ansatz zu Problemen führen, da ohne zusätzlichen Aufwand keine konstante Änderung des Zählerregisters garantiert werden kann. Aufgrund dieser Tatsache, der angestrebten Portierung auf verschiedene Rechnerarchitekturen und der in der Praxis erzielten Ergebnisse von `clock_gettime`, verwendet diese Implementierung die am Anfang beschriebene Funktion.

MPI stellt für Zeitmessungen die Funktion `MPI_Wtime` bereit, welche die seit einem unbekannten Punkt in der Vergangenheit verstrichene Zeit in Sekunden als Fließkommazahl doppelter Genauigkeit zurückgibt. OpenMPI verwendet auf einem Linux System der AMD64 Architektur entweder das Time Stamp Counter Register oder die POSIX Funktion `gettimeofday` [12, S. 1082][5], die als veraltet markiert ist. MPICH kann als Zeitgeber unter anderem `gettimeofday` und `clock_gettime` einsetzten[6]. Dementsprechend bietet `MPI_Wtime` keinen Vorteil gegenüber der direkten Verwendung von `clock_gettime`, zumal die Konvertierung des Zeitwertes in Sekunden durch die notwendige Fließkommadivision mit einem für diese Implementierung nicht notwendigem Aufwand verbunden ist.

Zur Realisierung der `WallTime` Metrik wird die Funktion `perfAssertion_getTime`, die einen Aufruf des Zeitgebers und die Konvertierung des Wertes in Nanosekunden kapselt, bereitgestellt.

4.1.2 Konfigurationssystem

Das Konfigurationssystem erlaubt es dem Benutzer, Werte für die mit `${NAME}` verwendeten Variablen in den Assertions dynamisch vor dem Programmstart ohne erneute Kompilierung festzulegen und das Verhalten des Messsystems zu beeinflussen. Hierzu lädt das Konfigurationssystem im Zuge der Initialisierung der Laufzeitumgebung eine Konfigurationsdatei, die mit der Umgebungsvariablen `PA_CONFIG_FILE` spezifiziert wird. Die Initialisierung wird durch den Aufruf der Funktion `perfAssertion_init` im Einstiegspunkt des Programmes veranlasst.

[5] Quelle: Analyse des Quellcodes von Version 1.6.5

[6] Quelle: Analyse des Quellcodes von Version 3.0.4

Das Format der Textdatei ist dabei so aufgebaut, dass in jeder Zeile einem Namen mittels „=“ ein Wert zugewiesen wird. Kommentare werden mit Hilfe von „#“ eingeleitet und können gegebenenfalls eine komplette Zeile ausmachen. Zusätzlich werden leere Zeilen unterstützt, um den Lesefluss zu verbessern.

Die Speicherung der normalen Fließkommawerte, die in einer Performance Assertion verwendet werden, erfolgt mit ihrem Bezeichner in einer Hashtabelle, um für die Auswertung der Assertions einen möglichst schnellen Zugriff zu gewährleisten.

Das Konfigurationssystem kennt folgende Einstellungen, die separat verwaltet werden:

PA_TRANSFER_LATENCY: Die für die Berechnung der `MPITransferTime` Metrik verwendete Latenz in Mikrosekunden. Der Standardwert beträgt $1\mu s$. Intern wird die Latenz in Nanosekunden gespeichert.

PA_TRANSFER_RATE: Die für die Berechnung der `MPITransferTime` Metrik verwendete Übertragungsrate in Mbit/s. Der Standardwert beträgt 100 Mbit/s. Intern wird die Übertragungsrate in Bytes pro Nanosekunde gespeichert.

PA_REPORT_FILE: Der Name der Datei, beziehungsweise deren Präfix, unter dem am Ende des instrumentierten Programmes der Bericht über die Auswertung der Assertions gespeichert wird. An den endgültigen Dateinamen wird im Falle eines MPI Prozesses eine Identifikationsnummer angehängt, die dem Rang des Prozesses unter `MPI_COMM_WORLD` entspricht. Wenn es sich um einen mittels `MPI_Comm_spawn` oder `MPI_Comm_spawn_multiple` zur Laufzeit erstellten Prozess handelt, wird zusätzlich die POSIX Prozess-ID angefügt, um einen möglichst eindeutigen Namen zu generieren, da Kinderprozesse einen separaten `MPI_COMM_WORLD` Kommunikator erhalten [7, S. 377].

Als Standardwert wird „`perfAssertion_`“ gefolgt vom aktuellen Datum zum Zeitpunkt der Berichterzeugung im Format *JahrMonatTag_StundeMinuteSekunde* verwendet.

4.1.3 MPI Metriken

Um die Daten für die Realisierung der MPI Metriken zu erfassen, verwendet die Implementierung das MPI Profiling Interface [7, S. 555-561], welches es erlaubt, die MPI Funktionen neu zu definieren und die ursprüngliche Funktion unter dem Präfix „PMPI“ anstatt „MPI“ zur Verfügung stellt. Auf diese Weise lassen sich unter anderem Funktionsparameter abfangen und die Zeit für die ursprüngliche Funktion erfassen. Die Neudefinitionen für die meisten MPI Funktionen werden automatisch generiert (siehe 4.2). Einige Funktionen, wie zum Beispiel `MPI_Init`, sind hiervon ausgenommen und sind manuell definiert. Das Profiling Interface wird beispielsweise auch von VampirTrace zur Überwachung von MPI Prozeduren eingesetzt [15, S. 143].

Die Erfassung der MPI Metriken muss berücksichtigen, dass mehr als eine Performance Assertion, die die gleichen oder unterschiedliche MPI Metriken für ihre Auswertung benötigt, zur selben Zeit aktiv sein kann und die Erfassung der Metriken nicht unabhängig voneinander ist, da die aktuellste Assertion im Gültigkeitsbereich der vorherigen liegt. Wird zum Beispiel im Gültigkeitsbereich einer Performance Assertion, die eine Aussage über die `MPITime` trifft, eine Prozedur aufgerufen, innerhalb der ebenfalls eine Assertion `MPITime` einsetzt, dann ist die im Rahmen der Prozedur-Assertion ermittelte MPI Zeit ein Anteil der Zeit für die vorherige Assertion. Um diese Zuordnung zu gewährleisten, muss im Falle eines seriell ablaufenden Programmes (ein Thread pro Prozess) für jede MPI Metrik der „Callstack“ der Assertions nachvollzogen werden.

Dementsprechend werden die Werte der Metriken während ihrer Erfassung jeweils in einer Stack-Datenstruktur gespeichert. Durch die Funktion `perfAssertion_enterMPICapture` wird eine neue Messanfrage für eine oder mehrere Metriken initialisiert, indem auf jeden spezifizierten Metrik-Stack der Wert 0 abgelegt wird. Die Metriken werden für den Funktionsaufruf als Zweierpotenz $(2^k, k >= 0)$ kodiert, sodass durch ein binäres Oder die Messung mehrerer Metriken mit einem Funktionsaufruf begonnen werden kann. Die einzelnen, durch den Aufruf von MPI Funktionen anfallenden Werte werden je Metrik auf dem obersten Stack-Element akkumuliert, sofern für die Metrik mindestens eine Messanfrage aktiv ist. Die Funktion `perfAssertion_leaveMPICapture` beendet eine Messanfrage für die spezifizierten Metriken, indem die obersten Werte aus den betroffenen Metrik-Stacks entfernt und in ein übergebenes Array kopiert werden. Falls die Stacks weitere Werte enthalten, so muss jeweils der alte Wert auf die neuen oberen Werte addiert werden. Auf diese Weise werden die oben genannten Anforderungen umgesetzt.

Eine mögliche Erweiterung für Multithreading wird in 6.2 beschrieben.

Berechnung der Transferzeit

Bis auf die `MPITransferTime` basieren alle anderen Metriken auf der Zeit, die für die Ausführung von MPI Funktionen benötigt wird. Bei der Transferzeit handelt es sich dagegen nicht um einen direkt messbaren Wert (siehe 3.3), stattdessen wird diese Metrik aus der Größe der übertragenen Daten in Bytes abgeleitet. Diese lässt sich für die meisten Funktionen der Punkt-zu-Punkt Kommunikation durch Abgreifen der Parameter für den MPI Datentyp und deren Anzahl berechnen. Mit der Größe der übermittelten Daten S und der vom Konfigurationssystem bereitgestellten Latenz L und Übertragungsrate R wird die heuristische Übertragungszeit durch $S/R + L$ berechnet und auf dem entsprechendem Stack akkumuliert. Im Unterschied zu den übrigen MPI Metriken werden diese Zeiten als Fließkommazahl gespeichert, da bei der Rechnung abhängig von den Dimensionen der drei Werte die Nachkommastellen des Ergebnisses entsprechend signifikant sein können.

MPI erlaubt die Erstellung von persistenten Kommunikationsanforderungen zur Vermeidung von Overhead, wenn eine Kommunikationsoperation wiederholt mit identischen Argumenten ausgeführt wird. Eine solche Anforderung wird durch den Aufruf einer entsprechenden Prozedur erstellt, der als Parameter unter anderem der Datentyp und dessen Anzahl übergeben wird. Der Benutzer erhält ein Handle mit dem er durch eine weitere Prozedur, die außer dem Handle keine weiteren Argumente benötigt, den eigentlichen Kommunikationsvorgang starten kann. Für persistente Kommunikation liegen daher die Informationen zur Berechnung der Datengröße zum Zeitpunkt der Ausführung nicht vor. Folglich wird bei der Erstellung der persistenten Kommunikationsanforderung über die Funktion `registerPersistentCommunicationRequest` die Größe berechnet und mit dem zugewiesenen Handle in einer Hashtabelle gespeichert. Hierfür muss es sich jedoch bei `MPI_Request` um einen Pointer oder eine Ganzzahl handeln, was zumindest bei MPICH und OpenMPI der Fall ist. Wird mittels `MPI_Start` oder `MPI_Startall` eine persistente Kommunikationsanforderung ausgeführt, so wird durch die manuell erstellte neue Definition der beiden Funktionen über das Handle die Größe bestimmt und die Transferzeit berechnet. Wenn eine persistente Kommunikationsanforderung mittels `MPI_Request_free` freigegeben wird, wird der Eintrag in der Hashtabelle anhand des Handles gelöscht.

4.1.4 Verwaltung des Assertionzustands

Zusätzlich zur Messung der Metriken übernimmt die Laufzeitumgebung die Verwaltung der Auswertungsergebnisse der Performance Assertions und die Erstellung des Berichts am Ende des Programmes. Für diese Aufgaben ist der `AssertionCollector` verantwortlich, der für eine Assertion die Anzahl an erfolgreichen und insgesamt durchgeführten Auswertungen in einer Hashtabelle speichert. Um eine Performance Assertion eindeutig zu identifizieren, wird der Name der Quellcodedatei sowie die Zeile, in der das die Erwartung formulierende Pragma steht, verwendet. Der Dateiname und die Zeile werden durch den Verbunddatentyp `AssertionKey` modelliert, die beiden Anzahlen durch den Datentyp `AssertionEvaluation`. Das Auswertungsergebnis einer Assertion wird der Laufzeitumgebung durch die Funktion `perfAssertion_reportAssertion` mitgeteilt, woraufhin die Gesamtanzahl an Auswertungen und abhängig von einem Funktionsparameter die Anzahl erfolgreicher Auswertungen für die spezifizierte Assertion inkrementiert wird.

Durch diesen Aufbau werden für jede Assertion eine Zeichenkette und drei Ganzzahlen gespeichert. Der Speicherverbrauch steigt daher linear mit der Anzahl der Performance Assertions in einem Programm und ist nicht abhängig von der Anzahl der Auswertungen einer Assertion. Der gesamte Speicherbedarf der Laufzeitumgebung hängt durch das Messsystem von der Anzahl der gleichzeitig aktiven Assertions ab.

Berichterstellung

Am Ende eines Programmes werden die Auswertungsergebnisse durch die Funktion `perfAssertion_finalizeReport` in eine Textdatei ausgegeben. Der Aufruf der Funktion erfolgt nicht durch Instrumentierung mit dem Compiler, sondern durch die Registrierung als „Exithandler“ mittels `atexit` während der Initialisierung der Laufzeitumgebung. Der Grund hierfür ist, dass eine normale Terminierung eines C/C++ Programmes anstatt über eine einfache Rückkehr aus dem Einstiegspunkt mit `return` auch über den Aufruf der `exit` Funktion an einer beliebigen Stelle erfolgen kann.

Der Name der Ausgabedatei wird vom Konfigurationssystem bestimmt. Gibt der Benutzer einen Ordnerpfad an, so muss dieser vorher manuell erstellt werden. Für jede Assertion wird der Quellcodedateiname, die Zeile sowie die Anzahl der erfolgreichen und der gesamten Auswertungen als Bruch und Prozentzahl ausgegeben. Die einzelnen Einträge werden für eine bessere Übersicht nach dem Namen der Quellcodedatei und im Falle von mehr als einer Assertion pro Datei nach der Zeile sortiert und durch Zeilenumbrüche getrennt. Eine Zeile des Berichtes entspricht daher folgendem Format: *Quellcodedateiname:Zeile -> #erfolgreiche Auswertungen/#gesamte Auswertungen = Prozent*.

4.2 Generator

Bei dem Generator (auch `extractMPI` genannt) handelt es sich um ein Programm auf der Basis von ROSE, das für die meisten MPI Funktionen automatisch die Definitionen für die Laufzeitumgebung erstellt (siehe 4.1.3). Dieses Vorgehen besitzt folgende Vorteile:

- Der aktuelle MPI Standard (3.0) umfasst weit mehr als 200 Funktionen[7]. Eine manuelle Erstellung der Definitionen ist aufwändig und fehleranfällig.
- Durch eine dynamische Generierung der Definitionen wird gewährleistet, dass nur von der installierten MPI Implementierung unterstützte Funktionen berücksichtigt werden. Je nach Version unterscheiden sich die Implementierungen hinsichtlich ihres Fortschritts bei der Umsetzung der Version des MPI Standards. Bei der Verwendung von Definitionen, die Funktionen enthalten, die nicht von der installierten MPI Implementierung unterstützt werden, kann es zu Fehlern beim Kompilieren der Laufzeitumgebung, dem Linken des instrumentierten Programmes oder dem Ausführen von jenem kommen.
- Zukünftige MPI Funktionen werden berücksichtigt und fließen in die `MPITime` Metrik ein, sofern das charakteristische Präfix bestehen bleibt.

4.2.1 Umsetzung

Als Eingabe für den Generator dient eine Quellcodedatei, die benötigte Include-Direktiven für Deklarationen der Laufzeitumgebung und MPI enthält. Zu Beginn werden alle Funktionsdeklarationen innerhalb der Eingabedatei ermittelt; anschließend wird über alle Ergebnisse iteriert. Die Funktionsdeklarationen werden anhand des Funktionsnamens durch die Klasse `MPIFunctionEvaluator` klassifiziert. Eine Funktion, die das Präfix „MPI_“ aufweist wird als MPI Funktion aufgefasst und weiter auf ihre Zugehörigkeit zu folgenden Teilmengen untersucht:

- Wartefunktionen
- Funktionen der Punkt-zu-Punkt Kommunikation
- Funktionen der kollektiven Kommunikation
- Funktionen, die von `MPITransferTime` berücksichtigt werden
- Funktionen, die eine persistente Kommunikationsanforderung erstellen

Folgende Funktionen sind von der automatischen Generierung ausgeschlossen und werden nicht als MPI Funktionen angesehen:

`MPI_Pcontrol`: In der Standardimplementierung führt diese Funktion keinerlei Operationen aus und kehrt sofort zurück. Ihr Zweck besteht darin, einem Benutzer die Möglichkeit zu geben das Verhalten eines Profilers oder Messsystems manuell zu beeinflussen. Für diese Implementierung konnte keine sinnvolle Verwendung festgestellt werden, weswegen die Funktion ignoriert wird.

`MPI_Wtime` und `MPI_Wtick`: Hierbei handelt es sich um zwei Funktionen, die zur Zeitmessung bereitgestellt werden. Folglich sind sie für eine Betrachtung des Kommunikationsverhaltens der Anwendung weniger interessant. Aufgrund dessen und um den Overhead für ihre Nutzung nicht zu erhöhen, werden die beiden Funktionen nicht neu definiert.

`MPI_Init` und `MPI_Init_thread`: Beide Funktionen dienen der Initialisierung von MPI und sind in der Laufzeitumgebung bereits manuell definiert, um die globale Prozess-ID für den Berichtsdateinamen zu generieren.

`MPI_Start`, `MPI_Startall` und `MPI_Request_free`: Diese drei Funktionen sind im Rahmen der Verwaltung von persistenten Kommunikationsanforderungen und der Berechnung der `MPITransferTime` in der Laufzeitumgebung manuell definiert.

Handelt es sich bei der betrachteten Funktion um eine MPI Funktion, so wird eine neue definierende Funktionsdeklaration mit der Signatur der ursprünglichen Funktion erstellt, um Messwerte für die Metriken erheben zu können. Die Neudefinitionen sind wie folgt aufgebaut:

[7] Siehe [7] und die Ausgabe des Generators.

- Es wird überprüft, ob die Daten der Funktion erfasst werden müssen:
  - Wenn zum Zeitpunkt des Aufrufs keine Messanfragen aktiv sind, kann die ursprüngliche Funktion normal ausgeführt werden.
  - Um ein doppeltes Zählen von Zeiten zu vermeiden, wenn bestimmte Funktionalitäten durch andere MPI Funktionen realisiert werden [7, S. 559f.], ist eine globale Statusvariable notwendig. Wenn diese gesetzt ist, wird die ursprüngliche Funktion ebenfalls normal ausgeführt.
- Bei einer normalen Erfassung wird der Zeitpunkt vor und nach dem Aufruf der ursprünglichen Funktion gemessen.
- Die Differenz der beiden Zeitpunkte wird bestimmt und damit die aktiven Messanfragen aktualisiert.
- Falls die Funktion eine persistente Kommunikationsanforderung initiiert hat, wird diese Anforderung registriert.

Diesen generischen Ablauf setzt der Generator folgendermaßen um: Sofern für keine MPI Metrik eine Messanfrage aktiv ist (alle Metrik-Stacks sind leer) oder die globale Statusvariable, die angibt, dass der Aufruf dieser Methode durch eine andere MPI Funktion erfolgt, gesetzt ist, wird die ursprüngliche Funktion unter dem „PMPI“ Präfix normal ausgewertet. Falls es sich um die Erzeugung einer persistenten Kommunikationsanforderung handelt, wird selbige im System registriert.

Für den Fall einer normalen Messung wird die globale Statusvariable gesetzt, die Zeit mittels `clock_gettime` (siehe 4.1.1) erfasst und die ursprüngliche Funktion unter dem „PMPI“ Präfix aufgerufen, wobei ihr Rückgabewert in einer Variablen zwischengespeichert wird. Nach diesem Aufruf wird die Zeit erneut gemessen und die Statusvariable deaktiviert. Wurde die Funktion keiner der oben genannten Funktionsteilmengen zugeteilt, wird eine Funktion aufgerufen, die aus den beiden Zeitwerten die Differenz berechnet und diese auf dem `MPITime`-Stack akkumuliert. Wenn die Funktion mindestens einer Teilmenge zugeordnet wurde, wird zuerst der Differenzwert berechnet und für jede Einteilung eine Funktion aufgerufen, die überprüft, ob für die jeweilige Metrik eine Messanfrage aktiv ist und der Metrik-Stack entsprechend aktualisiert.

Für eine Transferfunkton wird eine Prozedur aufgerufen, die aus dem Datentyp und dessen Anzahl (zweites und drittes Argument der MPI Funktion) die Transferzeit berechnet, sofern sie erfasst werden soll. Handelt es sich zudem um die Initialisierung einer persistenten Kommunikationsanforderung, wird diese registriert (siehe 4.1.3). Dabei wird der Umstand genutzt, dass für diese Funktionsgruppe die Reihenfolge der MPI Funktionsparameter für die Anzahl der Daten, deren Typ und das Handle der Anforderung gleich sind.

Am Ende der neu definierten Funktion wird der zwischengespeicherte MPI Rückgabewert zurückgegeben. Wenn alle Funktionsdeklarationen verarbeitet wurden, werden die erstellten Funktionsdefinitionen durch ROSE wieder in C++ Quellcode überführt. Die so erhaltene Datei kann zusammen mit der übrigen Laufzeitumgebung kompiliert werden, um die Informationen für die MPI Metriken zu erfassen.

Die nachfolgenden Listings zeigen beispielhaft die Neudefinitionen der Funktionen `MPI_Recv` (Punkt-zu-Punkt Kommunikation), `MPI_Send_init` (Initialisierung einer persistenten Kommunikationsanforderung) und `MPI_Bcast` (kollektive Kommunikation), zu beachten sind die Unterschiede aufgrund der verschiedenen Funktionsgruppen.

```
int MPI_Recv(void *arg0,int arg1,MPI_Datatype arg2,int arg3,int arg4,MPI_Comm arg5,MPI_Status *arg6)
{
  if (perfAssertion::MPIController::MPITimeCaptureDisabled() && perfAssertion::MPIController::Point2PointTimeCaptureDisabled
      () && perfAssertion::MPIController::TransferTimeCaptureDisabled() || perfAssertion::MPIController::inMPICall) {
    int returnValue = PMPI_Recv(arg0,arg1,arg2,arg3,arg4,arg5,arg6);
    return returnValue;
  }
  struct timespec t0;
  struct timespec t1;
  perfAssertion::MPIController::inMPICall = true;
  clock_gettime(CLOCK_MONOTONIC,&t0);
  int returnValue = PMPI_Recv(arg0,arg1,arg2,arg3,arg4,arg5,arg6);
  clock_gettime(CLOCK_MONOTONIC,&t1);
  perfAssertion::MPIController::inMPICall = false;
  perfAssertion_time diffTime = perfAssertion::MPIController::calculateElapsedTime(t0,t1);
  perfAssertion::MPIController::updatePoint2PointTime(diffTime);
  perfAssertion::MPIController::updateMPITime(diffTime);
  perfAssertion::MPIController::updateTransferTime(arg1,arg2);
  return returnValue;
}
```

Listing 4.1: Neudefinition von `MPI_Recv`

```
int MPI_Send_init(void *arg0,int arg1,MPI_Datatype arg2,int arg3,int arg4,MPI_Comm arg5,MPI_Request *arg6)
{
  if (perfAssertion::MPIController::MPITimeCaptureDisabled() && perfAssertion::MPIController::Point2PointTimeCaptureDisabled
      () || perfAssertion::MPIController::inMPICall) {
    int returnValue = PMPI_Send_init(arg0,arg1,arg2,arg3,arg4,arg5,arg6);
    perfAssertion::MPIController::registerPersistentCommunicationRequest(arg6,arg1,arg2);
    return returnValue;
  }
  struct timespec t0;
  struct timespec t1;
  perfAssertion::MPIController::inMPICall = true;
  clock_gettime(CLOCK_MONOTONIC,&t0);
  int returnValue = PMPI_Send_init(arg0,arg1,arg2,arg3,arg4,arg5,arg6);
  clock_gettime(CLOCK_MONOTONIC,&t1);
  perfAssertion::MPIController::inMPICall = false;
  perfAssertion_time diffTime = perfAssertion::MPIController::calculateElapsedTime(t0,t1);
  perfAssertion::MPIController::updatePoint2PointTime(diffTime);
  perfAssertion::MPIController::updateMPITime(diffTime);
  perfAssertion::MPIController::registerPersistentCommunicationRequest(arg6,arg1,arg2);
  return returnValue;
}
```

Listing 4.2: Neudefinition von `MPI_Send_init`

```
int MPI_Bcast(void *arg0,int arg1,MPI_Datatype arg2,int arg3,MPI_Comm arg4)
{
  if (perfAssertion::MPIController::MPITimeCaptureDisabled() && perfAssertion::MPIController::CollectiveTimeCaptureDisabled()
      || perfAssertion::MPIController::inMPICall) {
    int returnValue = PMPI_Bcast(arg0,arg1,arg2,arg3,arg4);
    return returnValue;
  }
  struct timespec t0;
  struct timespec t1;
  perfAssertion::MPIController::inMPICall = true;
  clock_gettime(CLOCK_MONOTONIC,&t0);
  int returnValue = PMPI_Bcast(arg0,arg1,arg2,arg3,arg4);
  clock_gettime(CLOCK_MONOTONIC,&t1);
  perfAssertion::MPIController::inMPICall = false;
  perfAssertion_time diffTime = perfAssertion::MPIController::calculateElapsedTime(t0,t1);
  perfAssertion::MPIController::updateCollectiveTime(diffTime);
  perfAssertion::MPIController::updateMPITime(diffTime);
  return returnValue;
}
```

Listing 4.3: Neudefinition von `MPI_Bcast`

4.3 Compiler

Die Aufgabe des mit ROSE geschriebenen Compilers ist es, die als Compiler-Pragmas formulierten Performance Assertions einzulesen und das Programm dementsprechend durch Modifikation des AST zu instrumentieren. Als Ausgabe erhält der Benutzer eine transformierte Quellcodedatei und durch den integrierten Aufruf des System-Compilers deren Kompilat, beziehungsweise ein ausführbares Programm, das die Assertions zur Laufzeit auswertet. Die Verwendung erfolgt dabei analog zum System-Compiler, sodass normalerweise der aufgerufene Compiler einfach ersetzt werden kann. Beim Aufruf des Compilers zum Linken von Objektdateien kommt es zu einem Fehler innerhalb von ROSE, sodass hierfür auf den System-Compiler zurückgegriffen werden muss.

Für die Kompilierung eines MPI Programmes werden spezielle Compiler-Flags, Include-Verzeichnisse und Bibliotheken benötigt. Aus diesem Grund stellen MPI Implementierungen einen gesonderten Compiler, wie etwa `mpicc`, bereit, der die erforderlichen Optionen an den Aufruf des Systemcompilers weiterleitet. Der MPI Standard [7] trifft keine Aussage darüber, wie die Einstellung, welcher Compiler eingesetzt wird, verändert werden kann. Aufgrund dessen variiert die Vorgehensweise je nach verwendeter Implementierung[8].

Als Alternative bietet der entwickelte Compiler daher über die Kommandozeilenoption „`-pa:mpi`“ die Möglichkeit, MPI Programme mit Performance Assertions transparent zu kompilieren. Die Option bewirkt, dass die benötigten Einstellungen für die Kompilierung eines MPI Programmes unter der installierten MPI Implementierung dem Aufruf hinzugefügt werden. Die Werte werden zum Zeitpunkt der Konfiguration des Buildvorgangs erfasst und in einem Header festgehalten. Auf diese Weise kann die direkte Verwendung der bereitgestellten Compiler vermieden werden.

[8] Vergleiche Handbucheinträge von OpenMPI und MPICH bezüglich `mpicc`

4.3.1 Aufbau

Um die Struktur des Compilers zu verdeutlichen, folgt ein kurzer Überblick über dessen Klassen. Eine detaillierte Beschreibung erfolgt in den nächsten Abschnitten.

`AssertionParser`: Führt das Parsing einer als Compiler-Pragma hinterlegten Performance Assertion durch und liefert als Ausgabe den AST des Auswertungsausdrucks sowie die Menge der verwendeten Identifier.

`AssertionProcessor`: Setzt eine vom Parser eingelesene Assertion durch Instrumentierung des betroffenen Codebereiches um.

`IdentifierProcessor`: Eine abstrakte Basisklasse, ihre Implementierungen kapseln die Bereitstellung eines Identifiers. Sofern es sich um Metriken und keine Konstanten handelt, wird hierfür eine Instrumentierung durchgeführt, die in eine Anfangsphase und eine Endphase aufgeteilt wird. Eine Implementierung kann gegebenenfalls für mehr als einen Identifier verantwortlich sein:

`TimeConstantsProcessor`: Stellt die Zeitkonstanten (`seconds`, ...) bereit.

`TimeProcessor`: Verantwortlich für die `WallTime` Metrik.

`MPIProcessor`: Übernimmt die Bereitstellung aller MPI Metriken.

Abbildung 4.2 zeigt das Klassendiagramm des Compilers, wobei nur öffentliche Artefakte sichtbar sind. Weiterhin werden nur Klassen der Domäne gezeigt. Klassen, welche die Verwendung von ROSE erleichtern, werden nicht dargestellt.

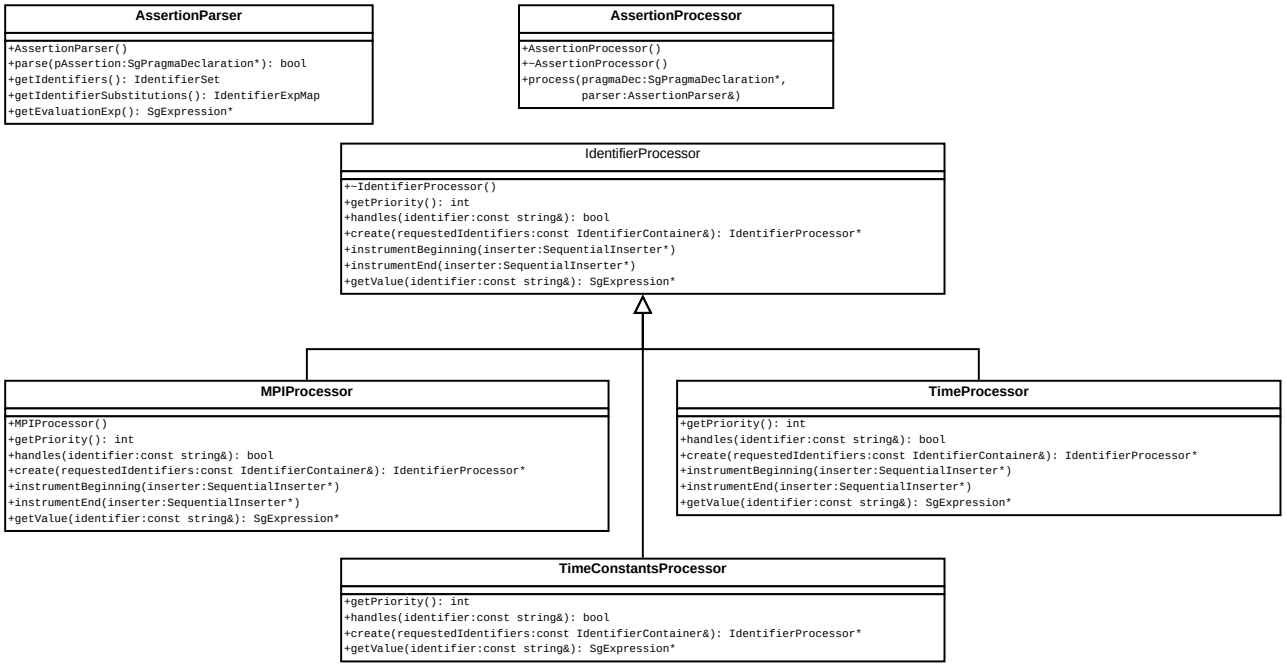

Abbildung 4.2: Klassendiagramm des Compilers

Ausgehend vom Einstiegspunkt des Compilers werden nach der Verarbeitung der Befehlszeile und dem Aufruf des Frontends mithilfe von ROSE alle Pragmadeklarationen in dem AST des Eingabeprogrammes ermittelt und über diese iteriert. Für jede Deklaration wird die `parse` Methode des `AssertionParser` aufgerufen und falls es sich um eine syntaktisch korrekte Performance Assertion handelt, die Deklaration und die Parser-Instanz an den `AssertionProcessor` weitergegeben. Handelt es sich um eine syntaktisch fehlerhafte Assertion, gibt der Parser eine Meldung auf der Konsole aus. Sofern die betrachtete Assertion aus einer logischen Verkettung von Aussagen besteht, von der mindestens ein Teil erfolgreich geparst werden konnte, enthält diese Meldung gegebenenfalls den Rest der Assertion, der nicht eingelesen werden konnte. Der Compiler überspringt Pragmadeklarationen, die keiner korrekten Performance Assertion entsprechen und untersucht die nächste gefundene Deklaration.

4.3.2 Parsing

Die Klasse `AssertionParser` implementiert einen Recursive-descent Parser[9] [24] unter Verwendung des `AstFromString` Namespace von ROSE, um die Assertion einzulesen und den Auswertungsausdruck aufzubauen. Bei dieser Technik wird jedes Nichtterminal der Grammatik durch eine Funktion umgesetzt. Die Funktionen rufen sich wechselseitig auf, sodass beim Parsing ausgehend vom Startsymbol der Syntaxbaum der Ableitung von oben nach unten aufgebaut wird.

Der `AstFromString` Namespace stellt Hilfsmethoden bereit, die unter anderem überprüfen, ob an der aktuellen Stelle des Assertiontexts ein bestimmtes Zeichen oder eine vorgegebene Zeichenkette vorhanden ist. Weiterhin gibt es Methoden, welche die Konstruktion des AST von Teilausdrücken der Programmiersprache C erlauben.

Innerhalb der `parse` Methode des `AssertionParsers` wird der interne Zustand zunächst zurückgesetzt und der Text des Pragmas extrahiert. Falls dieser mit dem Schlüsselwort „perfAssertion“ beginnt, wird der eigentliche Einlesevorgang mit dem Nichtterminal `PA_Assumptions` gestartet. Abgesehen von `PA_Pragma` sind alle anderen Nichtterminale als private Methoden der Klasse implementiert, die einen Wahrheitswert zurückgeben, der abhängig von ihrem Erfolg ist. Wenn die Methode für `PA_Assumptions` `true` zurückgibt und der Pragmatext vollständig verarbeitet wurde, war der Vorgang erfolgreich. Falls der Text nicht vollständig verarbeitet werden konnte oder es sich um kein gültiges `PA_Assumptions` handelt, liegt ein Syntaxfehler vor und die Assertion ist nicht gültig, weswegen eine Fehlermeldung ausgeben wird und die `parse` Methode `false` zurückgibt.

Die übrigen Nichtterminale (siehe 3.2) werden wie folgt realisiert:

`PA_Assumptions`: Der Aufruf wird an `PA_LogicalExp0` weitergeleitet, da es sich bei diesem Nichtterminal um einen Platzhalter handelt.

`PA_LogicalExp0`: Es wird die Funktion `match_recBinaryOps` eingesetzt, um die Ableitungen von `PA_LogicalExp1` so oft wie möglich mit den logischen Operatoren „|“ oder „->“ zu verknüpfen. Die Implikation wird dabei als Disjunktion der Negation des ersten Operanden und des zweiten Operanden realisiert ($\neg a \vee b$).

`PA_LogicalExp1`: Es wird die Funktion `match_recBinaryOps` eingesetzt, um die Ableitungen von `PA_Assumption` so oft wie möglich mit dem logischem Operator „&“ zu verknüpfen.

`PA_Assumption`: Falls die aktuelle Textstelle mit einem „!“ oder „(“ beginnt, wird versucht ein geklammertes `PA_Assumptions` Nichtterminal herzuleiten, welches gegebenenfalls negiert wird. Ansonsten wird versucht `PA_RelationalExp` abzuleiten.

`PA_RelationalExp`: Es wird versucht, das Nichtterminal `PA_MathExp` gefolgt von einem der relationalen Operatoren und einem weiteren `PA_MathExp` abzuleiten. Ist dies möglich, werden die beiden Ausdrücke durch den spezifizierten Operator verknüpft.

`PA_MathExp`: Der Aufruf wird an `PA_AdditiveExp` weitergeleitet, da es sich bei diesem Nichtterminal um einen Platzhalter handelt.

`PA_AdditiveExp`: Es wird die Funktion `match_recBinaryOps` eingesetzt, um die Ableitungen von `PA_MultiplicativeExp` so oft wie möglich mit den additiven Operatoren zu verknüpfen.

`PA_MultiplicativeExp`: Es wird die Funktion `match_recBinaryOps` eingesetzt, um die Ableitungen von `PA_UnaryMinusExp` so oft wie möglich mit den multiplikativen Operatoren zu verknüpfen.

`PA_UnaryMinusExp`: Es wird überprüft, ob sich an der aktuellen Textposition ein „-“ befindet und versucht, das Nichtterminal `PA_Atom` abzuleiten. Das optionale „-“ führt dazu, dass der unäre Minusoperator angewandt wird.

`PA_Atom`: Gemäß der Grammatik wird zunächst versucht eine Dezimalzahl (`PA_Number`) abzuleiten. Sofern dies nicht gelingt, wird die Herleitung von `PA_Identifier`, `PA_ConfigValueReference`, `PA_VariableReference`, `PA_BuiltinFunction` oder einem geklammerten `PA_MathExp` Nichtterminal versucht.

`PA_Identifier`: Anhand einer Tabelle mit allen bekannten Identifiern wird überprüft, ob die aktuelle Textposition mit einer solchen Zeichenkette beginnt. Ist dies der Fall, wird als temporärer Platzhalter eine opake Variablenreferenz mit dem Namen des Identifiers für den zu konstruierenden AST gebaut, da der eigentliche Wert erst später, nach der Instrumentierung, feststeht. Die Assoziation zwischen Identifier und der erstellten Referenz wird durch einen Eintrag in einer Multimap festgehalten.

[9] Im Deutschen wird dieser Ansatz auch „Rekursiver Abstieg“ genannt

PA_ConfigValueReference: Falls die aktuelle Textposition mit den Zeichen „$“ und „{“ beginnt, handelt es sich um den Zugriff auf einen Konfigurationswert, der durch die Funktion perfAssertion_getDblConfigValue der Laufzeitumgebung ausgeführt wird. Als Argument erwartet die Funktion den Namen des Konfigurationswerts, der vom Parser mithilfe von AstFromString als Kette von erlaubten Bezeichnersymbolen aufgebaut wird. Sofern der Name mit einem „}“ endet, wird der entsprechende Funktionsaufrufausdruck für den AST des Auswertungsausdrucks erstellt.

PA_VariableReference: Wenn die aktuelle Textposition mit dem Zeichen „$“ beginnt, wird mithilfe von AstFromString versucht, die Referenz auf einen Bezeichner abzuleiten. Falls für den Scope der Performance Assertion keine gültige Referenz gefunden werden kann, wird eine opake Variablenreferenz mit dem Namen des Bezeichners erstellt, damit Präprozessordefinitionen, wie beispielsweise MPI_COMM_WORLD, als Variablenreferenzen genutzt werden können. Handelt es sich bei der gefundenen Referenz nicht um eine Variable, sondern beispielsweise um eine Sprungmarke, schlägt diese Ableitung fehl.

PA_BuiltinFunction: Es wird zunächst versucht, den Funktionsaufruf der unären, eingebetteten Funktionen (exp, log, sqrt und abs) abzuleiten. Hierfür wird eine Liste verwendet, die Einträge bestehend aus dem Namen der Funktion und der Referenz auf eine Funktion, welche den passenden Funktionsaufrufausdruck erstellt, für jede eingebettete unäre Funktion beinhaltet. Die mathematischen Funktionen werden durch die (fast) gleichnamigen Funktionen der C Laufzeitbibliothek umgesetzt, wobei abs durch fabs realisiert wird[10].

Handelt es sich nicht um eine unäre, mathematische Funktion, wird überprüft, ob es sich um den Aufruf der binären pow Funktion handelt und gegebenenfalls deren Aufrufausdruck erstellt. Ist dies auch nicht der Fall, wird getestet, ob es sich um den Aufruf von nMPIProcesses handelt. Hierzu muss die aktuelle Textposition mit „nMPIProcesses“ und einer sich öffnenden Klammer beginnen und das Nichtterminal PA_VariableReference gefolgt von einer sich schließenden Klammer ableitbar sein. nMPIProcesses wird durch die Funktion perfAssertion_getMPIProcessesCount der Laufzeitumgebung umgesetzt.

Bei match_recBinaryOps handelt es sich um eine Hilfsfunktion zum Parsen von Produktionen der Form B { „x“ B }, die folgende Argumente übergeben bekommt:

- Eine Liste, die für jeden Operator dessen Symbol als Zeichenkette und eine Funktion enthält, welche aus zwei Ausdrücken einen Ausdruck zur Realisierung des Operators erstellt.
- Eine Funktion, die das nächste Nichtterminal ableitet.

Die Funktion versucht, so viele Ableitungen des Nichtterminals wie möglich mit den übergebenen Operatoren zu verknüpfen. Hierbei muss mindestens ein Nichtterminal erfolgreich abgeleitet werden können und auf jeden Operator ein erneutes Nichtterminal folgen, damit die Funktion true zurückgibt.

Während der Implementierung zeigte sich, dass das von ROSE eingesetzte Frontend beim Einlesen des Eingabeprogrammes bereits eine Vorverarbeitung der Compiler-Pragmas durchführt. Dadurch wird beispielsweise der Ausdruck „−2.0“ zu „−(2.0)“ umgewandelt und bestimmte Bezeichner durch Leerzeichen getrennt. Dieses Verhalten ist unter anderem der Grund dafür, dass bei der Erkennung der Konfigurationswerte getrennt nach den Zeichen „$“ und „{“ gesucht wird. Weiterhin scheint dieses Verhalten je nach der eingesetzten Version von ROSE zu variieren, sodass in der Zukunft Änderungen am Parser erforderlich sein könnten.

4.3.3 Instrumentierung

Nachdem eine Performance Assertion durch den Parser erfolgreich eingelesen wurde, muss der AST des Eingabeprogrammes so verändert werden, dass die angeforderten Metriken für den Gültigkeitsbereich der Assertion gemessen und an dessen Ende ausgewertet werden. Diese Aufgabe übernimmt die process Methode der AssertionProcessor Klasse. Aus der übergebenen AssertionParser Instanz wird die Menge der in der Assertion verwendeten Identifier extrahiert und daraus die benötigten IdentifierProcessor Instanzen erstellt, die für die Bereitstellung einer oder mehrerer Identifier verantwortlich sind:

Der AssertionProcessor verfügt über eine Liste von IdentifierProcessor Prototypen. Bei der Erstellung der für eine Assertion benötigten Instanzen wird über diese Liste iteriert und für jedes Element die Menge der angeforderten Identifier gebildet, die von diesem Typen verarbeitet werden können. Anschließend wird über die create Methode des IdentifierProcessor und die Identifier-Menge die benötigte Instanz erstellt und die abgearbeiteten Identifier aus der Liste der angeforderten entfernt.

[10] abs bezeichnet die ganzzahlige Variante des Betrages, fabs die Fließkommavariante

Im nächsten Schritt werden die Deklarationen der Laufzeitumgebungsfunktionen in den globalen Scope der übergebenen Pragmadeklaration eingefügt, sofern dies durch eine vorherige Performance Assertion in der gleichen Quellcodedatei nicht bereits geschehen ist. Dies schließt die Deklarationen der mathematischen Funktionen, wie `exp` oder `pow`, ein. Die Deklaration der Funktion `perfAssertion_getMPIProcessesCount` wird jedoch unabhängig davon bei der ersten Verwendung emittiert, da hierfür der MPI Header wegen des Parameters vom Typ `MPI_Comm` inkludiert werden muss.

Als nächstes wird mit Hilfe von ROSE überprüft, ob die Assertion in einer Schleife liegt und ob die Zielanweisung der Assertion, die nächste Anweisung nach dem Pragma, gültig ist. Akzeptiert werden Schleifen (`for`, `while`, `do-while`) und Basic Blocks. Eine ungültige Zielanweisung führt dazu, dass die Verarbeitung der Performance Assertion abgebrochen wird. Falls die Assertion in einer Schleife liegt, werden alle Body Anweisungen in Basic Blocks konvertiert, um die spätere Behandlung von Schleifenkontrollflussanweisungen zu vereinfachen.

Die Reihenfolge, in der die Messung der Metriken initiiert wird, ist relevant, wenn eine gegenseitige Beeinflussung von unterschiedlichen Metriken minimiert werden soll. Benötigt eine Assertion für ihre Auswertung beispielsweise sowohl die `MPITime` als auch die `WallTime` Metrik, muss die Initiierung der `MPITime` Metrik vor dem Start der `WallTime` Messung erfolgen, damit die dafür benötigte Zeit nicht in die Erfassung der real verstrichenen Zeit mit einfließt. Aus diesem Grund verfügen alle `IdentifierProcessor` Implementierungen über eine statische Priorität, welche die Aufrufreihenfolge der `instrumentBeginning` und `instrumentEnd` Funktionen festlegen. Erstere erfolgt in aufsteigender Reihenfolge der Priorität, letztere in der umgekehrten Reihenfolge. Folglich wird die Liste der benötigten `IdentifierProcessor` Instanzen nach ihrer Priorität sortiert und die Instrumentierung durch `instrumentBeginning` und in umgekehrter Reihenfolge durch `instrumentEnd` ausgeführt. Beiden Funktionen wird jeweils eine Instanz der Hilfsklasse `SequentialInserter` übergeben, die dafür sorgt, dass die `IdentifierProcessor` Instanzen ihre Anweisungen nacheinander vor beziehungsweise nach der Zielanweisung der Assertion einfügen.

Nach der Instrumentierung werden die Platzhalter für die Identifiers im Auswertungsausdruck des Parsers durch die richtigen Werte ersetzt. Dazu wird über alle Einträge der Multimap des `AssertionParsers` iteriert und aus der Liste der eingesetzten `IdentifierProcessor` Instanzen diejenige ermittelt, für welche die `handles` Funktion für den übergebenen Identifier zu `true` auswertet. Der eigentliche Wert des Identifiers wird über die `getValue` Funktion ermittelt und der Platzhalter durch diesen Ausdruck ersetzt. Danach wird der der Name der Quellcodedatei und die Zeile der Pragmadeklaration ermittelt und aus dem nun vollständigem Auswertungsausdruck der `perfAssertion_reportAssertion` Aufruf zur Meldung des Ergebnisses der Performance Assertion an die Laufzeitumgebung gebaut und in den AST integriert.

Falls die Assertion in einer Schleife liegt, müssen abschließend noch eventuelle Kontrollflussanweisungen, `continue` und `break`, behandelt werden, da ansonsten die Messung der Metriken nicht beendet werden würde und es unter Umständen zu einem Speicherleck käme. Um dies zu verhindern, werden mithilfe von ROSE alle `continue` und `break` Anweisungen im Gültigkeitsbereich der Assertion ermittelt. Vor jede der gefundenen Anweisungen werden mittels `instrumentEnd` die Instruktionen zum Beenden der Messung emittiert und die Meldung der Ergebnisse an die Laufzeitumgebung eingefügt.

Wenn alle gefundenen potenziellen Assertions verarbeitet worden sind, wird die Definition des Einstiegspunktes des Eingabeprogrammes gesucht und an dessen Anfang ein Aufruf der Laufzeitumgebungsfunktion `perfAssertion_init` sowie deren Deklaration eingefügt. Der Funktionsaufruf wird benötigt, damit das Konfigurationssystem geladen und am Ende des Programmes der Bericht über die Auswertung der Assertions generiert wird (siehe 4.1).

Umsetzung der Metriken

Nachdem im oberem Abschnitt der allgemeine Instrumentierungsprozess einer Performance Assertion erklärt wurde, folgt nun die Beschreibung, wie die Metriken durch die einzelnen `IdentifierProcessor` Implementierungen umgesetzt werden und welche Instrumentierungen hierfür gegebenenfalls durchgeführt werden müssen.

`TimeConstantsProcessor`: Diese Implementierung übernimmt die Umwandlung der Zeitkonstanten `seconds`, `milliseconds` und `microseconds` in ihre konkreten Werte. Dies erfordert keine Instrumentierung, weswegen eine Priorität von 0 verwendet wird. Stattdessen erstellt die Methode `getValue` abhängig vom Parameter folgende ganzzahlige Ausdruckskonstanten:

- Für `seconds` 1000000000, was 10^9 entspricht.
- Für `milliseconds` 1000000, was 10^6 entspricht.
- Für `microseconds` 1000, was 10^3 entspricht.

Diese Werte sind abhängig vom Zeitgeber der Laufzeitumgebung, siehe 4.1.1.

`TimeProcessor`: Dieser `IdentifierProcessor` ist für die `WallTime` Metrik verantwortlich und besitzt eine Priorität von 1000, damit er als letzter die Instruktionen zum Start der Messung emittiert und als erster die für das Beenden

der Messung. In der Anfangsphase der Instrumentierung wird über ROSE ein für den aktuellen Scope eindeutiger Variablenname generiert und daraus eine Variablendeklaration einer vorzeichenlosen 64 Bit Ganzzahl erstellt. Dieser Variablen wird der Rückgabewert von der Laufzeitumgebungsfunktion `perfAssertion_getTime` zugewiesen. In der Endphase wird der Wert der Variablen auf die Differenz zwischen einem erneuten Aufruf der Funktion und ihrem alten Wert gesetzt. Das Ergebnis der Metrik wird über eine Variablenreferenz in den Auswertungsausdruck der Performance Assertion übernommen.

`MPIProcessor`: Die Umsetzung aller MPI Metriken erfolgt durch diese Implementierung, die eine Priorität von 0 besitzt. Bei der Erstellung einer Instanz zum Instrumentieren aus dem Prototypen mittels `create` erfolgt die Kodierung der für die Assertion zu messenden Metriken als Ganzzahl (siehe 4.1.3). Hierzu wird ein Multi-index Container der Boost Bibliotheken mit Paaren aus den Zeichenketten der Metriken und der zugehörigen ganzzahligen Konstanten als Nachschlagetabelle verwendet. Die Werte der Konstanten müssen mit denen in der Laufzeitumgebung verwendeten übereinstimmen.

In der Anfangsphase der Instrumentierung wird anhand eines für den aktuellen Scope eindeutigen Variablennamens ein Array von 64 Bit Ganzzahlen erstellt, dessen Größe der Anzahl der angeforderten MPI Metriken, beziehungsweise der in der kodierten Ganzzahl gesetzten Bits, entspricht. Nach der Variablendeklaration wird der Aufruf der `perfAssertion_enterMPICapture` Laufzeitumgebungsfunktion mit den kodierten Metriken als Parameter eingefügt. In der Endphase wird die Messanfrage durch Emittieren eines `perfAssertion_leaveMPICapture` Aufrufes, der die Adresse des Arrays und die kodierten Metriken als Parameter erhält, beendet. Das Ergebnis einer Metrik wird als Index-basierter Zugriff auf das Array extrahiert. Der für eine Metrik benötigte Index ergibt sich aus der Anzahl der gesetzten Bits in der Kodierung, die vor dem Bit der abzurufenden Metrik liegen. Das Bitintervall, das dafür überprüft werden muss, wird mithilfe der Nachschlagetabelle bestimmt.

5 Evaluation

Dieses Kapitel beschäftigt sich mit der Evaluation der vorgestellten Performance Assertion Implementierung anhand der praktischen Anwendung auf die `su3_rmd` Applikation der MIMD Lattice Computation (MILC) Collaboration[1], die eine Codesammlung zur Simulation der vierdimensionalen SU(3) Gittereichtheorie auf parallelen MIMD Maschinen entwickelt hat. Hierbei handelt es sich um ein Problem aus dem Feld der Quantenchromodynamik, welche die Wechselwirkung zwischen Quarks und Gluonen beschreibt. Als Grundlage der Untersuchung wurde eine MILC Applikation verwendet, da diese bereits für mehrere Benchmarks, wie SPEC CPU2006[2], SPEC MPI2007[3] und NERSC-6[4], eingesetzt wurden und im Falle von `su3_rmd` bereits durch Bauer et al. [3] ein semianalytisches Performance Modell entwickelt wurde, das auch in anderen wissenschaftlichen Arbeiten eingesetzt wurde [8]. Die im Rahmen dieser Evaluation aufgestellten Performance Assertions basieren auf diesem Modell und eigenen Analysen mittels Vampir [15], einem Werkzeug zur visuellen Untersuchung aufgezeichneter Event Traces.

Das Ziel der Evaluation ist die Validierung der in Kapitel 4 vorgestellten Implementierung der Performance Assertions. Dabei stehen Funktion wie auch Overheads im Fokus der Untersuchung einer realen Anwendung und deren Laufzeitverhalten. Hierzu werden Performance Assertions basierend auf den Arbeiten von Bauer et al. für die MILC Anwendung aufgestellt und getestet. Der durch die Instrumentierung des Programmes entstandene Overhead im Vergleich zu einer normal ausgeführten Programmversion wird ebenso untersucht.

5.1 Analyse der MILC Anwendung

Die Aufstellung der Performance Assertions für MILC erfordert Kenntnisse über die folgenden Sachverhalte:

- Den allgemeinen Programmablauf,
- für das Laufzeitverhalten wichtige Codeblöcke beziehungsweise Codestellen
- und die Eigenschaften dieser Codeblöcke beziehungsweise zutreffende Aussagen.

Das von Bauer et al. [3] aufgestellte Modell umfasst alle drei Aspekte. Für ein genaueres Verständnis von MILC und die Untersuchung von im Modell nicht abgebildeten Zusammenhängen werden die Analysen mittels Vampir durchgeführt.

Für ihr Modell haben Bauer et al. Codeblöcke (Kernel) identifiziert, die für die Performance des Programmes ausschlaggebend sind. Diese Evaluation konzentriert sich auf die `imp_gauge_force`, `eo_fermion_force_twoterms` und `ks_congrad` Kernel. Die einzelnen Kernel wurden jeweils separat hinsichtlich ihrer Berechnungszeit und ihres Kommunikationsverhaltens untersucht. Die sequentielle Laufzeit wurde durch einen mathematischen Ausdruck modelliert, der auf der Annahme eines zweistufigen Cache basiert.

MILC abstrahiert die Punkt-zu-Punkt Kommunikation unter den Prozessen für den Austausch der Randbereichsdaten durch sogenannte `gather` Funktionen, sodass andere Kommunikationsschnittstellen genutzt werden können. Beim direkten Einsatz von MPI werden intern `MPI_Isend` und `MPI_Irecv` verwendet. Im Modell wird die Punkt-zu-Punkt Kommunikationszeit aus der ermittelten Nachrichtengröße und der Anzahl der `gather` Aufrufe pro Kernel in Abhängigkeit von den Eingabeparametern abgeleitet. Jeder Aufruf des `imp_gauge_force` Kernels erzeugt 828 `gather` Aufrufe, der des `eo_fermion_force_twoterms` Kernels 1616 Aufrufe. Die Nachrichtengröße ergibt sich für beide Kernel zu $18 \cdot f \cdot \sqrt[4]{V^3}$, wobei f die Größe einer Fließkommazahl in Bytes ist, V die Gittergröße pro Prozess repräsentiert und von einer Gleichverteilung aller Dimensionen ausgegangen wird. Im Rahmen der Evaluation werden gemäß der Standardeinstellung Fließkommazahlen einfacher Genauigkeit betrachtet, folglich gilt $f = 4$.

Durch die Aufteilung des Gitters auf die einzelnen Prozesse und die Verwendung von periodischen Randbedingungen besitzt jeder Prozess exakt acht Nachbarn. Die Analyse mithilfe von Vampir zeigte, dass es abhängig von der gewählten Aufteilung und der Anzahl der Prozesse vorkommt, dass nicht für jeden Nachbarn ein ausgeführter `gather` zum Austausch von Nachrichten führt. Dies ist beispielsweise der Fall, wenn es sich bei einigen Nachbarn um den eigenen Prozess handelt. Wenn Nachrichten ausgetauscht werden, bestehen die eingesetzten `gather` jeweils aus einem `MPI_Isend` und einem `MPI_Irecv`.

[1] `http://physics.indiana.edu/~sg/milc.html`, zuletzt geprüft am 23.11.13

[2] `http://www.spec.org/cpu2006/`, zuletzt geprüft am 12.11.13

[3] `http://www.spec.org/mpi2007/`, zuletzt geprüft am 12.11.13

[4] `http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/`, zuletzt geprüft am 12.11.13

Für die Modellierung der kollektiven Kommunikationszeit, die durch die Verwendung von `MPI_Allreduce` innerhalb des `ks_congrad` Kernels anfällt, setzen Bauer et al. eine „Tree-based“ Reduktion [17] voraus und kombinieren deshalb den Logarithmus der Prozessanzahl mit der Anzahl der aus den Eingabeparametern abgeleiteten Reduktionen.

Die Kernel stehen innerhalb der MILC Anwendung in folgendem Zusammenhang: Nach einem initialen Setup wird die eingestellte Anzahl an Elementen, in MILC Trajektorien, über eine Schleife im Einstiegspunkt berechnet. Die Berechnung einer einzelnen Trajektorie wird durch die `update` Prozedur realisiert, die ihrerseits für die Berechnung der Schritte die Methoden `update_u`, `update_h`, `restore_fermion_links_from_site` und `ks_congrad_two_src` benutzt. Erstere ist für die Laufzeit vernachlässigbar, da sie für 4 Prozesse einen Anteil von weniger als 2% an der Gesamtlaufzeit besitzt. `update_h` setzt sich dagegen aus den Kernel `imp_gauge_force` und `eo_fermion_force_twoterms` zusammen. Im Verlauf der Auswertung von `restore_fermion_links_from_site` wird der ebenfalls in [3] identifizierte Kernel `load_fatlinks` aufgerufen. Die `ks_congrad_two_src` Prozedur verwendet zwei Aufrufe des `ks_congrad` Kernels.

Anhand der Erkenntnisse aus dem Modell und der Analyse mit Vampir wurden die im nächsten Abschnitt vorgestellten Performance Assertions aufgestellt.

5.2 Formulierte Assertions

In diesem Abschnitt werden die für die Evaluation verfassten Performance Assertions vorgestellt. Jeder Assertion wird ein eindeutiger Bezeichner zugewiesen, der sich analog zu 4.1.4 aus der Quellcodedatei und der Zeile des definierenden Compiler-Pragmas zusammensetzt. Die Vorstellung der einzelnen Assertions erfolgt gruppiert nach der zugehörigen Quellcodedatei, wobei für das Verständnis des Kontextes und der genauen Platzierung ein Auszug gegeben wird.

Beim Formulieren und Testen der Assertions, die das Punkt-zu-Punkt Kommunikationsverhalten der Kernel `imp_gauge_force` und `eo_fermion_force_twoterms` aus dem Modell [3] validieren, zeigte sich, dass eine einfache Betrachtung der aufgerufenen `gather` nicht genügt, da diese, wie bereits in 5.1 beschrieben, abhängig von der Aufteilung des Gitters, nicht immer zu einem Nachrichtenaustausch führen und die auftretenden Differenzen zum realen Kommunikationsaufwand zu groß sind: Für den `imp_gauge_force` Kernel kommen beispielsweise bei 8 Prozessen auf 828 `gather` Aufrufe zusammen 1242 `MPI_Isend` und `MPI_Irecv` Aufrufe. Geht man davon aus, dass jeder `gather` Aufruf aus dem Senden und Empfangen einer Nachricht besteht, ist der reale Kommunikationsaufwand in diesem Fall um 25% geringer. Um diesen Umstand genauer abzubilden, wurde der MILC Code um die Funktion `pa_getGatherFactor` erweitert, die anhand der Anzahl von Dimensionen, in denen die Länge des Subgitters 1 beträgt, einen Faktor berechnet, der multipliziert mit der Anzahl an `gather` Prozeduraufrufen pro Kernel die Summe der eingesetzten `MPI_Isend` und `MPI_Irecv` ergibt.

Assertions in `control.c`

```
#pragma perfAssertion MPITime < 5.6*seconds & (nMPIProcesses($MPI_COMM_WORLD) == 1 -> MPITime/
    WallTime < 0.01)
for( traj_done=0; traj_done < trajecs; traj_done++ ){

  #pragma perfAssertion MPIWaitTime < MPITransferTime
  {
    /* do the trajectories */
    s_iters=update();
  }

  /* measure every "propinterval" trajectories */
  if( (traj_done%propinterval)==(propinterval-1) ){

    /* ... */

    #pragma perfAssertion WallTime <= 0
    {
      restore_fermion_links_from_site(fn_links, prec_pbp);
    }

    /* ... */
  }
} /* end loop over trajectories */
```

`perfAssertion MPITime < 5.6*seconds & (nMPIProcesses($MPI_COMM_WORLD) == 1 -> MPITime/WallTime < 0.01)` **(control.c:60):** Die Assertion bezieht sich auf die Schleife der zu berechnenden Trajektorien und wird folglich nur einmal im Laufe des Programmes ausgewertet. Sie drückt aus, dass die in der Berechnungsphase akkumulierte MPI Zeit kleiner als 5,6 Sekunden ist und falls nur ein Prozess ausgeführt wird, dass die MPI Zeit einen Anteil von

weniger als einem Prozent an der real verstrichenen Zeit besitzt. Die 5,6 Sekunden entsprechen der durchschnittlich pro Prozess gemessenen MPI Zeit während der Analysen mit Vampir, bei denen auffiel, dass dieser Wert für alle betrachteten Prozessanzahlen eine obere Schranke darstellt. Bei einem eingesetzten Prozess betrug der Anteil der MPI Zeit in den Analysen ca. 0,6%, was für die Assertion auf ein Prozent aufgerundet wurde.

Diese Assertion soll zeigen, wie ausdrucksvoll eine einmalig ausgewertete Aussage über das Laufzeitverhalten mit einer konstanten oberen Schranke ist und demonstriert gleichzeitig den Einsatz des Implikationsoperators, um eine Aussage für einen Sonderfall (Ausführung mit einem Prozess) zu verschärfen.

`perfAssertion MPIWaitTime < MPITransferTime` **(control.c:63):** Der Gültigkeitsbereich der Assertion beträgt einen `update` Aufruf und entspricht damit der Berechnung einer Trajektorie. Da MILC asynchrone Punkt-zu-Punkt Kommunikation verwendet (siehe 5.1), versucht diese Performance Assertion zu zeigen, dass sich die MPI Wartezeit durch die Transferzeit nach oben begrenzen lässt. Dies geschieht unter der Annahme, dass ohne eine Überlappung von Kommunikation und Berechnung die Wartezeit hauptsächlich durch die Übertragung der Nachricht zustande kommt. Hierbei werden einige Vereinfachungen getroffen:

- Die Transferzeit ist keine direkte Messgröße, sondern wird anhand der festgelegten Systemparameter heuristisch berechnet und weicht deshalb gegebenenfalls von der tatsächlich benötigten Übertragungszeit ab.

- Eine Rückkehr von `MPI_Wait` signalisiert, dass die betroffene Operation abgeschlossen ist. Im Falle einer Sendeoperation ist dies jedoch keine Garantie dafür, dass die Übertragung selbst vollständig ist: Die Nachricht kann vom Kommunikationssystem gepuffert worden sein [7, S. 52f.].

`perfAssertion WallTime <= 0` **(control.c:78):** Diese Performance Assertion dient als simple Kontrolle für die generelle Korrektheit der Implementierung, da die Aussage, dass für die Ausführung des Codeblocks keine oder negative Zeit vergeht, immer falsch sein sollte. Wenn die Assertion mindestens einmal zu wahr ausgewertet wird, liegt entweder ein Fehler in der Implementierung vor oder die Auflösung des Zeitgebers ist nicht ausreichend, da die im Gültigkeitsbereich aufgerufene Funktion nicht leer ist.

Im Zuge der Implementierung wäre es möglich gewesen, eine solche Assertion implizit in jedes bearbeitete Programm einzubauen. Da hierbei keine langfristigen Vorteile entstehen und dieses Verhalten vom Benutzer nicht gewünscht sein kann oder der zusätzliche Eintrag im Ereignisbericht, für den eine nachvollziehbare Zeilennummer generiert werden muss, verwirren kann, wurde auf diese Funktionalität verzichtet.

Assertion in `update.c`

```
#pragma perfAssertion 7.8 * pow(nMPIProcesses($MPI_COMM_WORLD), -0.93) * seconds > WallTime
{
  /* now update H by full time interval */
    update_h(epsilon);
}
```

`perfAssertion 7.8 * pow(nMPIProcesses($MPI_COMM_WORLD), -0.93) * seconds > WallTime` **(update.c:117):** Die Assertion geht auf die mit Vampir durchgeführte Analyse und die dadurch gewonnen Erkenntnisse zurück. Dementsprechend ergeben sich die verwendeten Konstanten durch Ausgleichen einer Kurve anhand der ermittelten Messwerte. Sie verdeutlicht, dass das Laufzeitverhalten der `update_h` Prozedur in Abhängigkeit von der Anzahl der verwendeten Prozesse sich durch eine Hyperbel beschränken lässt.

Assertion in `update_u.c`

```
void update_u( Real eps ){

  #pragma perfAssertion MPITime == 0
  {
    /* ... */
  }

} /* update_u */
```

`perfAssertion MPITime == 0` **(update_u.c:18):** Die Prozedur `update_u` führt im Unterschied zu der ähnlich benannten `update_h` Methode nur lokale Berechnungen aus und enthält keine MPI Kommunikation. Folglich sollte die `MPITime` immer 0 sein. Diese Aussage bezieht sich auf eventuelle zukünftige Änderungen und besitzt in dieser Evaluation daher den Charakter einer zusätzlichen Überprüfung der Implementierung.

Assertions in `update_h.c`

```
void update_h( Real eps ){
  /* ... */
  const double pa_gfactor = pa_getGatherFactor();
  #pragma perfAssertion abs((18 * 4 * pow($nx * $ny * $nz * $nt / nMPIProcesses($MPI_COMM_WORLD), 0.75) /
       (${PA_TRANSFER_RATE} / 8000) + ${PA_TRANSFER_LATENCY} * microseconds) * 828 * $pa_gfactor -
      MPITransferTime) / MPITransferTime < 0.1
  {
    imp_gauge_force(eps,F_OFFSET(mom));
  }

  /* ... */
  #pragma perfAssertion abs((18 * 4 * pow($nx * $ny * $nz * $nt / nMPIProcesses($MPI_COMM_WORLD), 0.75) /
       (${PA_TRANSFER_RATE} / 8000) + ${PA_TRANSFER_LATENCY} * microseconds) * 1616 * $pa_gfactor -
      MPITransferTime) / MPITransferTime < 0.1
  {
    eo_fermion_force_twoterms_site( eps, ((Real)nflavors1)/4.,
    ((Real)nflavors2)/4., F_OFFSET(xxx1),
    F_OFFSET(xxx2), ff_prec, fn_links );
  }
} /* update_h */
```

`perfAssertion abs((18 * 4 * pow($nx * $ny * $nz * $nt / nMPIProcesses($MPI_COMM_WORLD), 0.75) / (${PA_TRANSFER_RATE} / 8000) + ${PA_TRANSFER_LATENCY} * microseconds) * 828 * $pa_gfactor - MPITransferTime) / MPITransferTime < 0.1` **(update_h.c:26):** Diese Assertion geht auf das Performance Modell von Bauer et al. [3] zurück und validiert die Anzahl der `gather` Aufrufe und deren Nachrichtengrößen für den `imp_gauge_force` Kernel. Hierzu wird die Übertragungszeit der Nachrichten identisch zu 4.1.3 aus der Anzahl der gesendeten und empfangenen Nachrichten und deren Größe berechnet und mit dem Wert der `MPITransferTime` verglichen, wobei ein relativer Fehler von 10% toleriert wird. Die Größe einer Nachricht ergibt sich nach Bauer et al. zu $18 \cdot f \cdot \sqrt[4]{V^3} = 18 \cdot 4 \cdot \left(\frac{nx \cdot ny \cdot nz \cdot nt}{P}\right)^{0.75}$, wobei *nx*, *ny*, *nz* und *nt* für die globale Dimension des Gitters stehen und *P* für die Anzahl der eingesetzten MPI Prozesse. Die Nachrichtenanzahl wird über den layoutabhängigen Faktor `pa_gfactor` aus der Anzahl der eingesetzten `gather` pro Kernel-Aufruf (828) abgeleitet. Für die Berechnung der Übertragungszeit werden die Systemparameter der `MPITransferTime` aus der Konfigurationsdatei verwendet, sodass eine Abweichung der beiden Werte nur durch unterschiedliche Nachrichtengrößen und Anzahlen zustande kommen kann.

Ein Performance Modell erlaubt die Quantifizierung des Laufzeitverhaltens einer untersuchten Anwendung und basiert dabei auf vereinfachten Annahmen. Im betrachteten Modell [3] wird beispielsweise angenommen, dass die Gittergröße auf alle Dimensionen gleich verteilt ist. Diese Vereinfachung trifft allerdings für die im Rahmen der Evaluation gewählte Gittergröße nicht zu. Um zu gewährleisten, dass das Modell trotz der getroffenen Annahmen das Laufzeitverhalten von MILC beschreibt, erlaubt die Assertion durch Betrachtung des relativen Fehlers geringe Abweichungen von den gemessenen Nachrichten.

`perfAssertion abs((18 * 4 * pow($nx * $ny * $nz * $nt / nMPIProcesses($MPI_COMM_WORLD), 0.75) / (${PA_TRANSFER_RATE} / 8000) + ${PA_TRANSFER_LATENCY} * microseconds) * 1616 * $pa_gfactor - MPITransferTime) / MPITransferTime < 0.1` **(update_h.c:46):** Analog zu der oberen Assertion überprüft diese die Nachrichtenanzahl und Größe des `eo_fermion_force_twoterms` Kernels mit den Werten aus [3].

Assertion in `fermion_links_from_site.c`

```
void restore_fermion_links_from_site(fermion_links_t *fl, int prec){
  /* ... */
  #pragma perfAssertion MPITime / WallTime < 0.2
  {
    #if FERM_ACTION == HISQ
    restore_fermion_links_hisq(fl, prec, phases_in, links);
    #else
    restore_fermion_links(fl, prec, phases_in, links);
    #endif
  }

  free(links);
}
```

`perfAssertion MPITime / WallTime < 0.2` **(fermion_links_from_site.c:33):** Die Assertion geht auf die Analyse mit Vampir zurück und sagt aus, dass der Anteil der MPI Zeit an der Gesamtzeit für den Aufruf der Prozedur im Gültigkeitsbereich kleiner als 20% ist. Abhängig von der Anzahl der Prozesse und damit der Gittergröße pro Prozess, ist dieses

Verhältnis entweder klein oder deutlich größer als der veranschlagte Schwellenwert und verdeutlicht somit das Zusammenspiel von Synchronisationsaufwand und parallelisierbarer Arbeit.

Als Gültigkeitsbereich wurde die Funktion `restore_fermion_links` gewählt, da nach der Analyse für diese der Anteil geringfügig höher ist als beispielsweise innerhalb der `update_h` Prozedur. Weiterhin wird der `load_fatlinks` Kernel [3] im Verlauf der Funktion aufgerufen. Folglich kann ein hoher Kommunikationsaufwand in dieser Prozedur einen Bottleneck darstellen.

Assertion in `d_congrad5_fn.c`

```
int ks_congrad( field_offset src, field_offset dest, Real mass,
        int niter, int nrestart, Real rsqmin, int prec,
        int parity, Real *final_rsq,
        imp_ferm_links_t *fn){
  /* ... */
  #pragma perfAssertion MPICollectiveTime < ($iters + 1) * 14.1794 * log(1.23264 * nMPIProcesses(
      $MPI_COMM_WORLD)) * microseconds
  {
    /* Solve the system */
    iters = ks_congrad_site( src, dest, &qic, mass, fn );
  }
  /* ... */
}
```

`perfAssertion MPICollectiveTime < ($iters + 1) * 14.1794 * log(1.23264 * nMPIProcesses( $MPI_COMM_WORLD)) * microseconds` **(d_congrad5_fn.c:132):** Das semianalytische Performance Modell [3] modelliert die Zeit für eine Reduktion mittels `MPI_Allreduce` innerhalb des `ks_congrad` Kernels durch den Logarithmus der Prozessanzahl. Die beiden hierfür notwendigen Konstanten ergeben sich aus den Messungen mittels Vampir. Zu Beginn und für jede Iteration innerhalb des Kernels wird eine Reduktion durchgeführt, wobei die Anzahl der Iterationen sich nicht arithmetisch ausdrücken lässt. Da der Wert von der Funktion zurückgegeben wird, kann die Anzahl der Reduktionen pro Aufruf des Kernels aus der Programmvariable `iters` abgeleitet werden. Hierbei ist zu beachten, dass Variablen erst zum Zeitpunkt der Auswertung der Assertion ausgelesen werden und nicht beim Eintritt in den Gültigkeitsbereich der Assertion.

5.3 Durchführung

5.3.1 Systembeschreibung

Die Evaluation wird auf dem Lichtenberg-Hochleistungsrechner der Technischen Universität Darmstadt ausgeführt, der zum Zeitpunkt dieser Arbeit[5] über 704 MPI Knoten verfügt, wobei jeder Knoten mit 32 Gigabyte Arbeitsspeicher und zwei 2,6 GHz getaktete Intel Xeon E5-2670 Prozessoren mit jeweils 8 Kernen ausgestattet ist. Mittels Turbo Boost kann eine maximale Frequenz von 3,3 GHz erreicht werden, Hyper-Threading ist deaktiviert. Jeder Kern besitzt einen 64 KB L1 und 256 KB L2 Cache, der gemeinsame L3 Cache umfasst 20 MB. Die Verbindung der einzelnen Knoten erfolgt über FDR-10 InfiniBand. Als Betriebssystem findet der SUSE Linux Enterprise Server 11 SP2 mit einem Kernel der Version 3.0.13 Verwendung. Für die Verwaltung der Aufträge wird das LSF Stapelverarbeitungssystem eingesetzt.

5.3.2 Konfiguration

MILC

Als Eingabekonfiguration für MILC wird eine modifizierte Variante der `small.in` Konfiguration des NERSC-6 MILC Benchmarks verwendet, die folgende Merkmale besitzt:

- eine Gittergröße von $8 \times 16 \times 16 \times 16$
- 4 Trajektorien
- 10 Schritte pro Trajektorie

[5] Stand: Dezember 2013

Abbildung 5.1 zeigt die vollständige Liste der eingesetzten Optionen.

```
prompt 0                  u0 0.8441
nflavors1 2               microcanonical_time_step 0.02
nflavors2 1               steps_per_trajectory 10
nx 8                      max_cg_iterations 250
ny 16                     max_cg_restarts 5
nz 16                     error_per_site .00001
nt 16                     error_for_propagator .0001
iseed 5682304             npbp_reps 1
warms 0                   prec_pbp 1
trajecs 4                 fresh
traj_between_meas 1       forget
beta 5.60
mass1 0.01
mass2 0.01
```

Abbildung 5.1: MILC Eingabekonfiguration

Für die Kompilierung von MILC 7.7.8.1 wurde der C Compiler der GNU Compiler Collection (`gcc`) in Version 4.7.3 und OpenMPI in Version 1.6.5 verwendet.

Wechselt ein Prozess zwischen unterschiedlichen Prozessorkernen oder Prozessoren, kann dies den Speicherzugriff durch Invalidierung des Cache (Cache Thrashing) oder Zugriff auf den Speichercontroller der anderen CPU negativ beeinflussen, weswegen mit der OpenMPI Option `-bind-to-core` die Prozesse an CPU-Kerne gebunden werden.

Performance Assertion Laufzeitumgebung

Bei der `MPITransferTime` handelt es sich um keine direkt messbare Metrik, stattdessen werden Systemparameter für die Abbildung genutzt. Für die Berechnung wird eine Übertragungsrate und Latenz benötigt, die über die Konfigurationsdatei festgelegt werden. Im Rahmen der Evaluation wird eine Übertragungsrate von 11200 Mbit/s und eine Latenz von 30μ verwendet. Diese Werte wurden mit Netgauge[6] mit der Option `-mode=mpi` und 32 Prozessen ermittelt. Dabei entsprechen die Werte bei einer Nachrichtengröße von 48 KiB der durchschnittlichen Übertragungsrate und der durchschnittlichen halben Round-Trip Time der Messung.

Stapelverarbeitungssystem

Das Stapelsystem ist für das Scheduling und die Verwaltung der Jobs (den Aufträgen der Benutzer) verantwortlich. Um Einflüsse von anderen Jobs während der Messung zu minimieren, wird für die Evaluation jeder Auftrag mit exklusivem Scheduling ausgeführt. Exklusive Jobs werden auf Knoten ausgeführt, die für die Bearbeitungszeitspanne keine weiteren Jobs ausführen. Da die gewählte MILC Konfiguration für die Ausführung auf einem Prozessor weniger als 10 Minuten erfordert, wird die `short` Warteschlange, die eine maximale Laufzeit von 30 Minuten erlaubt, benutzt.

5.3.3 Testablauf

Die Validierung der aufgestellten Performance Assertions wird in 10 Testserien durchgeführt, wobei eine Testserie je eine Ausführung von MILC mit $P = \{1, 2, 4, 8, 16, 32\}$ Prozesse(n) umfasst. Die Ergebnisse der einzelnen Testserien werden abhängig von der Prozessanzahl zu einer Datenreihe zusammengefasst, da das Verhalten von MILC und damit die erfasste Zeit variieren kann und auf diese Weise der Einfluss von starken Abweichungen gemindert wird.

Die Betrachtung des Overheads wird analog für die gleiche Anzahl an Prozessen P ausgeführt, um eine Einschätzung des Verhaltens abhängig von der steigenden Prozessanzahl zu erlauben. Für den Vergleich werden folgende Varianten von MILC verwendet:

- eine originale, unveränderte Version
- eine mit Score-P instrumentierte Version im Profiling-Modus
- eine mit Score-P instrumentierte Version im Tracing-Modus

[6] Homepage `http://htor.inf.ethz.ch/research/netgauge/`, zuletzt geprüft 20.11.13

Den Score-P Varianten werden über `SCOREP_TOTAL_MEMORY` jeweils 1024M Bytes an Speicher zugewiesen. Die Auswahl des Modus erfolgt über `SCOREP_ENABLE_PROFILING` und `SCOREP_ENABLE_TRACING`.

Alle Varianten werden jeweils zehnmal mit der gewählten Anzahl von Prozessen ausgeführt. Als Vergleichswert wird die von MILC gemessene Zeit der Berechnungsphase eingesetzt, die der Zeit für die Berechnung aller Trajektorien entspricht. Dementsprechend wird ein eventueller einmaliger Overhead bei der Initialisierung und Finalisierung des Programmes vernachlässigt, da die eigentlich Berechnung einen signifikant größeren Anteil an der Gesamtlaufzeit des Programmes besitzt. Ein solcher Overhead kann beispielsweise durch das Einlesen der Konfiguration, die Allokation von Pufferspeicher oder das Abspeichern von ermittelten Daten entstehen.

5.4 Ergebnisse

5.4.1 Performance Assertions

Die bei der Durchführung gesammelten Auswertungsberichte werden von einem Python-Skript weiterverarbeitet, das für jede Assertion und Prozessanzahl folgende Werte für den prozentualen Anteil von erfolgreichen zu insgesamt ausgewerteten Assertions berechnet:

- den Median
- das untere Quantil (0,25-Quantil)
- das obere Quantil (0,75-Quantil)
- das Minimum und Maximum

Diese abgeleiteten Werte lassen Rückschlüsse auf das allgemeine Auswertungsverhalten der einzelnen Assertions, dessen Streuung und die angenommenen Extremwerte zu. Durch die Verwendung des Medians wird der Einfluss von Ausreißern verringert. Die abgeleiteten Daten werden pro Assertion in einem Boxplot Diagramm visualisiert, bei dem die vertikalen Striche jeweils das Minimum und Maximum, der horizontale Strich den Median und die Box den Interquartilsabstand darstellen. Innerhalb des Interquartilsabstand liegen 50% aller Daten, da die Grenzen durch das untere und obere Quantil gegeben sind.

Assertions in `control.c`

`perfAssertion MPITime < 5.6*seconds & (nMPIProcesses($MPI_COMM_WORLD) == 1 -> MPITime/WallTime < 0.01)` **(control.c:60):** Für einen eingesetzten Prozess wurde die Assertion in allen Testdurchläufen erfolgreich ausgewertet. Bei der Verwendung von mehreren Prozessen kommt es häufiger zu Abweichungen, was bedeutet, dass einige Prozesse für die Trajektorienschleife eine höhere MPI Zeit benötigt haben. Da in diesen Fällen das untere Quantil mit dem Median und oberen Quantil zusammenfällt, waren weniger als 25% der Prozesse betroffen. Es kann keine Aussage darüber getroffen werden, wie hoch die aufgetretenen Abweichungen sind.

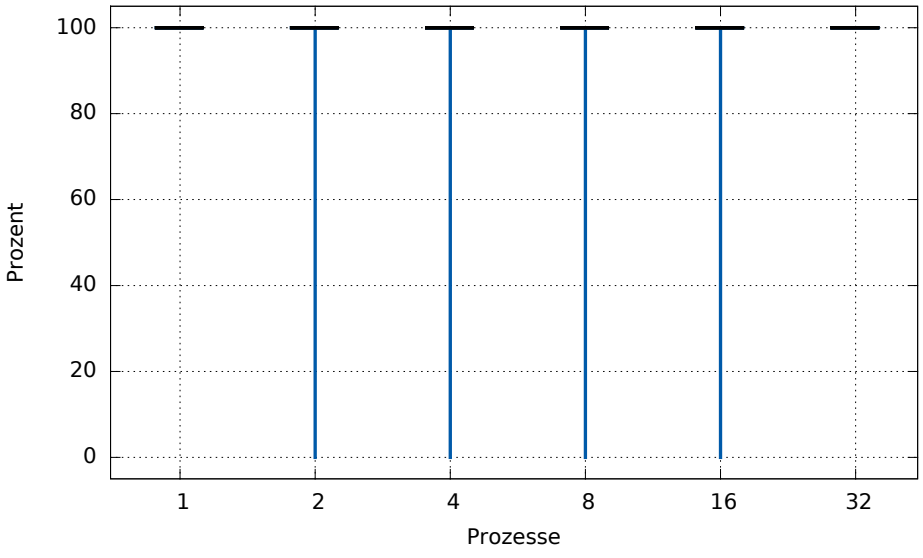

Abbildung 5.2: Diagramm für control.c:60

`perfAssertion MPIWaitTime < MPITransferTime` **(control.c:63):** Bei der Verwendung von nur einem Prozess werden keine Nachrichten übertragen. Folglich betragen die `MPIWaitTime` und `MPITransferTime` jeweils 0. Die Assertion trifft in diesem Fall nicht zu, da die Aussage $0 < 0$ falsch ist. Für die übrigen Prozessanzahlen trifft die Assertion unterschiedlich oft zu, der Median liegt immer bei einem Anteil von 100% erfolgreichen Auswertungen pro Prozess.

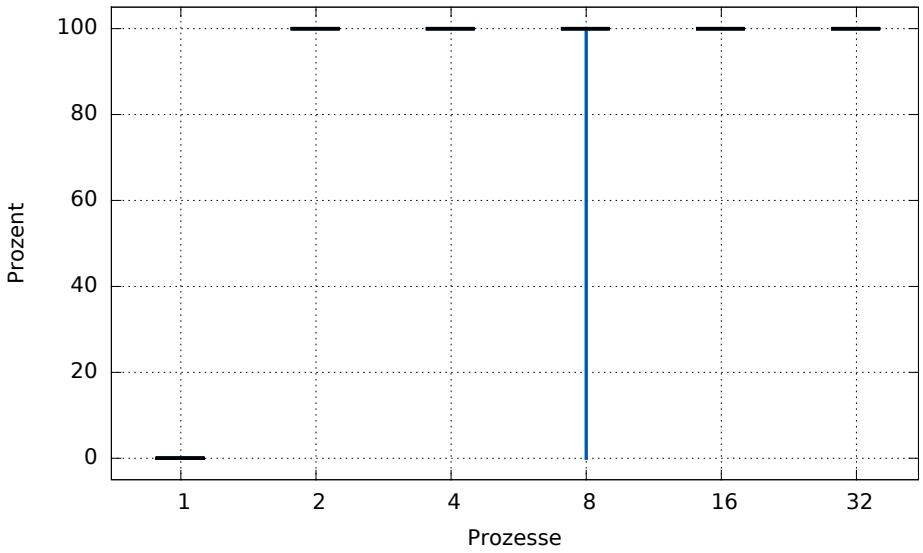

Abbildung 5.3: Diagramm für control.c:63

`perfAssertion WallTime <= 0` **(control.c:78):** Wie das zugehörige Diagramm zeigt, sind alle Auswertungen fehlgeschlagen. Die Implementierung hat folglich diesen allgemeinen Test bestanden, da die real verstrichene Zeit für einen nicht-leeren Abschnitt niemals kleiner oder gleich 0 sein sollte.

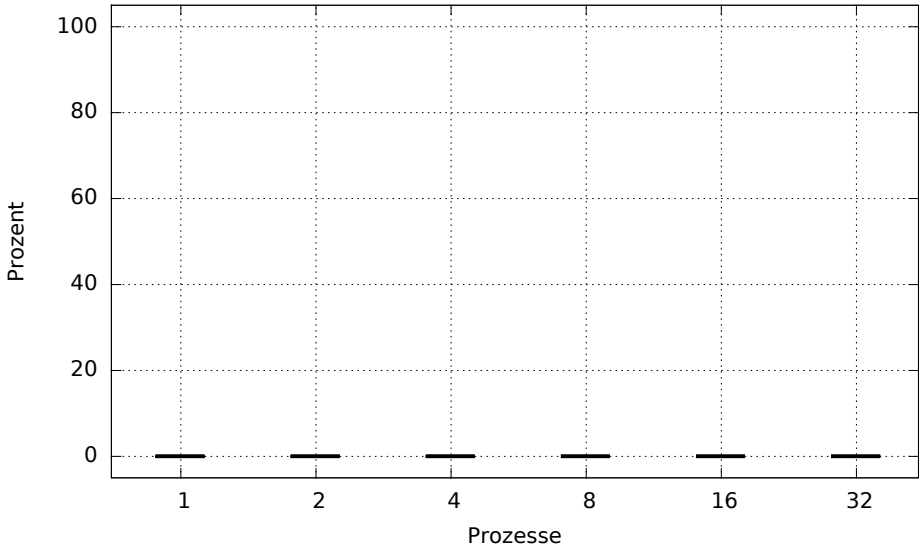

Abbildung 5.4: Diagramm für control.c:78

Assertion in `update.c`

`perfAssertion 7.8 * pow(nMPIProcesses($MPI_COMM_WORLD), -0.93) * seconds > WallTime`
(update.c:117): Für 1, 2, 16 und 32 eingesetzte Prozesse liegt der Median der erfolgreichen Auswertungen bei 100%, für 4 und 8 Prozesse treten jedoch signifikante Abweichungen auf, wobei keine Aussage getroffen werden kann, wie hoch die Differenz bei der Auswertung der Performance Assertions war. Dieser Trend wird durch einen Vergleich der in 5.4.2 ermittelten Berechnungszeiten bestätigt.

Zu beachten ist, dass die eingesetzten 8 Prozesse auf einem Rechenknoten ausgeführt wurden und Effekte des Netzwerks daher ausgeschlossen werden können. Stattdessen ist eine Beeinflussung durch das Speicherzugriffsmuster oder die Speicherbandbreite denkbar.

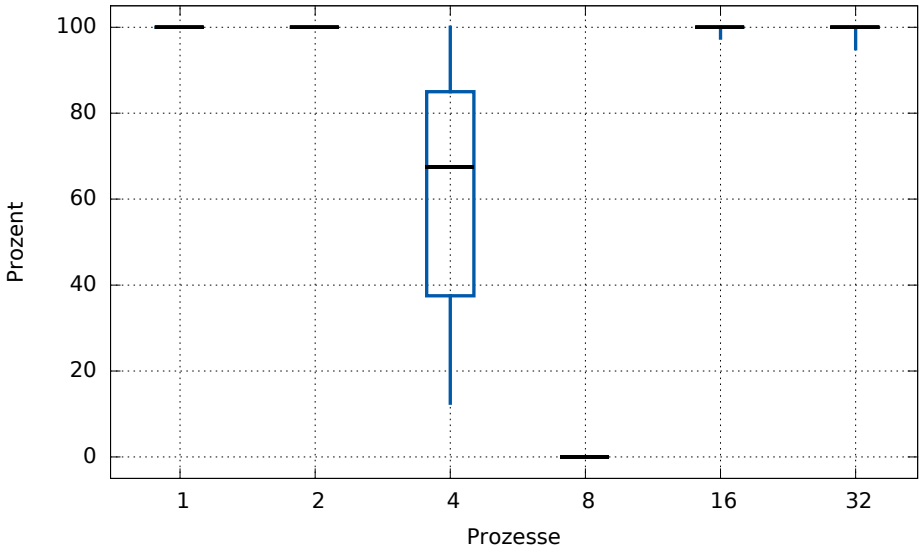

Abbildung 5.5: Diagramm für update.c:117

Assertion in `update_u.c`

`perfAssertion MPITime == 0` **(update_u.c:18):** Aufgrund der Tatsache, dass die Prozedur `update_u` lokale Berechnungen und keine Kommunikation über MPI ausführt, wurde die Assertion für alle Testfälle erfolgreich ausgewertet.

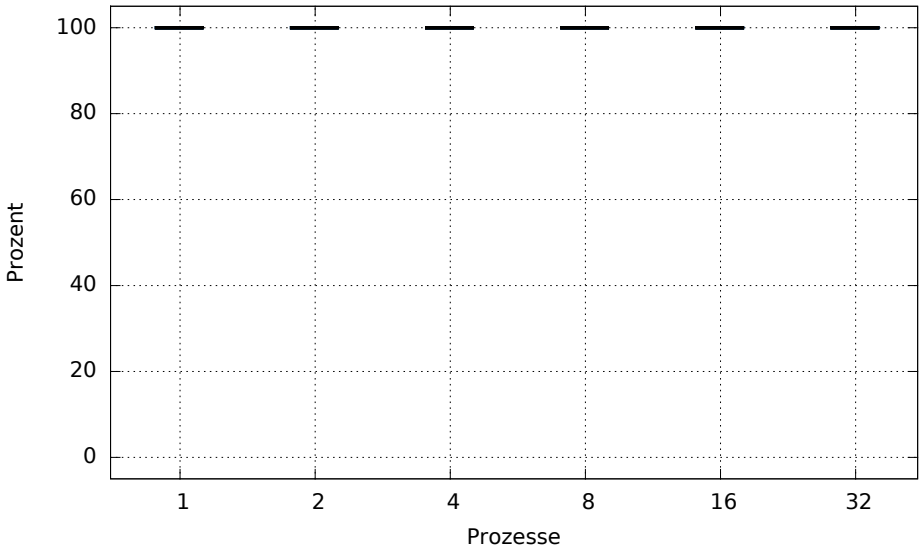

Abbildung 5.6: Diagramm für update_u.c:18

Assertions in `update_h.c`

`perfAssertion abs((18 * 4 * pow($nx * $ny * $nz * $nt / nMPIProcesses($MPI_COMM_WORLD), 0.75) / (${PA_TRANSFER_RATE} / 8000) + ${PA_TRANSFER_LATENCY} * microseconds) * 828 * $pa_gfactor - MPITransferTime) / MPITransferTime < 0.1` **(update_h.c:26):** Das Diagramm zeigt, dass die modellierte Anzahl und Größe der Nachrichten nur bei der Verwendung von mehr als 8 Prozessen in den akzeptierten Bereich fällt. Da in 5.2 bereits die Anzahl der ausgetauschten Nachrichten angepasst wurde, muss die Modellierung der Nachrichtengrößen genauer betrachtet werden.

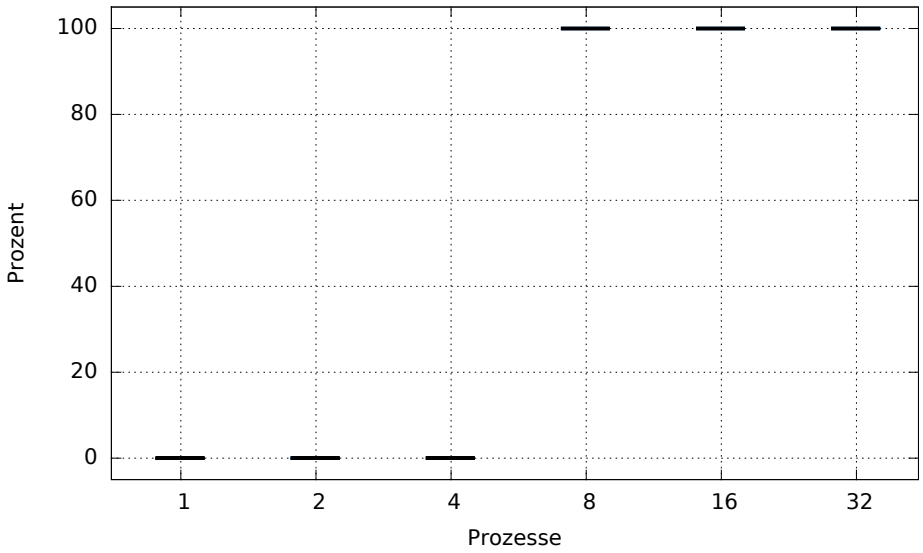

Abbildung 5.7: Diagramm für update_h.c:26

Die Analyse mit Vampir bestätigt, dass die Größen nicht mit den anhand der Formel berechneten Werten übereinstimmen. Nachfolgende Tabelle zeigt für den in dieser Assertion betrachteten `imp_gauge_force` Kernel exemplarisch für $P = \{2, 4, 8, 16\}$ Prozesse die nach dem Modell berechneten und real beobachteten Werte:

P	2	4	8	16
Modell	101,8 KiB	60,5 KiB	36 KiB	21,4 KiB
Real	144 KiB	72 KiB	36 KiB	18 oder 36 KiB

Für $P = 8$ stimmen die Nachrichtengrößen überein, für $P = 16$ verwendet MILC Nachrichten mit zwei unterschiedlichen Größen, wobei die kleineren häufiger versendet werden.

`perfAssertion abs((18 * 4 * pow($nx * $ny * $nz * $nt / nMPIProcesses($MPI_COMM_WORLD), 0.75) / (${PA_TRANSFER_RATE} / 8000) + ${PA_TRANSFER_LATENCY} * microseconds) * 1616 * $pa_gfactor - MPITransferTime) / MPITransferTime < 0.1` **(update_h.c:46):** Analog zu der vorherigen Performance Assertion wurde auch diese nur bei einer bestimmten Prozessanzahl erfolgreich ausgewertet. Allerdings liegen die Modellwerte hier für 2 und mindestens 16 verwendete Prozesse im akzeptierten Bereich.

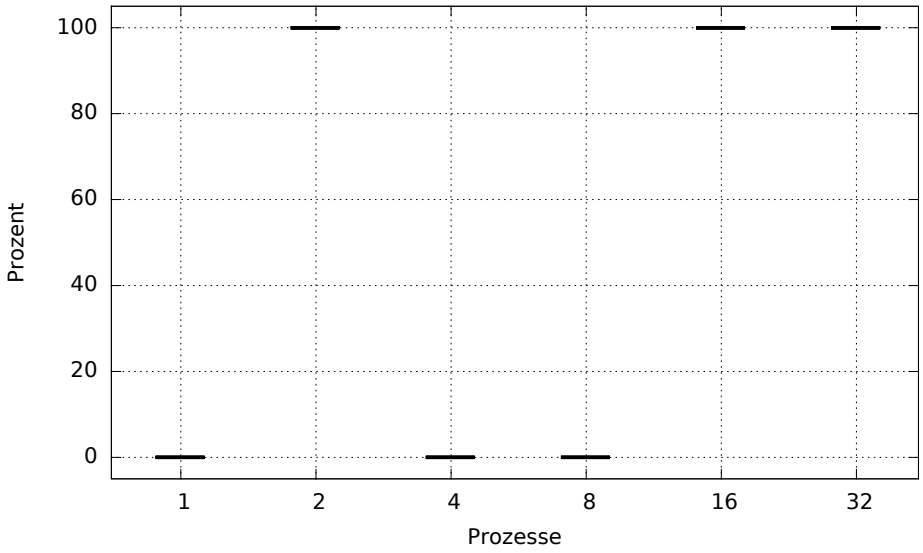

Abbildung 5.8: Diagramm für update_h.c:46

Eine Gegenüberstellung von modellierten und realen Nachrichtengrößen für $P = \{1, 2, 4, 8, 16\}$ Prozesse ergibt folgende Werte:

P	1	2	4	8	16
Modell	171,2 KiB	101,8 KiB	60,5 KiB	36 KiB	21,4 KiB
Real	0 KiB	96 KiB	48 KiB	24 KiB	12 oder 24 KiB

Bei $P = 1$ werden keine Nachrichten ausgetauscht. Die Auswertung schlägt fehl, weil die Formel der Assertion dies nicht berücksichtigt. Im Unterschied zu `update_h.c:26` stimmen bei der erfolgreichen Auswertung bei $P = 2$ die Werte nicht exakt überein, sondern weichen leicht voneinander ab. Außerdem sind die Nachrichtengrößen für $P = 4$ und $P = 8$ kleiner als im `imp_gauge_force` Kernel.

Je kleiner die Nachrichtengröße wird, desto schwächer wird auch der Einfluss der festgelegten Übertragungsrate und der Nachrichtengröße auf die berechnete Transferzeit, die stattdessen stärker von der konstanten Latenz und der Anzahl der Nachrichten abhängt. Auf diese Weise lässt sich die erfolgreiche Auswertung beider Assertions für $P = 16$ und $P = 32$ erklären. Bei der Wahl einer geringeren Latenz fällt dieser Effekt geringer aus und die Assertions werden für diese Prozessanzahl gegebenenfalls nicht erfolgreich ausgewertet.

Aus den Ergebnissen für die beiden Assertions in `update_h` folgt, dass die Nachrichtengrößen für die `imp_gauge_force` und `eo_fermion_force_twoterms` Kernel von der Modellformel abweichen und sich auch nicht untereinander gleichen. Beide Assertions wurden noch einmal separat mit der älteren, in [3] genutzten, Version (7.6.3) überprüft, mit dem selben Ergebnis. Aus diesem Grund lassen sich eventuelle Änderungen zwischen den beiden Versionen als Ursache ausschließen. Möglicherweise handelt es sich um eine bei der Modellierung bewusst getroffene Vereinfachung, die in [3] leider nicht erwähnt wird.

Assertion in `fermion_links_from_site.c`

`perfAssertion MPITime / WallTime < 0.2` **(fermion_links_from_site.c:33):** Aus dem Diagramm geht hervor, dass für vier oder weniger eingesetzte Prozesse der Anteil der MPI Zeit an der verstrichenen Zeit weniger als 20 Prozent beträgt. Bei dem Einsatz von mehreren Prozessen nimmt die Anzahl der Prozesse, die diesen Anteil überschreiten, jedoch zu, was sich zum Beispiel durch einen erhöhten Synchronisationsaufwand und einen geringeren Arbeitsanteil pro Prozess erklären lässt.

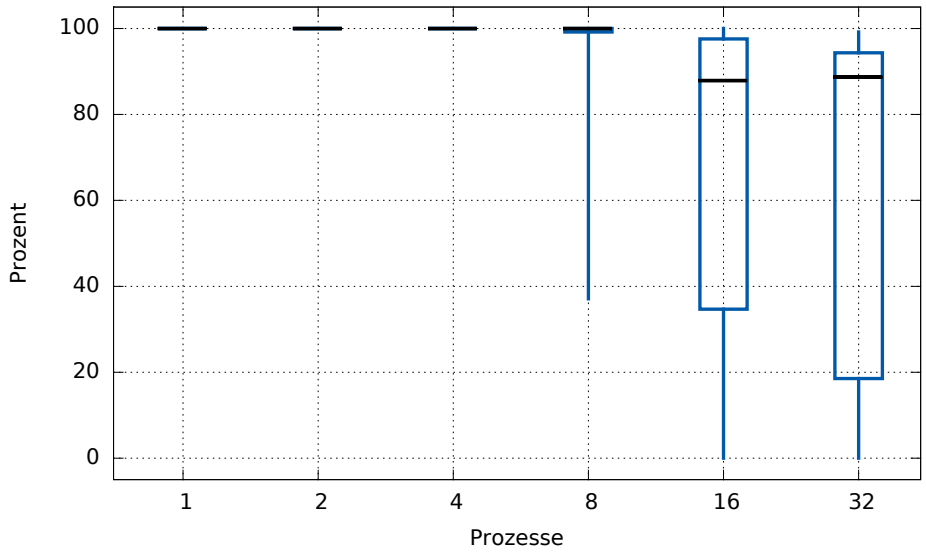

Abbildung 5.9: Diagramm für fermion_links_from_site.c:33

Assertion in `d_congrad5_fn.c`

`perfAssertion MPICollectiveTime < ($iters + 1) * 14.1794 * log(1.23264 * nMPIProcesses( $MPI_COMM_WORLD)) * microseconds` **(d_congrad5_fn.c:132):** Die Streuung des Anteils der erfolgreichen Auswertungen pro Prozess variiert unterschiedlich stark, nimmt aber leicht mit der Anzahl der Prozesse zu. Der Median liegt für alle Werte bei einem Anteil von mindestens 80%. Bei der Verwendung von einem Prozess ist die Zeit für kollektive Kommunikation nicht 0, da `MPI_Allreduce` auch in diesem Fall aufgerufen wird und einen geringen Overhead verursacht, der unter der berechneten Zeit liegt.

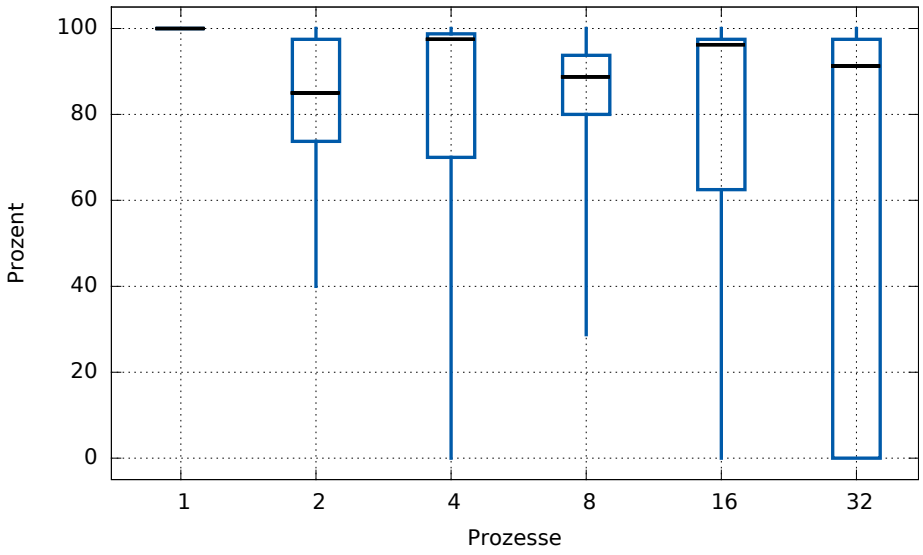

Abbildung 5.10: Diagramm für d_congrad5_fn.c:132

5.4.2 Overhead

Die Berechnungszeiten der ausgeführten Varianten werden durch ein Python-Skript aus der gespeicherten Standardausgabe extrahiert und jeweils die Stichprobenvarianz $s^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \overline{x})$ für die Beurteilung der Streuung der Messwerte und der Median bestimmt, da dieser weniger anfällig als das arithmetischen Mittel gegenüber Ausreißern ist. Die Mediane werden für $P = 1$, $P = 32$ und $P = \{2, 4, 8, 16\}$ aufgrund der unterschiedlichen Wertebereiche in separaten Diagrammen visualisiert. Die Stichprobenvarianz und prozentuale Abweichung der Mediane von denen der originalen Variante werden tabellarisch angegeben. Hierbei stehen die Begriffe in der Legende für folgende Varianten:

Original: das unveränderte Original

PA: Performance Assertions

Profiling: Score-P im Profiling-Modus

Tracing: Score-P im Tracing-Modus

	Median	prozentuale Abweichung	s^2
$P = 1$			
Original	399,83	0	8,03
PA	401,3	0,4	8,85
Profiling	411,99	3	5,2
Tracing	412,07	3,1	2,5
$P = 2$			
Original	216,42	0	0,28
PA	216,71	0,13	3,82
Profiling	221,62	2,4	1,22
Tracing	221,49	2,3	14,37
$P = 4$			
Original	122,24	0	1,65
PA	124,55	1,9	3,36
Profiling	126,89	3,8	1,43
Tracing	128,57	5,2	1,06

	Median	prozentuale Abweichung	s^2
$P = 8$			
Original	81,32	0	0,25
PA	81,73	0,5	0,17
Profiling	83,68	2,9	1,58
Tracing	83,32	2,5	1,36
$P = 16$			
Original	31,22	0	0,56
PA	31,3	0,2	1,16
Profiling	29,6	−5,2	0,18
Tracing	29,66	−5	36,37
$P = 32$			
Original	13,28	0	0,47
PA	13,54	1,9	15,19
Profiling	13,71	3,2	0,01
Tracing	13,94	4,9	0,15

Tabelle 5.1: Werte der Berechnungszeiten

Für $P = 1$ liegt der Median bei den Score-P Varianten deutlich höher als bei der originalen oder der mit Performance Assertions ausgestatteten Variante, wobei der Tracing-Modus eine marginal längere Berechnungszeit aufweist. Der Unterschied zwischen der originalen und der mit Performance Assertions modifizierten Variante ist gering: Der Median weicht um weniger als ein halbes Prozent ab.

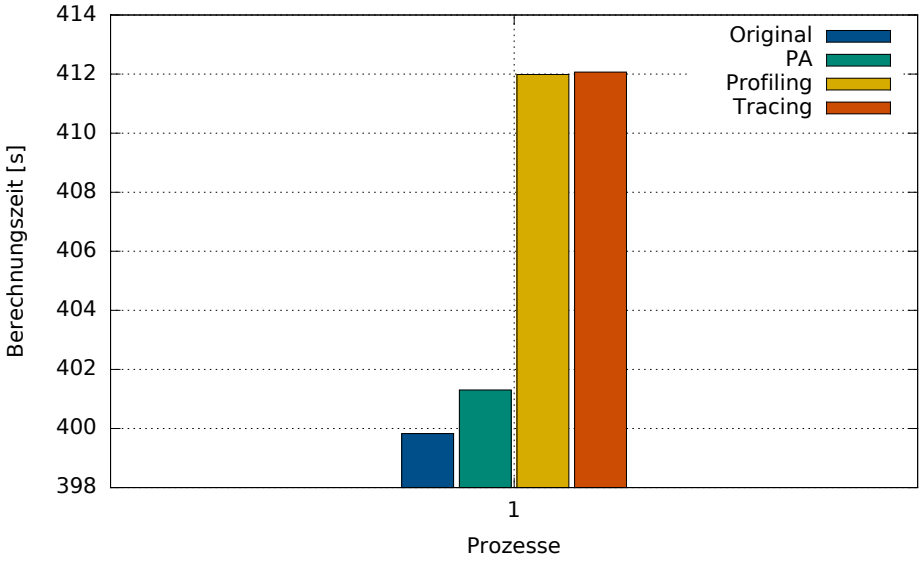

Abbildung 5.11: Berechnungszeit für $P = 1$

Für $P = 32$ nimmt der Median der Berechnungszeit in der Reihenfolge der Varianten zu. Die unveränderte Variante besitzt den niedrigsten Wert, Score-P im Tracing-Modus den höchsten. Die prozentuale Abweichung von der unveränderten Variante beträgt bei den Performance Assertions fast zwei Prozent und ist damit deutlich höher als bei $P = 1$ oder $P = 8$.

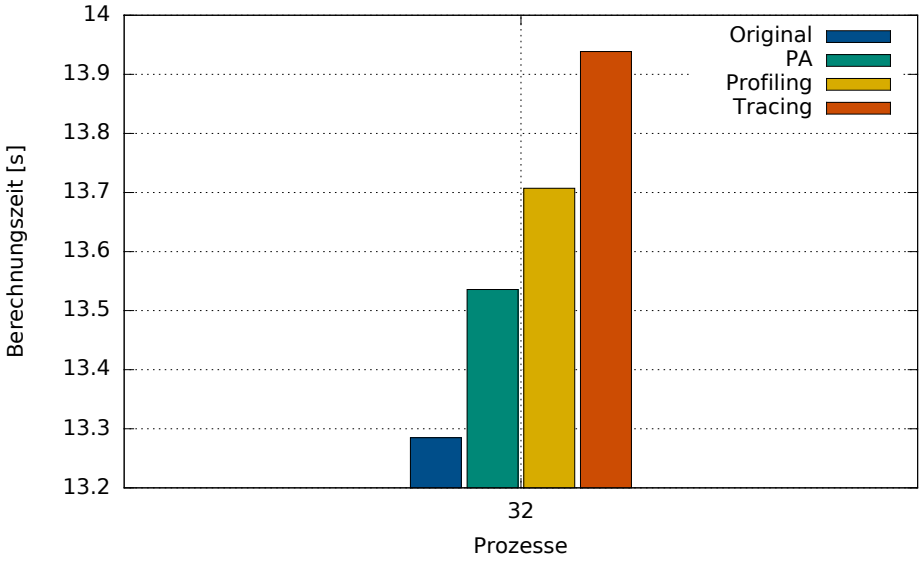

Abbildung 5.12: Berechnungszeit für $P = 32$

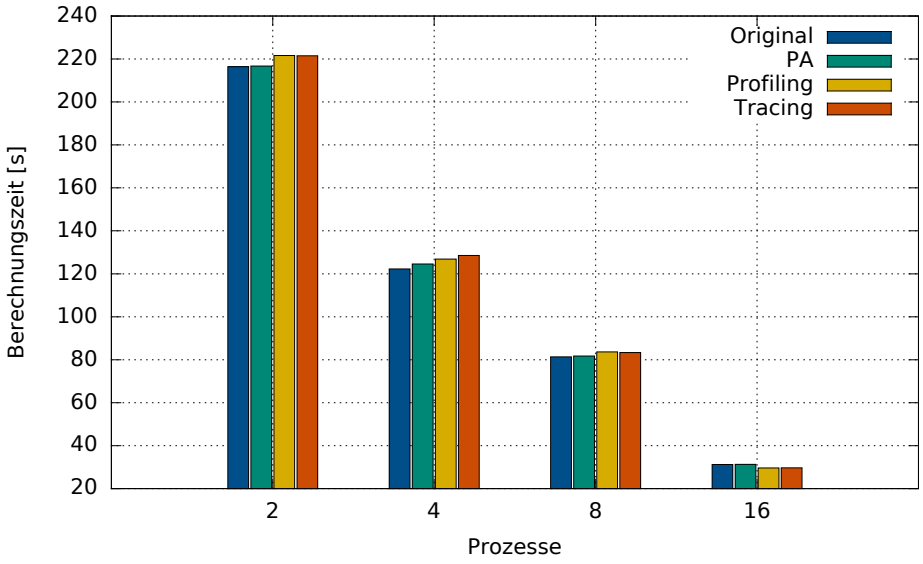

Abbildung 5.13: Berechnungszeit für $P = \{2, 4, 8, 16\}$

Bei der Verwendung von 16 Prozessen kommen die Score-P Varianten auf eine geringere Berechnungszeit als das Original und die Performance Assertion Version. Der Grund für dieses Verhalten kann ein durch die Instrumentierung hervorgerufenes Speicherzugriffsmuster sein, das durch die Auslastung aller Prozessorkerne bei $P = 16$ den Cache effizienter nutzt. Die übrigen Prozessanzahlen folgen den zuvor bei $P = 1$ und $P = 32$ beobachteten Trend.

Der Overhead der Performance Assertion Implementierung für die `su3_rmd` Applikation von MILC lässt sich aufgrund der Ergebnisse wie folgt klassifizieren:

- Im Vergleich zu einer Instrumentierung mit Score-P fällt der Overhead bei der Verwendung von Performance Assertions deutlich geringer aus: Abgesehen von $P = 16$ ist der gemessene Overhead um einen Faktor von 1,7 bis 18 niedriger. Der Grund hierfür ist, dass beim Einsatz von Performance Assertions weniger Daten erfasst werden müssen. Eine Performance Assertion führt nur die Messungen durch, die für die Auswertung der getroffenen Aussage erforderlich sind. Beim Profling wird dagegen das Zeitverhalten einer größeren Menge von Funktionen erfasst und Tracing erfordert die Aufzeichnung einer mitunter großen Anzahl festgelegter Events, um den Programmablauf rekonstruieren zu können.
- Der Unterschied zu der unveränderten Originalversion ist als gering einzuschätzen, da für die betrachteten Fälle die Zeiten um maximal 2% von der ursprünglichen Berechnungszeit abweichen.
- Die prozentualen Abweichungen nehmen nicht mit der Anzahl der eingesetzten Prozesse zu, variieren dafür aber unterschiedlich stark.
- Der Overhead bei der Verwendung von Performance Assertions ist abhängig von der Anzahl der Assertions und deren Auswertungsanzahl. Im Rahmen der Evaluation führte jeder Prozess unabhängig von der Prozessanzahl insgesamt 493 Auswertungen aus.

6 Fazit

6.1 Zusammenfassung

Im Rahmen dieser Bachelor-Arbeit wurde eine Implementierung von Performance Assertions für MPI Metriken erarbeitet. Bei Performance Assertions handelt es sich um ein Konzept zur automatischen Validierung von benutzerdefinierten Aussagen über das Laufzeitverhalten eines Programmes. Die Aussagen werden in der in Kapitel 3 definierten Sprache formuliert, deren Syntax und Semantik der Programmiersprache C ähnelt. Der Fokus dieser Realisierung liegt auf der Betrachtung und Charakterisierung von MPI Programmen, die durch Austausch von Nachrichten auf der Prozessebene parallele Berechnungen durchführen. Die eingesetzte Sprache erlaubt die Verkettung von logischen, relationalen sowie arithmetischen Ausdrücken, sodass Aussagen über die bereitgestellten Metriken formuliert werden können. Diese Aussagen können unter anderem auf den Inhalt von Programmvariablen zurückgreifen und einfache mathematische Funktionen, wie beispielsweise die Quadratwurzelfunktion, nutzen. Die verfügbaren Metriken basieren hauptsächlich auf Zeitmessungen und erlauben die Klassifikation des MPI Zeitverhaltens durch Betrachtung der für Teilmengen der Kommunikationsprozeduren gemessenen Zeit. Im Unterschied dazu versucht die `MPITransferTime` Metrik die Übertragungszeit der beobachteten Nachrichten anhand ihrer Größe und einer eingestellten Übertragungsrate linear zu approximieren.

Die Implementierung der Performance Assertions wird in Kapitel 4 beschrieben und setzt sich aus einem Compiler, einer Laufzeitumgebung und einem Generator zusammen. Der Compiler übersetzt die als Compiler-Pragma formulierten Assertions jeweils in einen Auswertungsausdruck und instrumentiert das Eingabeprogramm, indem Aufrufe von Funktionen der Laufzeitumgebung hinzugefügt werden, sodass die benötigten Daten gemessen und die Performance Assertions zur Laufzeit ausgewertet werden. Bei der Laufzeitumgebung handelt es sich um eine statische Bibliothek, die für die Messung der Metriken und die Verwaltung der Auswertungsergebnisse zuständig ist. Die Erfassung der MPI Metriken greift auf das MPI Profiling Interface [7] zurück, das es erlaubt, Neudefinitionen für die MPI Funktionen zu erstellen und die ursprünglichen Funktionen aufzurufen. Die automatische Erstellung dieser Funktionsdefinitionen übernimmt der Generator, welcher die Funktionen aufgrund ihrer Bedeutung kategorisieren muss.

In Kapitel 5 wird die Implementierung unter Verwendung einer MILC Applikation aus dem Bereich der Quantenchromodynamik evaluiert. Hierzu werden Performance Assertions aufgestellt, die sowohl auf einem bestehenden semianalytischen Performance Modell [3] als auch auf Erkenntnissen aus einer Analyse mit Vampir basieren. Durch die Evaluation konnte gezeigt werden, dass die Modellierung der Nachrichtengrößen in [3] für die Punkt-zu-Punkt Kommunikation zweier betrachteter Kernel unzureichend ist, da die tatsächlichen Werte deutlich von den erwarteten abweichen. Ein Vergleich der von MILC gemessenen Berechnungszeiten zwischen einer unveränderten, der mit dieser Implementierung und den formulierten Performance Assertions instrumentierten sowie einer mit Score-P (siehe 2.2) instrumentierten Variante ergab, dass der Overhead für die Assertions signifikant kleiner als bei der Instrumentierung und Vermessung mit Score-P ist. Insbesondere hat die Evaluation gezeigt, dass Performance Assertions ein geeignetes Konzept zur automatisierten Überprüfung von Performance Modellen für Programme sind, die MPI für die Ausnutzung von Parallelität auf Prozessebene nutzen.

6.2 Ausblick

Die Funktionalitäten und Eigenschaften der Performance Assertion Implementierung werden wesentlich von der Sprache bestimmt, in der die Aussagen formuliert werden. Ergänzt man die Sprache um neue Merkmale, hat dies direkte Auswirkungen auf die Ausdrucksstärke und Nutzbarkeit. Durch ein exklusives Oders kann zum Beispiel leichter formuliert werden, dass bei einer Verkettung zweier Aussagen nur eine der beiden zu wahr auswerten soll (entweder a oder b). Eine Konstante `ApplicationTime`, welche die Anwendungszeit als Differenz aus `WallTime` und `MPITime` beschreibt, könnte beispielsweise genutzt werden, um intuitiver Aussagen über die eigentliche Berechnungszeit ohne den anfallenden Kommunikationsaufwand zu treffen. Da diese Konstante eine reine Komfortfunktion ist und für ein exklusives Oder keine konkreten Anwendungsfälle bekannt sind, wurden beide Konzepte innerhalb dieser Arbeit nicht umgesetzt. Durch die Aufnahme weiterer Metriken kann das mögliche Anwendungsgebiet der Performance Assertions vergrößert werden. In ihrem vorgestellten Konzept benutzen Vetter und Worley [25] die Programmierschnittstelle PAPI [5] zum Auslesen von Hardware Countern, speziellen Zählerregistern, die Ereignisse der Hardware akkumulieren und setzen diese für die

Charakterisierung des Laufzeitverhaltens sequentieller Programme ein. Für eine Integration in diese Implementierung bietet sich diese Schnittstelle ebenfalls an, allerdings müssen mehrere Aspekte berücksichtigt werden:

- Die Wahl der verfügbaren Ereignisse, die als Metrik bereitgestellt werden sollen: PAPI stellt mehrere vordefinierte Ereignisse bereit. Daher muss ermittelt werden, welche dieser Ereignisse am geeignetsten sind, um das Laufzeitverhalten eines Programmes abzuleiten und als Metriken in die Sprache aufgenommen werden sollten.
- Ein Prozessor kann nur eine begrenzte Anzahl von Ereignissen simultan erfassen. Durch die Verwendung der PAPI Multiplexing Funktion, bei der die verfügbaren Zählerressourcen auf mehrere Ereignisse über die Zeit hinweg aufgeteilt werden, kann dieses Problem umgangen werden.
- Die Performance Assertion Laufzeitumgebung sollte die Messdetails, wie die Verwaltung der aktiven Ereignisse und eventuell notwendiges Multiplexing, vor dem Benutzer verbergen, damit dieser sich auf das Schreiben der Assertions konzentrieren kann.

Die `MPITransferTime` ist eine Schätzung der Übertragungszeit der Punkt-zu-Punkt Kommunikation durch die Auswertung einer linearen Funktion, die sich aus der Übertragungsrate und einer Latenz ergibt. Diese Heuristik kann verbessert werden, um einen genaueren Wert zu erhalten, indem beispielsweise das LogGP Modell [1], dessen Eignung zur Modellierung von MPI Kommunikation von Al-Tawil und Moritz [23] untersucht wurde, für die Abbildung der Kommunikation eingesetzt wird. Bauer et al. [3] verwenden dieses Modell ebenfalls für ihr aufgestelltes Performance Modell, nehmen jedoch einige Anpassungen vor:

- Unterscheidung zwischen Inter- und Intra-Rechnerknoten Kommunikation
- Berücksichtigung der ausführenden Hardware, um Netzwerk-Congestions miteinzubeziehen

Eine Verfeinerung der `MPITransferTime` könnte daher darin bestehen, dem Benutzer die Definition einer Funktion zur Auswertung der Heuristik zu erlauben, entweder als Compiler-Pragma oder als Eintrag in der Konfigurationsdatei. Die Transferzeit ist für die Analyse von MPI Programmen von Interesse, da sie eine Klassifikation des „Overlaps“, das heißt der Überlappung von nichtblockierender Kommunikation und Berechnung erlaubt, die das Laufzeitverhalten einer Applikation positiv beeinflussen kann [4]. Eine direkte Messung der Übertragungszeit ist wie in 3.3 beschrieben nicht ohne weiteres möglich, da auch kontinuierliches Polling des Übertragungsstatus keine exakten Messergebnisse garantieren kann und die Nutzung der dafür benötigten Ressourcen einen Effekt auf das ursprüngliche Ausführungsverhalten des betrachteten Programmes haben kann.

Mit Version 3.0 des MPI Standards wurde das MPI Tool Information Interface [7, S. 561-590] eingeführt, das den Zugriff auf Kontroll- und Performance-Variablen der MPI Implementierung erlaubt. Die so verfügbaren Daten können gegebenenfalls für weitere oder bestehende Metriken verwendet werden. Welche Variablen bereitgestellt werden, ist der Implementierung überlassen, was eine feste Verwendung für eine portable Performance Assertion Implementierung erschwert. Hinzu kommt, dass der Fortschritt der MPI Implementierungen bei der Umsetzung der neuen Version unterschiedlich ist.

Die Implementierung des Compilers setzt einen manuell geschriebenen Recursive-descent Parser ein (siehe 4.3.2). Die Entscheidung, keinen Parsergenerator, wie zum Beispiel ANTLR[1], zu verwenden, basierte auf der Annahme, dass der ROSE `AstFromString` Namespace Vorteile bei der Konstruktion des von ROSE verwendeten AST bieten würde. Schlussendlich wurden jedoch nur allgemeine Funktionen zur Behandlung von Zeichenketten und eine Funktion zum Einlesen von Variablenreferenzen verwendet. Wäre der Namespace als Klasse mit überschreibbaren Funktionen zum Parsen der Teilausdrücke realisiert worden, hätten selbige in einer erbenden Klasse entsprechend den Anforderungen der Performance Assertion Sprache spezialisiert werden können. Da bei einer Erweiterung der Sprache die Wartbarkeit komplexer und der Quellcode fehleranfälliger wird, empfiehlt sich auf längere Sicht der Einsatz eines Parsergenerators, der aus einer Grammatikspezifikation automatisch einen Parser generiert. Ein auf diese Weise erhaltener Parser kann leistungsstärker als die aktuelle Implementierung sein, da diese auf einen „Lookahead“ bei der Wahl der Produktion verzichtet und es daher zu Backtracking kommen kann. Zu beachten ist, dass die Grammatik in ihrer vorgestellten Form nicht in $LL(1)$ ist (siehe gemeinsames Präfix von `PA_ConfigValueReference` und `PA_VariableReference` in 3.2).

Bei der Durchführung der Evaluation hat sich gezeigt, dass eine manuelle Auswertung der Ergebnisberichte der einzelnen Prozesse eines Testlaufs ab einer bestimmten Anzahl aufwendig wird und die Übersicht über Abweichungen unter den Prozessen verloren geht. Dieses Problem kann abgemildert werden, indem eine Funktion eingeführt wird, die vor dem Aufruf von `MPI_Finalize` die Auswertungsergebnisse aller Prozesse austauscht und zusammengefasst abspeichert. Dabei können Kenngrößen, wie Quantile und Median, abgeleitet werden.

In seltenen Fällen kann für einzelne Prozesse einer Programmausführung das automatisch generierte Präfix für den Dateinamen des Berichts von dem der übrigen Prozesse abweichen, da die Zeit auf den Rechenknoten nicht perfekt synchronisiert werden kann und die Prozesse zu leicht unterschiedlichen Zeitpunkten terminieren. Eine Lösungsmöglichkeit

[1] Homepage `http://www.antlr.org/`, zuletzt geprüft am 26.11.13

besteht darin, das Präfix vor dem Aufruf von `MPI_Finalize` auf einem Prozess zu generieren und an die übrigen Prozesse zu verschicken.

Cluster-Systeme, die aus mehreren verbundenen Mehrprozessorsystemen aufgebaut sind, bieten aufgrund der pro Rechenknoten vorhandenen Ressourcen je nach Anwendung das Potenzial, MPI mit auf Threads basierten Programmierschnittstellen, wie OpenMP, zu kombinieren [21, 13]. Die entwickelte Laufzeitumgebung unternimmt keine expliziten Maßnahmen, um Threadsicherheit zu gewährleisten. Betroffen von eventuellen Race Conditions sind das MPI Messsystem und die Verwaltung der Auswertungsergebnisse der Assertions. Während letztere Komponente beispielsweise durch einen Mutex geschützt werden kann, erfordert die Erfassung der MPI Metriken für mehrere aktive Threads weitreichende Änderungen: Um eine Zuordnung von Messwert und aktiver Performance Assertion zu erreichen, genügen die eingesetzten Metrik-Stacks nicht mehr. Die Messwerte von Threads, die innerhalb des Gültigkeitsbereichs einer Assertion erstellt und beendet werden, müssen für die Auswertung der Assertion akkumuliert werden. Der geänderte Programmfluss lässt sich beispielsweise als gerichteter Graph modellieren, wobei jede Kante einem Thread entspricht. Bei der Zusammenführung von Kanten an einem Knoten, das heißt dem Beenden des Threads, müssen die Werte der einzelnen Kanten miteinander kombiniert und das Ergebnis der weitergehenden Kante zugewiesen werden. Die Realisierung erfordert eine Instrumentierung des Programmes abhängig von der eingesetzten Threadingtechnologe (OpenMP, pthreads, C++11 Threads).

Literatur

[1] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser und Chris Scheiman. „LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation“. In: *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '95. ACM, 1995, S. 95–105.

[2] Michal Bali. *Drools JBoss Rules 5.0: Developer's Guide*. Packt Publishing, 2009.

[3] G. Bauer, S. Gottlieb und T. Hoefler. „Performance Modeling and Comparative Analysis of the MILC Lattice QCD Application su3_rmd“. In: *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2012, S. 652–659.

[4] Ron Brightwell, Rolf Riesen und Keith D. Underwood. „Analyzing the Impact of Overlap, Offload, and Independent Progress for Message Passing Interface Applications“. In: *International Journal of High Performance Computing Applications* 19.2 (2005), S. 103–117.

[5] S. Browne et al. „A Portable Programming Interface for Performance Evaluation on Modern Processors“. In: *International Journal of High Performance Computing Applications* 14.3 (2000), S. 189–204.

[6] James Cownie und William Gropp. „A Standard Interface for Debugger Access to Message Queue Information in MPI“. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Bd. 1697. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, S. 51–58.

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. High Performance Computing Center Stuttgart, 2012.

[8] Torsten Hoefler, William Gropp, William Kramer und Marc Snir. „Performance Modeling for Systematic Performance Tuning“. In: *State of the Practice Reports*. SC '11. ACM, 2011, 6:1–6:12.

[9] Henry Hoffmann et al. „Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments“. In: *Proceedings of the 7th international conference on Autonomic computing*. ICAC '10. ACM, 2010, S. 79–88.

[10] K.A. Huck et al. „Capturing Performance Knowledge for Automated Analysis“. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.* 2008, S. 1–10.

[11] Kevin A. Huck und Allen D. Malony. „PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing“. In: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. SC '05. IEEE Computer Society, 2005.

[12] Institute of Electrical and Electronics Engineers. „IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7“. In: *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)* (2008).

[13] Christian Iwainsky, Samuel Sarholz, Dieter an Mey und Ralph Altenfeld. „Leveraging Multicore Cluster Nodes by Adding OpenMP to Flow Solvers Parallelized with MPI“. In: *High Performance Computing Systems and Applications*. Bd. 5976. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, S. 62–69.

[14] Andreas Knüpfer et al. „Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir“. In: *Tools for High Performance Computing 2011*. Springer, 2012, S. 79–91.

[15] Andreas Knüpfer et al. „The Vampir Performance Analysis Tool-Set“. In: *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2008, S. 139–155.

[16] Chunhua Liao et al. „OpenUH: An Optimizing, Portable OpenMP Compiler“. In: *Concurrency and Computation: Practice and Experience* 19.18 (2007), S. 2317–2332.

[17] Timothy G. Mattson, Beverly A. Sanders und Berna L. Massingill. *Patterns for Parallel Programming*. Pearson Education, 2004.

[18] Dieter an Mey et al. „Score-P: A Unified Performance Measurement System for Petascale Applications“. In: *Competence in High Performance Computing 2010*. Springer, 2012, S. 85–97.

[19] Sharon E. Perl und William E. Weihl. „Performance Assertion Checking“. In: *Proceedings of the fourteenth ACM symposium on Operating systems principles*. SOSP '93. ACM, 1993, S. 134–145.

[20] Dan Quinlan und Chunhua Liao. „The ROSE Source-to-Source Compiler Infrastructure“. In: *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*. Okt. 2011.

[21] R. Rabenseifner, G. Hager und G. Jost. „Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes“. In: *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 2009, S. 427–436.

[22] Sameer S. Shende und Allen D. Malony. „The Tau Parallel Performance System“. In: *International Journal of High Performance Computing Applications* 20.2 (2006), S. 287–311.

[23] Khalid Al-Tawil und Csaba Andras Moritz. „Performance Modeling and Evaluation of MPI“. In: *Journal of Parallel and Distributed Computing* 61.2 (2001), S. 202–223.

[24] Linda Torczon und Keith Cooper. *Engineering a Compiler*. 2. Aufl. Morgan Kaufmann Publishers Inc., 2011.

[25] Jeffrey S. Vetter und Patrick H. Worley. „Asserting Performance Expectations“. In: *Supercomputing, ACM/IEEE 2002 Conference*. IEEE. 2002, S. 33–33.