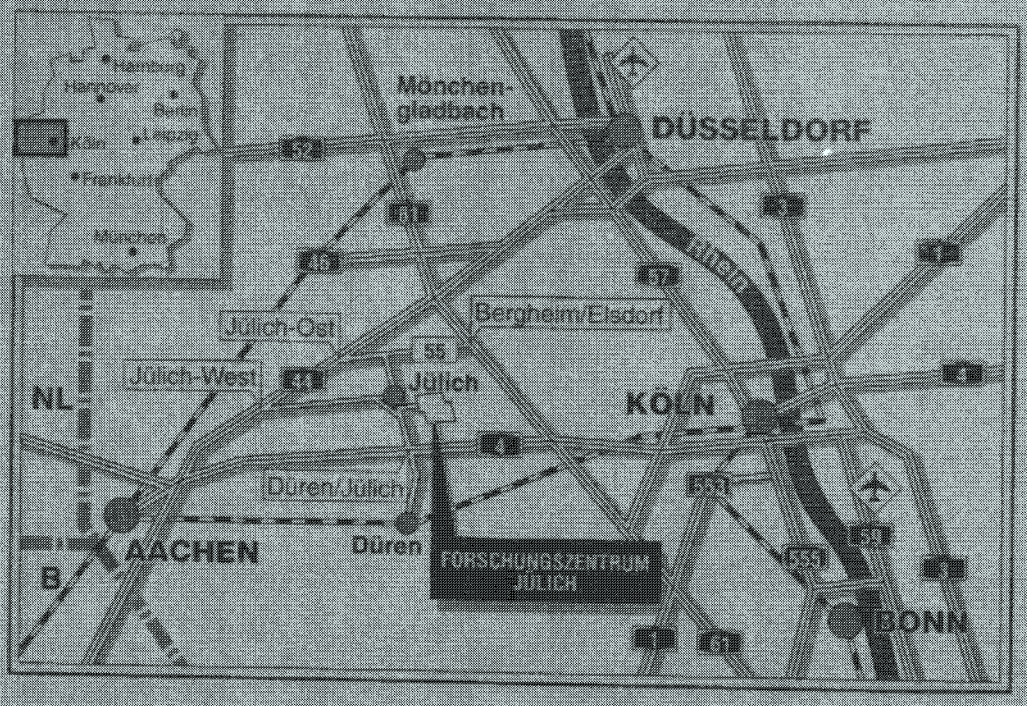
Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Versionsverwaltung zur Unterstützung der Analyse und Optimierung von SVM-Fortran Programmen

Carsten Pitz

JUL-3179



Berichte des Forschungszentrums Jülich ; 3379
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3379

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461/61-6102 · Telefax: 02461/61-6103 · e-mail: zb-publikation@fz-juelich.de

Abstract

By using the shared memory programming model it is possible to parallelize applications step by step (in contrast to the message passing programming model), because there is no need for an initial data distributing. SVM-Fortran is a programming language for shared vitual memory architectures. Due to the high abstraction level of SVM-Fortran a sophisticated performance analysis tool is needed. OPAL (Optimizer and Locality Analyzer) is a source code based performance analysis tool developped for SVM-Fortran. It supports selective instrumentation to reduce the impact of measuring on the runtime. So also the analyzing is done incrementally.

Practice shows, that this approach leads to a large number of program versions and trace files, which are difficult to handle. To evaluate the efficiency of an optimization step, an easy to use and flexible access to all program versions and traces is a must. So a version management facility has been integrated into OPAL, which leads the user to document his work by adding comments to the program versions and the traces. The comment consists of a single line abstract and a structured, detailed part. The single line abstracts are used to help selecting a program version or a trace and to show what is actually loaded, while the main purpose of the structured, detailed part is documentation.

Further features have been included into OPAL to support the complete development cycle consisting of analyzing, editing, compiling and executing of SVM-Fortran programs under a single graphical interface.

Kurzfassung

Im Shared-Memory-Programmiermodell können im Gegensatz zum Message-Passing-Programmiermodell Anwendungen schrittweise parallelisiert werden, da der globale Adreßraum eine initiale Datenaufteilung unnötig macht. SVM-Fortran ist eine Programmiersprache für Shared-Virtual-Memory-Architekturen. Der hohe Grad an Abstraktion den SVM-Fortran bietet bedingt die Verwendung von komplexen Performance-Analyse-Werkzeugen. OPAL (Optimizer and Locality Analyzer) ist ein quelltextbasiertes Analysewerkzeug, das für SVM-Fortran entwickelt wurde. Es unterstützt selektive Instrumentierung, um den Einfluß der Messung auf die Laufzeit zu reduzieren. Somit ist auch die Analyse inkrementell.

Wie die Praxis zeigt führt dieses Vorgehen zu einer schwer überschaubaren Anzahl verschiedener Programmversionen mit den zugehörigen Performance-Daten. Zur Effizienzbeurteilung einzelner Optimierungsschritte wird jedoch ein einfacher flexibler Zugriff auf die einzelnen Programmversionen und Performance-Daten benötigt. Hierzu wurde OPAL um eine Versionsverwaltung erweitert, die den Anwender die Dokumentation seiner Arbeit erleichtert. Der Kommentar besteht aus einem einzeiligen Kurzkommentar und einem strukturierten, ausführlichen Teil. Der einzeilige Kurzkommentar dient zur Auswahl von Programmversionen und Messungen und zur Anzeige der momentan geladenen Version. Dieser wird durch den ausführlichen Kommentar ergänzt.

Durch zusätzliche Erweiterungen wird OPAL zu einer integrierten Entwicklungsumgebung ausgebaut, die den vollständigen Zyklus von Analyse, Modifikation, Übersetzung und Ausführung von SVM-Fortran-Programmen innerhalb einer Bedienoberfläche unterstützt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Parallelrechner haben sich mit der Zeit in einigen Bereichen der Datenverarbeitung etabliert. Zu diesen Bereichen gehören, neben wissenschaftlichen Anwendungen, Transaktionssysteme und Prozeßsteuerungen. Dennoch ist die Euphorie der letzten Jahre einer nüchternen Betrachtung von Parallelrechnern gewichen. Ein Grund hierfür ist der lange Zeit unterschätzte Aufwand für die Portierung von Anwendungen auf parallele Architekturen.

1.1 Motivation

Der hohe Rechenleistungsbedarf wissenschaftlicher Anwendungen, die zum Teil Rechenleistungen benötigen, die zur Zeit verfügbare Supercomputer nicht erreichen, fördert die Entwicklung immer leistungsstärkerer Supercomputer [1, 2, 3, 4].

Es gibt verschiedene Wege, die Rechengeschwindigkeit eines Rechners zu erhöhen. Der klassische Weg ist die Erhöhung des Prozessortaktes. Die Rechenleistung eines Prozessors steigt linear mit der Taktfrequenz. Werden für einen Prozessor die Herstellungskosten über die Taktrate aufgetragen, wächst die Funktion zunächst sublinear, geht aber ab einer bestimmten Taktrate in ein superlineares Wachstum über. Daher ist eine Erhöhung des Prozessortaktes nur in Grenzen ökonomisch sinnvoll. Ein Weg, der in vielen Prozessoren genutzt wird, ist prozessorinterner Parallelismus. Jedoch der zur Zeit favorisierte Weg sind Parallelrechner.

Moderat parallele Supercomputer (Cray T90, NEC SX-3, Fujitsu VP2600, ...) haben einen physikalisch gemeinsamen Hauptspeicher, jedoch wird bei wachsender Prozessorzahl ein physikalisch gemeinsamer Speicher sehr bald zu einem Engpaß.

Bei massiv-parallelen Rechnern haben daher die Rechenknoten lokale Speicher. Ein Netzwerk verbindet die Rechenknoten. Das Standardprogrammiermodell für solche Rechner ist Message-Passing. Hierbei wird der Parallelrechner als Verbund autonomer Rechenknoten angesehen, die über das Netz kommunizieren, um Daten auszutauschen.

Für einen Anwendungsprogrammierer ist es jedoch angenehmer, den Parallelrechner als einen Rechner mit einem globalen Adreßraum zu betrachten. *Compiler* für High Performance Fortran (HPF) [22] übersetzen Zugriffe auf globale Daten, jenachdem wo sich das Datum befindet, automatisch in lokale Zugriffe bzw. in Message-Passing. Ein weiterer Ansatz, dem Programmierer diese Betrachtungsweise zu ermöglichen, ist Distributed Shared Memory (DSM). Hier ist bereits auf Systemebene ein gemeinsamer virtueller Adreßraum vorhanden. Bei Shared Virtual Memory (SVM) [5], einer Implementierungsart von DSM, wird der gemeinsame virtuelle Adreßraum durch das Betriebssystem erzeugt.

Zur komfortablen Programmierung von SVM-Systemen wird eine parallele Programmiersprache mit einer Programmierumgebung benötigt. Diese Programmierumgebung wird am Zentralinstitut für Mathematik des Forschungszentrum Jülich im Rahmen des SVM-Fortran-Projektes [8, 14, 15] erstellt. SVM-Fortran (SVMF) ist eine Spracherweiterung von Fortran77 zur Programmierung massiv-paralleler Rechner mit SVM. SVMF ist auf der Intel Paragon, basierend auf dem Advanced Shared Virtual Memory (ASVM) [6, 7, 9], das SVM als Erweiterung des MACH3 Kerns implementiert, realisiert. ASVM verfügt über flexibles Monitoring, das über eine Software-Schnittstelle selektiert werden kann. Auf dieser Schnittstelle basiert der SVM-Fortran Application Monitor (SAM), der Laufzeitdaten erfaßt und diese in Trace Data Files (TDF) ablegt, die von den *Performance*-Analyse-Werkzeugen OPAL [11, 12, 13] und PARvis [26] weiterverarbeitet werden.

- **OPAL** : Optimizer and Locality Analyzer der SVMF-Entwicklungsumgebung ist ein quelltextorientiertes Analysewerkzeug, das den Anwender neben der Analyse auch bei der Instrumentierung der Anwendung unterstützt.
- **PARvis** ist ein X-basiertes Visualisierungswerkzeug zur Darstellung von Zustandswechseln. Visualisierungstechniken wie z.B. das Zooming erlauben dem Anwender, die vorliegenden Trace-Daten schnell und präzise auszuwerten.

Die Parallelisierung von Anwendungen unter Benutzung eines globalen Adreßrau-

mes, wie er in SVM-Architekturen vorhanden ist, verkürzt die Entwicklungszeit im Vergleich zur Message-Passing-Programmierung erheblich. Gründe hierfür sind, daß eine initiale Datenaufteilung und eine explizite Codierung der Kommunikation, wie es das Message-Passing-Programmiermodell erfordert, nicht notwendig sind. Andererseits ist durch den hohen Grad der Abstraktion der Einsatz von Analysewerkzeugen zur Programmoptimierung unabdingbar.

In der SVM-Fortran Entwicklungsumgebung sind zwei neue Konzepte verwirklicht, die die Portierung von Anwendungen auf Parallelrechner beschleunigen.

Inkrementelle Parallelisierung

Im Gegensatz zum Message-Passing-Programmiermodell kann eine Anwendung mit dem Shared-Memory-Programmiermodell inkrementell parallelisiert werden, da die Existenz eines globalen Adreßraumes eine initiale Datenaufteilung unnötig macht. Die bereits bei der Optimierung von sequentiellen Programmen bewährte Strategie, zunächst die Programmteile mit dem höchsten Rechenzeitanteil zu optimieren, kann somit auch zur Parallelisierung von Anwendungen mit SVM-Fortran benutzt werden. So ist bereits frühzeitig eine Aussage über den zu erwartenden Speedup möglich.

Inkrementelle Performance-Analyse

Bestehende *Performance*-Analyse-Werkzeuge basieren zumeist auf einer globalen Instrumentierung des Quelltextes. In einem Programmlauf werden alle verfügbaren Informationen für sämtliche Unterprogramme generiert. Dies führt in der Regel zu einer starken Laufzeitbeeinflussung und zu sehr großen Trace-Dateien. Bei der inkrementellen Parallelisierung von Anwendungen werden jedoch für die ersten Optimierungsschritte nur grobgranulare Informationen benötigt, die einen Überblick über das Laufzeitverhalten des gesamten Programms geben. In den folgenden Optimierungsschritten können zur Lösung eines *Performance*-Engpasses immer feingranularere *Performance*-Daten angefordert werden. Diese Daten brauchen jedoch nur für die zu optimierenden Programmteile – zum Teil nur für eine einzelne parallele Schleife – generiert werden. Die selektive Instrumentierung minimiert sowohl die Laufzeitbeeinflussung, als auch den Umfang der *Performance*-Daten, was eine anschließende Analyse stark vereinfacht und beschleunigt.

1.2 Ziele

Die bei der Portierung von wissenschaftlichen Anwendungen [18, 19, 20, 21] gewonnenen Erfahrungen zeigen, daß durch die inkrementelle Parallelisierung sehr schnell erste parallele Versionen der Anwendung verfügbar sind. Dies führt zu einer frühzeitigen Abschätzung der erreichbaren Geschwindigkeitssteigerung und Lokalisierung von Engpässen. Durch diese Informationen wird der weitere Optimierungsprozeß von Iteration zu Iteration geleitet und nach Amdahl's Gesetz ineffektive Optimierungen weitgehend vermieden, so daß der Optimierungsprozeß selbst optimiert wird.

Die inkrementelle Analyse beschleunigt die *Performance*-Analyse der einzelnen Programmversionen, obwohl im Allgemeinen für eine Programmversion mehrere Testläufe mit verschiedenen Meßkonfigurationen gestartet werden.

Dieses Vorgehen führt jedoch selbst bei kleinen Projekten schnell zu einer schwer überschaubaren Anzahl von untereinander abhängigen Dateien. Für jede Programmversion existieren zumeist mehrere Meßkonfigurationen, mit denen zum Teil mehrere Testläufe mit verschiedenen Prozessorzahlen ausgeführt werden. Zur Beurteilung einzelner Optimierungsschritte ist jedoch ein Rückgriff auf den Quelltext und die Meßwerte älterer Programmversionen notwendig, so daß diese nicht gelöscht bzw. überschrieben werden können.

Das Ziel dieser Arbeit ist, dem Entwickler die Verwaltung und den Zugriff auf den Quelltext und die Meßergebnisse der einzelnen Programmversionen durch Automatisierung dieser Aufgaben zu erleichtern.

Anhand eines iterativen Gleichungssystemlösers wird im zweiten, auf diese Einleitung folgenden Kapitel, das Vorgehen bei der Parallelisierung und Optimierung einer Anwendung mit SVM-Fortran aufgezeigt.

Die aus den Erfahrungen hergeleiteten Anforderungen legen den zu implementierenden Funktionsumfang fest. In einer anschließenden Untersuchung von UNIX-Werkzeugen zur Verwaltung von Quelltexten und von Datenbanken wird aufgezeigt, inwieweit diese Werkzeuge zur Realisierung des geforderten Funktionsumfangs geeignet sind. Mit der Vorstellung der verwendeten Konzepte endet die in Kapitel 3 behandelte Entwurfsphase.

Ausgewählte Aspekte der Realisierung und abschließende allgemeine Anmerkungen zur Kodierung, die die Einarbeitung Dritter in den Quelltext erleichtern, sind das Thema von Kapitel 4.

Auf die Anwendung der Versionsverwaltung und die hierdurch gegebene Vorgehensweise bei der Optimierung wird anhand des bereits in Kapitel 2 beschriebenen iterativen Gleichungssystemlösers in Kapitel 5 näher eingegangen. Dieses Kapitel ist so geschrieben, daß es nach Änderung des ersten Unterkapitels als eigenständige Einführung in die Anwendung verwendbar ist.

Ein Ausblick über mögliche Erweiterungen und weitere Einsatzgebiete für die Versionsverwaltung, sowie eine Zusammenfassung der Arbeit wird in Kapitel 6 gegeben.

Kapitel 2

Parallelisierung eines Gleichungssystemlösers

Dieses Kapitel gibt eine kurze Einführung in die SVM-Fortran Entwicklungsumgebung. Zunächst werden hierzu die Konzepte der Programmiersprache SVM-Fortran anhand ausgewählter Sprachelemente kurz umrissen. Die anschließende Vorstellung des Analysewerkzeugs OPAL ist etwas umfangreicher, da dieses Werkzeug das Vorgehen bei der Optimierung von SVM-Fortran Programmen prägt. Im dritten Unterkapitel werden schließlich die bei der Parallelisierung des iterativen Gleichungssystemlösers durchgeführten Optimierungsschritte detailliert beschrieben.

2.1 SVM-Fortran

SVM-Fortran [15, 16] ist eine Erweiterung von Fortran77 zur komfortablen Programmierung daten- und task-paralleler Programme auf SVM-Systemen. Die Entwicklung von SVM-Fortran wurde durch High Performance Fortran (HPF) [22], Fortran-S [25], Vienna Fortran [23] und KSR-Fortran [24] beeinflußt. Eine weitere Eigenschaft von SVMF ist die Unterstützung von geschachteltem Parallelismus.

Das SVM-System stellt einen gemeinsamen virtuellen Adreßraum zur Verfügung, den SVM-Fortran zur Allokation gemeinsamer (*shared*) Variablen nutzt. Alle Variablen sind standardmäßig *shared*. Sollen einzelne Variablen privat (*private*) sein, so sind diese explizit als *private* zu deklarieren. In SVM-Fortran werden *shared* und *private* Variablen nicht syntaktisch unterschieden und dürfen in arithmetischen Ausdrücken gemischt verwendet werden.

In SVM-Fortran wird ein Programm durch Verteilung der Arbeitslast auf die Prozessoren parallelisiert. Die vom Programmierer vorgegebene Arbeitsverteilung (*work distribution*) bestimmt das Zugriffsschema der einzelnen Prozessoren auf den gemeinsamen virtuellen Adreßraum und damit die Verteilung der Datenstrukturen auf die lokalen Speicher der Rechenknoten, die sich durch die Migration der Speicherseiten ergibt. Die Verteilung der Arbeitslast wird durch Direktiven spezifiziert. Neben Direktiven, die Prozessormengen definieren und die Verteilung der Arbeit beschreiben, sind auch Direktiven vorhanden, die Regionen (z.B. parallele Schleifen) einleiten. Durch Regionen wird das SVMF-Programm in parallel bzw. sequentiell auszuführende Abschnitte zerlegt. In SVM-Fortran können Regionen beliebig geschachtelt werden.

Die folgende Liste beschreibt eine Auswahl von SVMF-Direktiven und gibt einen Überblick der in SVMF vorhandenen Regionen.

PDO

Die `PDO`-Direktive spezifiziert parallele Schleifen. Als optionale Parameter sind die zu parallelisierenden Schleifen und die Verteilungsstrategie anzugeben. Wird keine Schleife explizit angegeben, bezieht sich die `PDO`-Direktive auf die direkt folgende Schleife. Die voreingestellte Arbeitsverteilung ist die Blockverteilung.

PSECTION, SECTION

Die in einer parallelen Sektion (`PSECTION`) enthaltenen Abschnitte (`SECTION`) werden parallel ausgeführt.

REPLICATED_REGION

Innerhalb einer *Replicated Region* wird das Programm von allen der Region zugeordneten Prozessoren redundant ausgeführt.

EXCLUSIVE_REGION

Der sequentielle Code innerhalb einer *Exclusive Region* wird nur von einem Prozessor ausgeführt. Paralleler Code, wie parallele Schleifen und parallele Sektionen werden parallel ausgeführt.

CRITICAL_SECTION

Der Code innerhalb einer *Critical Section* wird nur von einem Prozessor ausgeführt.

In SVM-Fortran wurde das *Template*-Konzept von HPF in erweiterter Form übernommen. Anders als in HPF, wo Feldelemente auf *Template*-Elemente abgebildet werden, werden in SVM-Fortran Schleifeniterationen und Tasks auf die *Template*-Elemente abgebildet. Der HPF-*Compiler* bestimmt aus der Datenverteilung durch Invertieren der Indizierungsabbildung die Arbeitsverteilung. Diese Berechnung ist zu zeitaufwendig, um sie zur Laufzeit auszuführen. Daher können *Templates* in HPF nur sehr eingeschränkt verwendet werden. In SVM-Fortran werden hingegen die Schleifeniterationen direkt auf *Template*-Elemente abgebildet, so daß diese Abbildung zur Laufzeit durchgeführt werden kann. Hierdurch können in SVM-Fortran irreguläre, dynamische Verteilungen implementiert werden, die eine effektive Parallelisierung unstrukturierter Anwendungen ermöglichen.

2.2 Das Analysewerkzeug OPAL

OPAL ist ein Werkzeug zur inkrementellen *Performance*-Analyse, das den Anwender bei der Optimierung der Datenlokalität von SVM-Fortran-Programmen unterstützt. Die inkrementelle *Performance*-Analyse stellt folgende Anforderungen an das Analyse-Werkzeug.

Selektive Instrumentierung
: Das Analyse-Werkzeug muß den Anwender bei der selektiven Instrumentierung des Programms unterstützen.

Quelltextbezug
: Das Analyse-Werkzeug muß die ermittelten *Performance*-Daten den untersuchten Quelltextbereichen zuordnen können.

Hierarchisches Datenformat
: Das Analyse-Werkzeug muß eine bestimmte Information in mehreren Granularitätsstufen anbieten.

Abbildung 2.1 zeigt die Integration von OPAL in die SVMF-Entwicklungsumgebung. Beim Einlesen der *Performance*-Daten (*trace data file*) verknüpft OPAL diese mit dem SVM-Fortran-Quelltext (*SVMF source code*) und der vom SVMF-*Compiler* generierten Informationsdatei (*compiler information file*) und stellt so einen Quelltextbezug der *Performance*-Daten her. So werden z.B. der Laufzeitanteil und die

Seitenfehlersummen als Annotationen rechts neben der jeweiligen Region dargestellt.

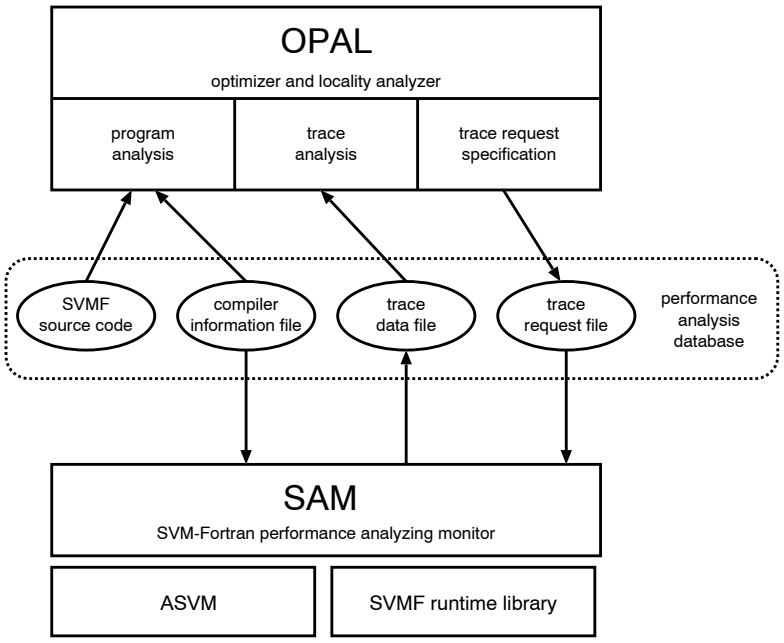


Abbildung 2.1: Integration von OPAL in die SVMF-Entwicklungsumgebung

Der *Performance* Analyse Monitor SAM [10] sammelt die Daten für die Analyse der *Performance* in OPAL. SAM ist eine Bibliothek, die die Aufzeichnung von Laufzeitinformation ausgeführter SVM-Fortran-Programme ermöglicht. Wird eine SVMF-Anwendung mit der Option `-tracing` übersetzt, werden im generierten Fortran77-Programm für jeden Regionenanfang und jedes Regionenende Einsprünge in SAM erzeugt. Das generierte Fortran77-Programm wird dann mit einem Fortran77-*Compiler* in eine Objektdatei übersetzt. Der *Loader* verbindet die Objektdatei mit der SAM-Bibliothek, der SVMF-Laufzeit-Bibliothek und der ASVM-Bibliothek zu einem ausführbaren Programm.

In OPAL werden mittels „Mausklick“ Regionen spezifiziert und die gewünschten *Performance*-Informationen über Menüs ausgewählt. Aus den durch den Anwender spezifizierten Trace-Anforderungen generiert OPAL das *Trace Request File*. Bei der Ausführung der so übersetzten Anwendung liest SAM während seiner Initialisierung das *Compiler Information File* und das *Trace Request File* und aktiviert die notwendigen Einsprungpunkte. Die während des Programmlaufs gesammelten

Daten schreibt jeder Rechenknoten in einen internen Puffer, der entweder bei anstehendem Überlauf, spätestens aber zum Programmende auf Festplatte geschrieben wird.

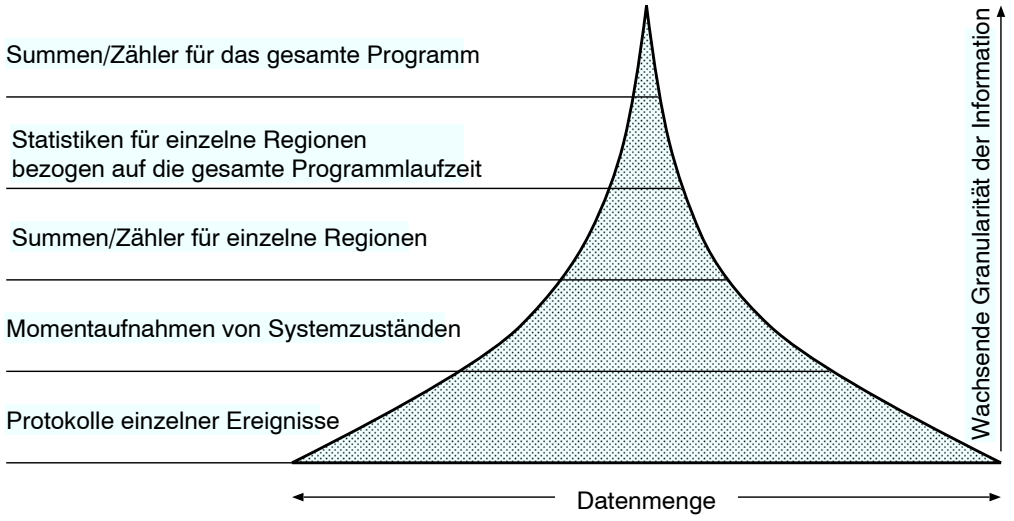


Abbildung 2.2: Hierarchie der Performance-Information

Die in Abbildung 2.2 dargestellte Hierarchie der *Performance*-Information wird durch das hierarchische Trace-Format von OPAL wiedergegeben, das gleichartige Informationen in verschiedener Granularität darstellen kann. Hierzu folgendes Beispiel: Die Informationen über Speicherseitenmigrationen sind wichtig für die Beurteilung der Datenlokalität einer SVM-Anwendung. Eine Speicherseite wird migriert, falls diese nicht im lokalen Speicher des anfordernden Prozessors vorhanden ist. Die Protokollierung jeder einzelnen Migration ergibt Daten mit der kleinstmöglichen Granularität, erzeugt jedoch eine gewaltige Menge an Daten. Diese Information ist notwendig um z.B. das Seitenflattern (*page trashing*) in einer parallelen Schleife zu analysieren. Ein guter Hinweis auf das Vorkommen von Seitenflattern ist eine hohe Anzahl von Seitenfehlern in einer Region. Daher bietet OPAL auch Zähler für Seitenfehler. Diese Zähler können auf die gesamte Programmlaufzeit, aber auch auf einzelne Regionen angewendet werden. Wird in einer Region aufgrund einer extrem hohen Anzahl von Seitenfehlern Seitenflattern vermutet, können in einem weiteren Programmlauf die Speicherseitenmigrationen in dieser Region ermittelt und analysiert werden.

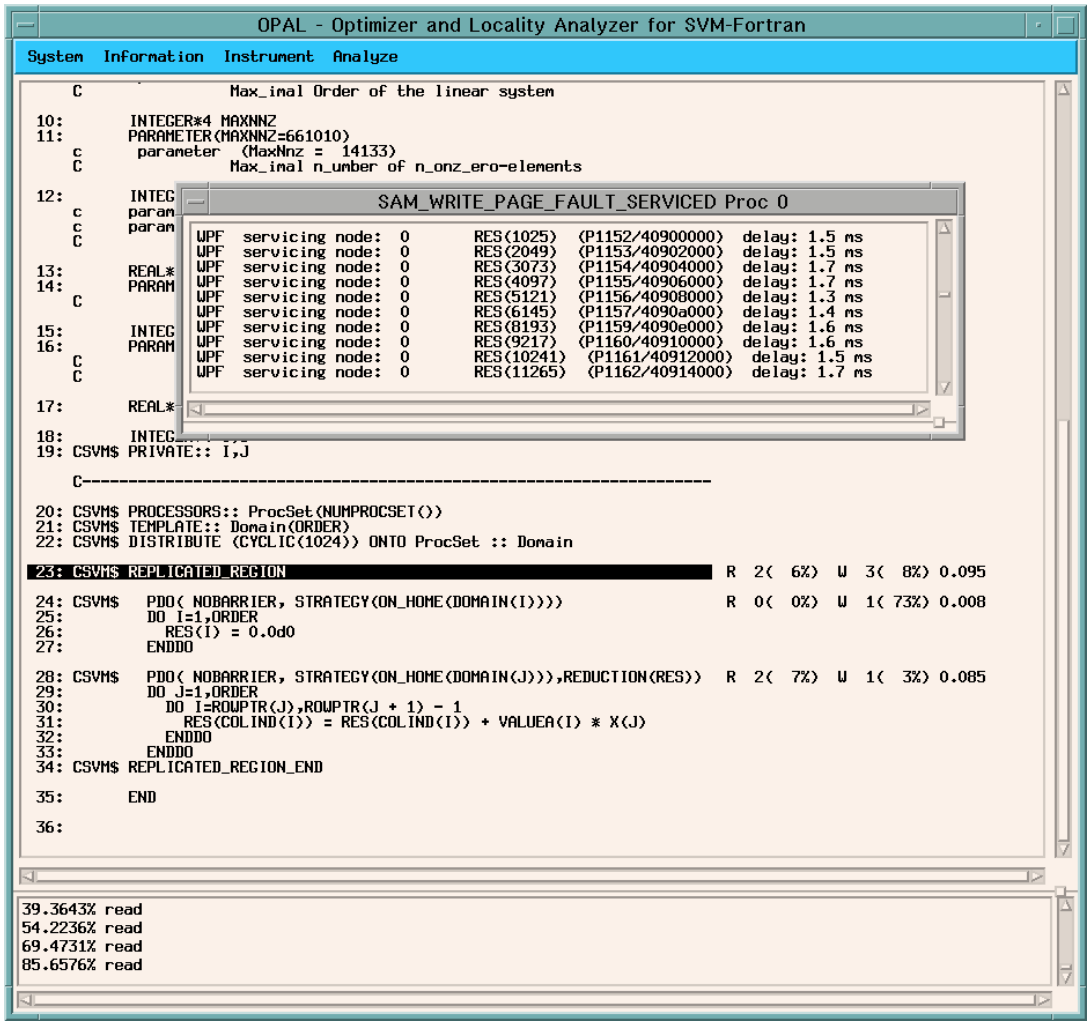


Abbildung 2.3: Die Bedienoberfläche von Opal

In Abbildung 2.3 ist die Bedienoberfläche von OPAL dargestellt. Die Annotationen rechts neben dem Quelltext geben Auskunft über die Anzahl der *Read Page Faults* und *Write Page Faults* und ihre Anteile an der Ausführungszeit der Region. Über ein *Pop Up*-Menü kann man sich anstelle dieser Information auch Synchronisationszeiten oder den gesamten *Overhead* anzeigen lassen.

Benötigt der Anwender mehr Informationen über eine einzelne Region, kann er diese per „Mausklick“ markieren und über Menüs die gewünschten *Trace Events* auswählen. Als Beispiel ist in Abbildung 2.3 im kleinen Fenster die Liste der aufge-

tretenen *Write Page Faults* für den Prozessor 0 dargestellt. Zur Identifikation einer migrierten Speicherseite wird neben dem ersten Feldelement, das auf dieser Speicherseite liegt, auch die Speicherseitennummer und ihre Adresse angegeben.

2.3 Parallelisierung einer QMR-Variante

Das Vorgehen bei der Parallelisierung und Optimierung einer Anwendung mit SVM-Fortran ist das Hauptthema dieses Unterkapitels. Dieses wird im folgenden anhand eines iterativen Lösers für Gleichungssysteme exemplarisch aufgezeigt. Hierzu wird die Aufgabe und die Realisierung dieses Lösers zunächst kurz beschrieben.

In vielen wissenschaftlichen und technischen Aufgabestellungen treten lineare Gleichungssysteme $Ax = b$ als Teilprobleme auf. Einen Sonderfall stellen Systeme mit dünnbesetzten, regulären Koeffizientenmatrizen A dar, wie sie in Finite-Elemente-Codes oder bei der Simulation elektrischer Schaltkreise auftreten.

Direkte Verfahren, wie beispielsweise die Gauß-Elimination lösen ein lineares Gleichungssystem, indem sie das Gleichungssystem durch sukzessives Ersetzen von Gleichungen durch Linearkombinationen von Gleichungen in ein äquivalentes Gleichungssystem überführen, das einfach gelöst werden kann. Die Zeitkomplexität dieser Verfahren ist $\mathcal{O}(N^3)$ und der Speicherbedarf beträgt $\mathcal{O}(N^2)$, wobei N die Anzahl der Unbekannten bezeichnet.

Bei iterativen Verfahren zum Lösen von linearen Gleichungssystemen [34, 35] wird ausgehend von einem vorgegebenen Startvektor $x^{(0)}$ in jeder Iteration eine Näherung der Lösung $x^{(i)}$ bestimmt, die bzgl. eines vorgegebenen Gütekriteriums verbessert werden soll. Ein mögliches Gütekriterium ist die euklidische Norm des Residuums der $i-$ten Iteration $\|r^{(i)}\|$ mit $r^{(i)} = b - Ax^{(i)}$. Es werden so lange bessere Näherungen berechnet, bis entweder die Näherung ausreichend genau ist oder die vorgegebene maximale Anzahl von Iterationen überschritten wird.

Die Methode der quasi-minimalen Residuen (QMR) nach Freund und Nachtigal [32] zur Lösung linearer Gleichungssysteme ist ein iteratives Verfahren, das für beliebige reguläre Koeffizientenmatrizen anwendbar ist. Dieses Verfahren kombiniert den klassischen, unsymmetrischen Lanczos-Algorithmus mit dem Ansatz der quasi-minimalen Residuen und enthält in jeder Iteration ein Matrix-Vektor-Produkt mit

der Koeffizientenmatrix und ein Matrix-Vektor-Produkt mit der transponierten Koeffizientenmatrix.

In der vorliegenden Anwendung ist der *Algorithmus 7.1* aus [33] implementiert. Dieser Algorithmus basiert auf einer gekoppelten Zwei-Term-Rekursion und besitzt drei globale Synchronisationspunkte. In jeder Iteration werden jeweils eine Matrix-Vektor-Multiplikation mit der Koeffizientenmatrix und der transponierten Koeffizientenmatrix, zwei Skalarprodukte und drei euklidische Normen berechnet.

2.3.1 Struktur des Codes

Abbildung 2.4 zeigt die Aufrufhierarchie der QMR-Implementation. Nach dem Einlesen des Datensatzes in `ReadSystem` wird das Gleichungssystem in `QMRFreund` gelöst. Zur Überprüfung der berechneten Lösung wird in `SolutionTest` die euklidische Norm des tatsächlichen Residuums berechnet.

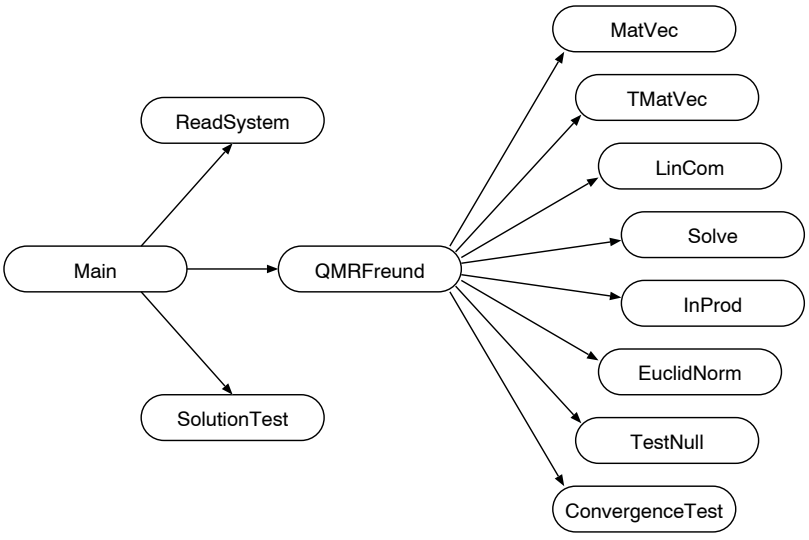


Abbildung 2.4: Aufrufhierarchie der QMR-Implementation

Dünnbesetzte Koeffizientenmatrizen zeichnen sich durch $t \ll N^2$ aus, wobei t die Anzahl der von Null verschiedenen Einträge der Koeffizientenmatrix bezeichnet. Daher verwendet die hier vorliegende Implementation des QMR-Algorithmus eine komprimierte Darstellung der Koeffizientenmatrix, in der nur die von Null verschiedenen Koeffizienten gespeichert werden. Neben einer Reduktion des Speicherverbrauchs

hat diese Darstellung, wie im folgenden gezeigt wird, den Vorteil, daß die Zeitkomplexität der Matrix-Vektor-Multiplikation $\mathcal{O}(t)$ statt $\mathcal{O}(N^2)$ beträgt.

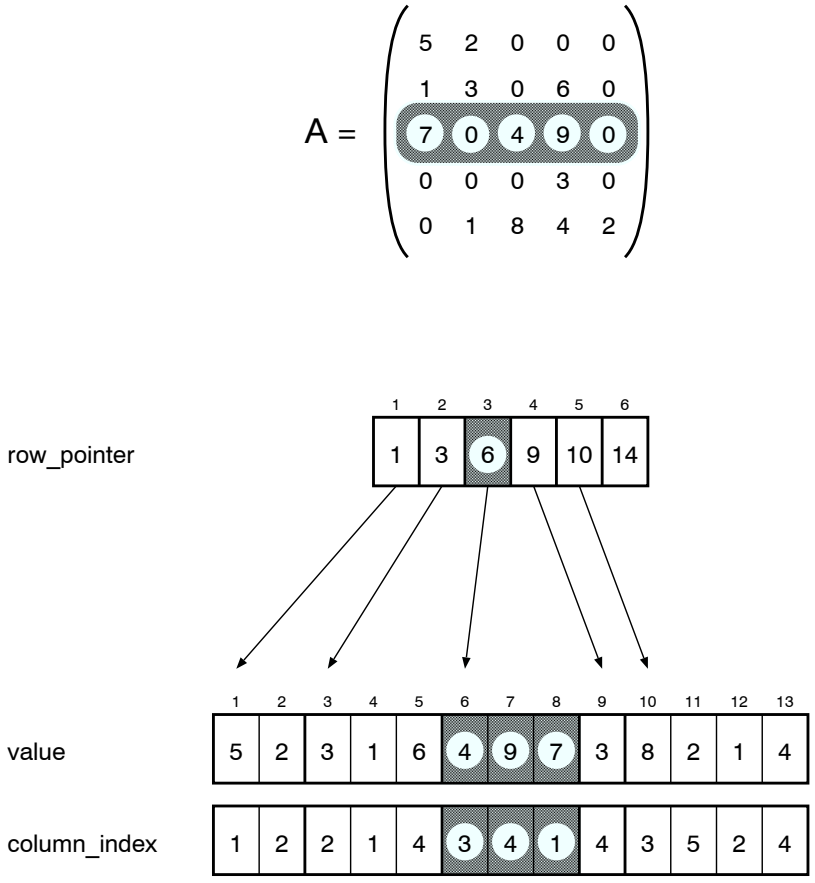


Abbildung 2.5: Compressed Row Storage (CRS)

Die *Compressed Row Storage* (CRS) genannte Darstellung ist in Abbildung 2.5 anhand einer Beispielmatrix veranschaulicht. In dieser Darstellung wird die Matrix in drei verschiedene eindimensionale Felder aufgespalten. Die Werte und Spaltenindizes der von Null verschiedenen Koeffizienten einer Zeile der Matrix sind in aufeinander folgenden Speicherplätzen der Felder `value` und `column_index` abgespeichert. Sie bilden somit einen Block innerhalb dieser Felder. In dem Feld `row_pointer` sind die Startindizes der Blöcke abgespeichert, wobei die Position innerhalb des Feldes die Zeilennummer der Matrix identifiziert. Dabei wird `row_pointer(N+1)=t+1` gesetzt, um eine Fallunterscheidung zu vermeiden. Die Reihenfolge der Elemente einer Zeile in `value` bzw. `column_index` ist beliebig.

So sind z.B. die von Null verschiedenen Koeffizienten der dritten Zeile der Koeffizientenmatrix ab dem sechsten Element des Feldes `value` abgelegt. Da die vierte Zeile mit dem neunten Element beginnt, hat die dritte Zeile $9 - 6 = 3$ von Null verschiedene Koeffizienten, deren Werte im dritten Block des Feldes `value`, also im sechsten, siebten und achten Element des Feldes `value` gespeichert sind. Die Spaltenindizes der Koeffizienten stehen in den entsprechenden Elementen des Feldes `column_index`. Somit ist $a_{3,3} = 4$, $a_{3,4} = 9$ und $a_{3,1} = 7$.

Die $i-$te Komponente des Matrix-Vektor-Produktes $y = \mathbf{A}x$ ist durch das innere Produkt des i-ten Zeilenvektors der Matrix $\mathbf{A}$ mit dem Vektor x gegeben. Für eine im CRS-Format abgespeicherte Matrix ergibt sich somit folgender Code.

```
do i = 1, N
  y(i) = 0
  do k = row_pointer(i), row_pointer(i+1)-1
    y(i) = y(i) + value(k) * x(column_index(k))
  end do
end do
```

Der Index i läuft über alle Zeilen der Koeffizientenmatrix und der Index k über die Elemente von `value` bzw. `column_index`, in denen die Werte und Spaltenindizes der Koeffizienten der i-ten Zeile stehen. Bei der Bildung der Skalarprodukte werden also nur die Produkte der von Null verschiedenen Koeffizienten mit den jeweils korrespondierenden Elementen des Vektors x aufsummiert. Somit beträgt die Zeitkomplexität hier nur $\mathcal{O}(t)$, im Gegensatz zu $\mathcal{O}(N^2)$ bei vollständiger Abspeicherung der Matrix.

Bei dem Matrix-Vektor-Produkt mit der transponierten Koeffizientenmatrix ist die i-te Komponente des Produktes $y = \mathbf{A}^T x$ hingegen durch das innere Produkt aus i-ten Spaltenvektor der Matrix $\mathbf{A}$ mit dem Vektor x gegeben. Dies ergibt einen spaltenweisen Zugriff auf die Matrix $\mathbf{A}$, der in der verwendeten Datenstruktur sehr zeitaufwendig ist. Durch die indirekte Adressierung des Ergebnisvektors y erhält man jedoch auch hier einen zeilenweisen Zugriff auf die Matrix $\mathbf{A}$. Im folgenden Fortran-Quellcode wird dies angewendet.

```
do i = 1, N
  y(i) = 0
end do

do k = 1, N
  do i = row_pointer(k), row_pointer(k+1)-1
    y(column_index(i)) = y(column_index(i)) + value(i)*x(k)
  end do
end do
```

Zum *Performance*-Vergleich wurde ein System mit 13860 Gleichungen und 661010 von Null verschiedenen Koeffizienten verwendet. Dieses System beschreibt den Pressvorgang einer Motorhaube mit Hilfe der Finite-Elemente-Methode. Um die Rechenzeit einzuschränken, wurden in den Testläufen nur jeweils 100 Iterationen gerechnet.

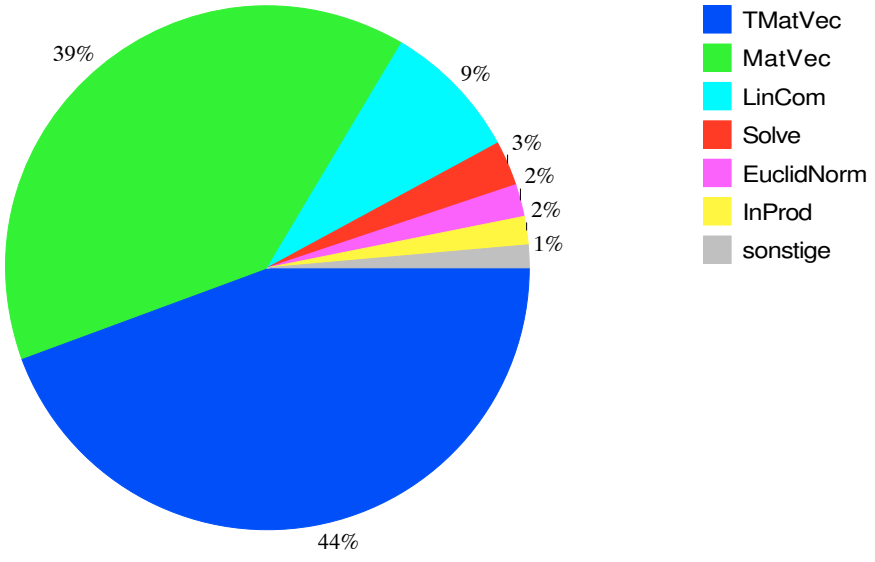


Abbildung 2.6: Profil des sequentiellen Codes

Die Verteilung der Rechenzeit innerhalb der sequentiellen Version von `QMRFreund` ist in Abbildung 2.6 dargestellt. Für die 100 Iterationen werden 49.2s benötigt. Die Matrix-Vektor-Produkte mit der Systemmatrix und ihrer Transponierten sind dominierend. Zusammen benötigen diese mehr als 80% der Rechenzeit.

Ein kleineres System mit 2205 Gleichungen und 14133 von Null verschiedenen Ko-

effizienten aus der Harwell-Boeing Sparse Matrix Collection diente zur Überprüfung des Konvergenzverhaltens der parallelen Programmversionen. Auch dieses System ist nicht synthetisch, es stammt aus dem Bereich der Erdölförderung.

2.3.2 Parallelisierungsstrategien

Neben den parallelen Schleifen zur Abbildung von Daten-Parallelismus, wird mit den parallelen Sektionen auch funktionaler Parallelismus unterstützt. Der QMR-Algorithmus besitzt neben dem von den Vektor-Vektor- und Matrix-Vektor-Operationen stammenden Daten-Parallelismus auch funktionalen Parallelismus. So können z.B., wie in Abbildung 2.7 dargestellt, die beiden Matrix-Vektor-Multiplikationen `MatVec` und `TMatVec` einer Iteration parallel ausgeführt werden. Das Unterprogramm `InProd` berechnet das innere Produkt zweier Vektoren und bildet in dem gezeigten Abschnitt Barrieren. `LinCom` berechnet die Linearkombination zweier Vektoren.

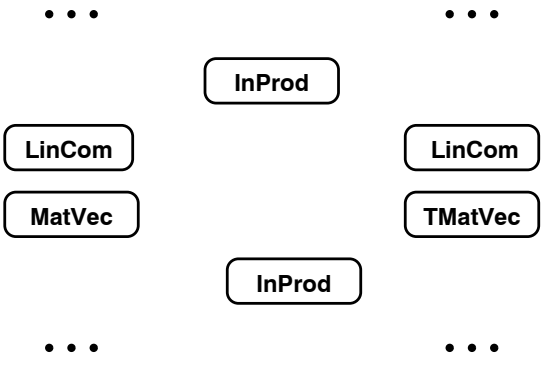


Abbildung 2.7: Ausschnitt aus der Iteration

Beim Datenparallelismus wird der *Speedup* durch die Anzahl der parallel zu verarbeitenden Datenelemente bestimmt. Bei den in diesem Algorithmus vorkommenden Vektor-Vektor- und Vektor-Matrix-Operationen legt somit die Dimension der Vektoren bzw. der Matrizen den maximal erreichbaren *Speedup* fest. Bei reinem funktionalen Parallelismus ist der *Speedup* hingegen durch die Anzahl der parallel laufenden *Tasks* begrenzt. Durch Hinzufügen des funktionalen Parallelismus zum Datenparallelismus kann der *Speedup* somit maximal verdoppelt werden.

2.3.3 Initiale Parallelisierung

In der verwendeten Implementation des QMR-Algorithmus sind alle Rechenschritte gekoppelt. Daher sind bereits in der initialen Parallelisierung neben den beiden Matrix-Vektor-Produkten auch sämtliche Vektor-Vektor-Operationen durch die Angabe der Direktive `CSVM$ PDO` parallelisiert.

In SVM-Fortran sind verteilt berechnete Reduktionen durch die Option `reduction` in der `PDO`-Direktive zu spezifizieren. Ist die Reduktionsvariable *shared*, werden lokale Kopien angelegt, auf denen jeder Prozessor eine Teilreduktion ausführt. Hierdurch wird die in Abbildung 2.8 geschilderte Situation umgangen.

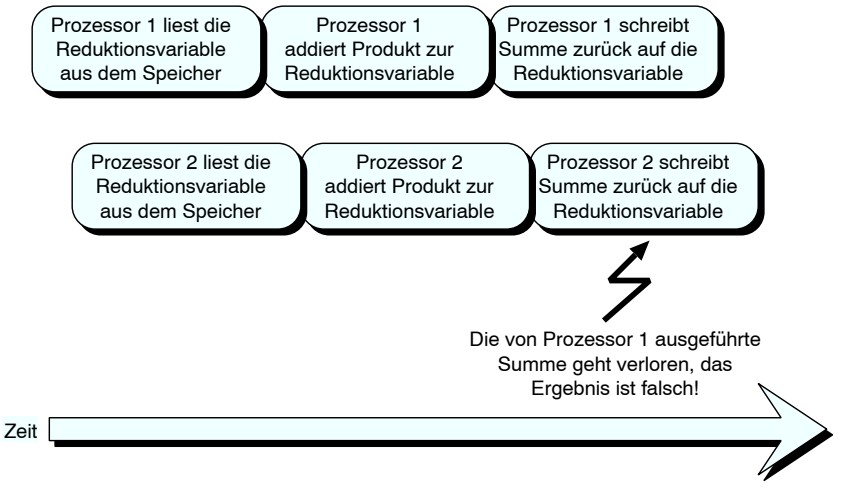


Abbildung 2.8: Verteilt berechnete Reduktion

In den Funktionen `EuclidNorm` und `Inprod` wird das innere Produkt zweier Vektoren gebildet. Diese Skalar-Reduktionen werden verteilt berechnet und sind somit zu spezifizieren. Innere Produkte treten auch bei den beiden Matrix-Vektor-Produkten auf. Bei dem Produkt mit der Systemmatrix werden die inneren Produkte auf die Rechenknoten verteilt, so daß jedes innere Produkt vollständig sequentiell auf einem Rechenknoten berechnet wird. Bei dem in Abbildung 2.9 aufgeführten Produkt mit der transponierten Systemmatrix werden jedoch die einzelnen inneren Produkte parallel berechnet. Somit tritt auch hier eine verteilt berechnete Reduktion auf, die zu spezifizieren ist.

```
CSVM$ PDO(STRATEGY(ON_HOME(Domain(j)))
      do i = 1, N
        y(i) = 0
      end do

CSVM$ PDO(STRATEGY(ON_HOME(Domain(j)),REDUCTION(y)))
      do k = 1, N
        do i = row_pointer(k), row_pointer(k+1)-1
          y(column_index(i)) = y(column_index(i)) + value(i)*x(k)
        end do
      end do
```

Abbildung 2.9: Parallele Berechnung des Produktes mit der Transponierten

2.3.4 Explizite Verteilung der Iterationen

Der Hauptgrund für die hohen Ausführungszeiten der initialen parallelen Version (vgl. Abbildung 2.10) ist *False Sharing*. Ein Beispiel ist die folgende parallele Schleife zur Bildung von Linearkombinationen.

```
CSVM$ PDO
      do i = 1, N
        r(i) = alpha * x(i) + beta * y(i)
      end do
```

Hier kann es beim Zugriff auf das Feld `r` vorkommen, daß benachbarte Prozessoren gleichzeitig auf eine Speicherseite schreiben und diese Speicherseite mehrfach zwischen diesen beiden Prozessoren hin und her wandert.

Anzahl Proz.	Seitenfehler pro Iteration		Sync-Overhead	Zeit für eine Iteration
4	87	31%	14%	810ms
8	74	31%	22%	894ms
16	65	18%	15%	2134ms

Abbildung 2.10: Performance der initialen Version

Nach Ausrichten der Felder auf Seitengrenzen und block-zyklischer Verteilung der Iterationen derart, daß ein Block von Iterationen genau eine Speicherseite bearbei-

tet, ist das *False Sharing* von Vektoren beseitigt.

```
CSVM$ PROCESSORS:: ProcSet(NUMPROC())
CSVM$ TEMPLATE:: Domain(ORDER)
CSVM$ DISTRIBUTE (CYCLIC(1024)) ONTO ProcSet :: Domain
```

Diese Wahl der Arbeitsverteilung bestimmt die Anzahl der zu verwendenden Rechenknoten. Das für den *Performance*-Vergleich verwendete Gleichungssystem hat die Dimension 13860. Somit sind $\lceil 13860/1024 \rceil = 14$ Blöcke zu verteilen. Soll z.B. jeder Rechenknoten drei Blöcke erhalten, sind $\lceil 14/3 \rceil = 5$ Rechenknoten notwendig.

Die nach expliziter Verteilung der Iterationen ermittelten *Performance*-Daten sind in Abbildung 2.11 tabellarisch aufgeführt. Gegenüber der initialen Version ist die Anzahl der Seitenfehler deutlich reduziert und die *Performance* entsprechend verbessert. Störend ist jedoch der hohe Synchronisations-*Overhead*, der im folgenden Schritt reduziert wird.

Anzahl Proz.	Seitenfehler pro Iteration		Sync-Overhead	Zeit für eine Iteration
3	62	19%	15%	728ms
4	52	17%	19%	742ms
5	43	18%	18%	763ms
7	37	19%	20%	632ms
14	24	19%	26%	686ms

Abbildung 2.11: Performance nach block-zyklischer Arbeitsverteilung

2.3.5 Elimination unnötiger Barrieren und Privatisierung von Skalaren

Es gibt mehrere weitere *Overheads*, die den Speedup begrenzen. So wird z.B. standardmäßig am Eingang einer Region eine Barriere eingesetzt. Das Einfügen dieser Barrieren kann mit der `nobarrier`-Option abgeschaltet werden. Die drei im Algorithmus enthaltenen globalen Synchronisationspunkte fallen mit Reduktionen zusammen, so daß sämtliche Barrieren an den Regionseingängen unnötig sind.

Die sequentiellen Teile des Programms werden vom Master-Prozessor ausgeführt. Die anderen Prozessoren müssen jedoch dem Kontrollfluß folgen. Dieses wird durch Nachrichten, die der Master-Prozessor an die anderen Prozessoren verschickt, erreicht. Dieser *Overhead* wird durch replizierte Ausführung des Codes innerhalb von *Replicated Regions* vermieden. Damit die Semantik des Codes hierbei nicht geändert wird, müssen Variablen privatisiert werden. Desweiteren sind die Prozeduraufrufe, die parallele Konstrukte enthalten, mit der `coordinated_call`-Direktive zu annotieren.

```
CSVM$ PRIVATE:: j, norm
CSVM$ REPLICATED_REGION(NOBARRIER)
      norm = 0.0d0
CSVM$ PDO(NOBARRIER,STRATEGY(ON_HOME(Domain(j))),REDUCTION(norm))
      DO j = 1, order
        norm = norm + x(j) * x(j)
      END DO
      norm = sqrt(norm)
CSVM$ REPLICATED_REGION_END
```

Reduktionsoperationen auf *Shared Variables* werden in SVM-Fortran verteilt ausgeführt, dennoch wird das Ergebnis dem *Master*-Prozessor zugeordnet. Spätere Zugriffe von anderen Prozessoren auf dieses Ergebnis führen zu *Page Faults*. Die Privatisierung von Skalaren, wie im obigen Beispiel aufgezeigt, eliminiert diese *Page Faults*, da nach der Reduktion jeder Prozessor eine private Kopie des Ergebnisses enthält.

Anzahl Proz.	Seitenfehler pro Iteration		Sync-Overhead	Zeit für eine Iteration
3	30	14%	5%	480ms
4	24	13%	6%	433ms
5	21	13%	7%	400ms
7	16	14%	10%	356ms
14	10	14%	10%	352ms

Abbildung 2.12: Performance nach Eliminierung unnötiger Barrieren

Abbildung 2.12 zeigt die erreichten Verbesserungen. Die Elimination von Barrieren resultiert in einer Reduzierung des Synchronisations-*Overhead*. Durch die Privati-

sierung wurde die Anzahl der Seitenfehler nochmals reduziert.

2.3.6 Weitere Optimierungen

Eine Optimierung des Cache-Zugriffs, mit der beim AVL FIRE Benchmark [18] eine große *Performance*-Steigerung erreicht wird, bringt bei diesem Code keinen Gewinn, da hier keine Wiederverwendung von Daten vorhanden ist.

Bei jedem Unterprogrammaufruf werden die lokalen Variablen angelegt und am Ende des Unterprogramms wieder gelöscht. Durch manuelles Inlining wurde dieser *Call Overhead* in der endgültigen Programmversion eliminiert.

Anzahl Proz.	Seitenfehler pro Iteration		Sync-Overhead	Zeit für eine Iteration
3	19	9%	$< 1\%$	423ms
4	16	10%	$< 1\%$	375ms
5	15	11%	$< 1\%$	335ms
7	12	12%	$< 1\%$	286ms
14	7	13%	$< 1\%$	249ms

Abbildung 2.13: Performance nach manuellem Inlining

Die Meßwerte in Abbildung 2.13 zeigen, daß auch diese Programmversion unzureichend skaliert. Auch in den parallelen Programmversionen dominieren die beiden Matrix-Vektor-Produkte mit der Systemmatrix und ihrer Transponierten. Laufzeitmessungen der Unterprogramme `MatVec` und `TMatVec` ergeben, daß das Produkt mit der Systemmatrix nahezu perfekt skaliert, das Produkt mit der Transponierten (siehe Abbildung 2.14) hingegen kaum durch eine höhere Prozessorzahl gewinnt.

Nähere Untersuchungen waren jedoch zunächst nicht möglich. Vermutet wurde, daß die vorhandenen Reduktions- oder Dispatch-Zeiten eine Skalierung verhinderten. Also wurden OPAL, SAM und der SVM-Fortran-*Compiler* dahingehend erweitert, daß eine Analyse des *Reduction-* und *Dispatch-Overheads* möglich wurde.

Die Tabelle in Abbildung 2.14 gibt die Laufzeiten für das Matrix-Vektor-Produkt mit der Systemmatrix (`MatVec`), sowie die Laufzeiten und die Reduktionszeiten des Produktes mit der Transponierten für 7 und 14 Prozessor-Läufe wieder. Beide Produkte

benötigen in etwa die gleiche „Nettorechenzeit“, d.h. nur die mit der Prozessorzahl steigende Reduktionszeit verhindert das Skalieren des Produktes mit der Transponierten.

Anzahl Proz.	MatVec gesamte Laufzeit	TMatVec gesamte Laufzeit	TMatVec Reduktions-zeit
7	60ms	103ms	41ms
14	32ms	86ms	52ms

Abbildung 2.14: Reduktionszeiten im Unterprogramm `TMatVec`

In Abbildung 2.15 ist der Quelltext der Matrix-Vektor-Multiplikation mit der transponierten Matrix gezeigt.

```
CSVM$ PRIVATE:: i, j
CSVM$ SHARED:: res, x, valueA, rowptr, colind

CSVM$ PROCESSORS:: ProcSet(NUMPROCSET())
CSVM$ TEMPLATE:: Domain(ORDER)
CSVM$ DISTRIBUTE (CYCLIC(1024)) ONTO ProcSet :: Domain

CSVM$ REPLICATED_REGION(NOBARRIER)
CSVM$ PDO(NOBARRIER,STRATEGY(ON_HOME(Domain(i))))
      DO i = 1, order
        res(i) = 0.0d0
      END DO
CSVM$ PDO(NOBARRIER,STRATEGY(ON_HOME(Domain(j))),REDUCTION(res))
      DO j = 1, order
        DO i = rowptr(j), rowptr(j+1)-1
          res(colind(i)) = res(colind(i)) + valueA(i) * x(j)
        END DO
      END DO
CSVM$ REPLICATED_REGION_END
```

Abbildung 2.15: Ursprünglicher Quelltext von `TMatVec`

Der Ergebnis-Vektor von `TMatVec` ist *shared*, für die Berechnung der Reduktion werden zunächst private Kopien des Ergebnis-Vektors angelegt. Jeder Prozessor bildet innerhalb der zweifach geschachtelten Schleife eine Teilsumme auf seiner privaten Kopie. Anschließend wird mit der Bibliotheksfunktion `gdsum` [31] die globale Summe derart gebildet, daß diese nach Abschluß der Summation in allen privaten Kopien des Ergebnis-Vektors vorliegt. Der Master-Prozessor addiert danach seine lokale Kopie auf den Ergebnis-Vektor. Die Elemente des Ergebnis-Vektors werden vor der Reduktion in einer parallelen Schleife mit Null initialisiert und damit über die Rechenknoten verteilt. Bei der Addition der lokalen Kopie auf den Ergebnis-Vektor wird zunächst ein Element des Ergebnis-Vektors gelesen. Ist dieses nicht im Speicher des *Master*-Prozessors, resultiert hieraus ein *Read Page Fault*. Danach wird dieses Element mit dem korrespondierenden Element der lokalen Kopie addiert und danach zurückgeschrieben. Ist das Element zuvor aus dem Speicher eines anderen Rechenknotens importiert worden, generiert der schreibende Zugriff einen *Write Page Fault*. Diese Vorgehensweise ist in Abbildung 2.16 illustriert.

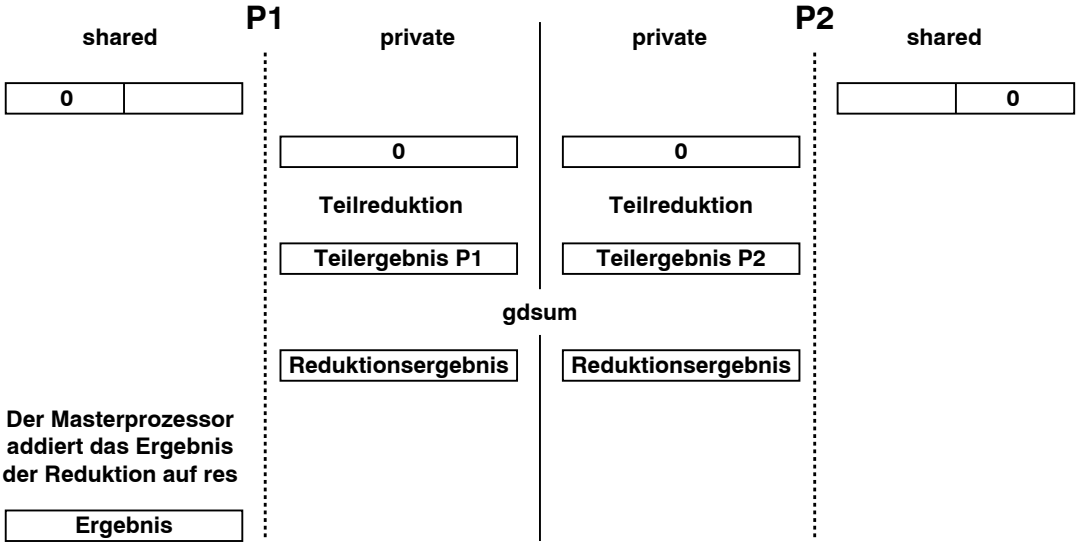


Abbildung 2.16: Datenfluß in der Vektorreduktion

In Abbildung 2.17 sind die bei dieser Variante gemessenen *Performance*-Daten zusammengefaßt. Die starke Lastinbalance zwischen dem *Master*-Prozessor und den restlichen Prozessoren verhindert ein Skalieren von `TMatVec`.

Anzahl Proz.	Master-Prozessor			restliche Prozessoren		
	Seitenfehler		Reduktions-	Seitenfehler		Reduktions-
	lesend	schreibend	zeit	lesend	schreibend	zeit
3	9	9	80ms	1	0	36ms
4	10	10	82ms	1	0	52ms
5	11	11	101ms	1	0	55ms
7	12	12	108ms	1	0	45ms
14	13	13	127ms	1	0	68ms

Abbildung 2.17: Seitenfehler im Unterprogramm `TMatVec`

Abbildung 2.18 illustriert den Datenfluß des in Abbildung 2.19 gezeigten Quelltextes. Hier ist der Engpaß durch explizites Anlegen privater Hilfsvektoren `r` und paralleles Kopieren der Ergebnisse der Teilreduktionen auf den globalen Ergebnisvektor beseitigt ist.

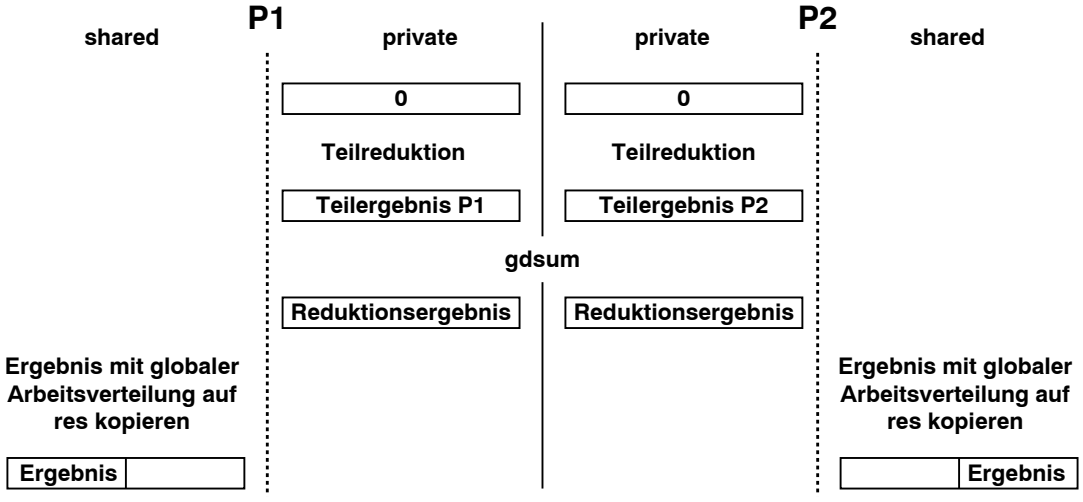


Abbildung 2.18: Datenfluß in der optimierten Vektorreduktion

In einer sequentiellen Schleife werden die privaten Hilfsvektoren mit 0 vorbelegt. Diese Schleife ist in eine repliziert ausgeführte Region (*replicated region*) eingebettet, so daß sie von allen Prozessoren parallel augeführt wird. Wie vorher werden die

```
CSVM$ PRIVATE:: r, i, j
CSVM$ SHARED:: res, x, valueA, rowptr, colind

CSVM$ PROCESSORS:: ProcSet(NUMPROCSET())
CSVM$ TEMPLATE:: Domain(ORDER)
CSVM$ DISTRIBUTE (CYCLIC(1024)) ONTO ProcSet :: Domain

CSVM$ REPLICATED_REGION(NOBARRIER)
      DO i = 1, order
        r(i) = 0.0d0
      END DO
CSVM$ PDO(NOBARRIER,STRATEGY(ON_HOME(Domain(j))),REDUCTION(r))
      DO j = 1, order
        DO i = rowptr(j), rowptr(j+1)-1
          r(colind(i)) = r(colind(i)) + valueA(i) * x(j)
        END DO
      END DO
CSVM$ PDO(NOBARRIER,STRATEGY(ON_HOME(Domain(i))))
      DO i = 1, order
        res(i) = r(i)
      END DO
CSVM$ REPLICATED_REGION_END
```

Abbildung 2.19: Quelltext der optimierten Version von `TMatVec`

Teilreduktionen lokal ausgeführt und mit der Bibliotheksfunktion `gdsum` [31] kombiniert. Im Gegensatz zum ursprünglichen Vorgehen wird im Anschluß daran das Reduktionsergebnis von allen Prozessoren parallel auf den globalen Ergebnisvektor kopiert. Hierbei wird die globale Arbeitsverteilung verwendet, so daß keine Seitenfehler entstehen. Dadurch skaliert das Programm deutlich besser.

Für die Untersuchung und Optimierung der Vektorreduktion in `TMatVec` wurde die letzte Version ohne *Inlining* verwendet. Der Ergebnis-Vektor `res` des Matrix-Vektor-Produktes mit der transponierten Systemmatrix wird im weiteren Verlauf der Iteration zur Bildung einer Linearkombination gelesen und kann somit privatisiert werden. Dies wurde beim Nachtragen der Optimierung in die Version mit *Inlining* ausgenutzt. Damit entfällt zusätzlich das Kopieren der privaten Kopien auf einen globalen Vektor und die *Performance* wird nochmals verbessert.

In der Tabelle in Abbildung 2.20 sind die *Performance*-Zahlen der endgültigen parallelen Version aufgeführt. Die Anzahl der Seitenfehler ist weniger als halb so groß wie bei der vorigen Version (siehe Abbildung 2.13). Die mit 14 Prozessoren benötigte Zeit für eine Iteration sank von 249ms auf 163ms.

Anzahl Proz.	Seitenfehler pro Iteration		Sync-Overhead	Zeit für eine Iteration
3	7	5%	$< 1\%$	354ms
4	6	5%	$< 1\%$	304ms
5	6	6%	$< 1\%$	262ms
7	5	9%	$< 1\%$	201ms
14	3	11%	$< 1\%$	163ms

Abbildung 2.20: Performance nach allen Optimierungen

Kapitel 3

Eine Versionsverwaltung für OPAL

Dieses Kapitel beschreibt den Entwurf der Versionsverwaltung. Aus den durch die Anwendung von OPAL gewonnenen Erfahrungen werden Anforderungen an die Versionsverwaltung aufgestellt. Basierend auf diesen Anforderungen wird der notwendige Funktionsumfang festgelegt. Hierauf folgt ein Überblick über verfügbare Werkzeuge, die für die Versionsverwaltung eingesetzt werden können. Abschließend werden die Konzepte der Versionsverwaltung dargestellt.

3.1 Ist-Zustand und Anforderungen

Meßkonfigurationen

OPAL erwartet bis auf die *Trace Data Files* alle Dateien in dem Verzeichnis, aus dem es gestartet wurde. Die Namen des *Trace Request File* und der vom SVM-Fortran-*Compiler* erzeugten Hilfsdateien, die zusammen mit dem Quelltext die Meßkonfiguration bilden, sind fest in OPAL eincodiert, so daß nur die momentan verwendete Meßkonfiguration gespeichert ist. Bei den Meßergebnissen, die in den *Trace Data Files* gespeichert sind, kann der Anwender hingegen den Präfix der Dateinamen frei wählen. Somit ist eine Speicherung von mehreren Messungen möglich.

Zuordnung einer Messung zum Quelltext

Eine Zuordnung eines *Trace Data File* und der Hilfsdateien zu einer Quelltextversion ist nicht gegeben. Dies kann zu Fehlern führen, wenn ein *Trace Data File* mit einem

zwischenzeitlich veränderten Quelltext kombiniert wird. Ebenso ist aufgrund der ausschließlichen Verfügbarkeit des aktuellen *Trace Data File* ein schnelles Ermitteln der einer Messung zugrundeliegenden Instrumentierung nicht möglich, obwohl dies bei einer erneuten Überarbeitung eines länger ruhenden Projektes oder zur Einarbeitung in ein Projekt notwendig ist.

Versionsverwaltung

Bei der Parallelisierung von Anwendungen wird zum Teil auf ältere Quelltextversionen zurückgegriffen, z.B. um eine alternative Parallelisierungsstrategie zu verfolgen. Hierdurch erhält eine Version mehrere alternative Nachfolger, und es entsteht ein Versionsbaum. Für die Beurteilung einzelner Optimierungsschritte ist ein Zugriff auf alle Quelltextversionen notwendig.

Bei der inkrementellen Parallelisierung und Optimierung von Anwendungen mit SVM-Fortran entstehen schon bei kleineren Projekten eine schwer überschaubare Anzahl voneinander abhängiger Dateien, die bislang vom Anwender selbst zu verwalten sind.

Um die Verwaltung der voneinander abhängigen Dateien zu automatisieren, muß der Quelltext zusammen mit den zugehörigen *Trace Request Files*, *Trace Data Files* und Hilfsdateien auf eine Datenstruktur abgebildet werden, die neben den reinen Daten auch die Verknüpfungen zwischen den einzelnen Dateien wiedergibt.

Unterstützung des gesamten Optimierungsprozesses

Die Durchführung des Optimierungsprozesses ist für nicht erfahrene Anwender umständlich. Neben der OPAL-Umgebung wird jeweils eine UNIX-Konsole auf der *Workstation*, auf der OPAL läuft, zum Aufbereiten der *Trace Data Files* und auf dem Parallelrechner zum Übersetzen und Starten der Programme benötigt. Auch hier können Fehler entstehen, wenn der Entwickler vergißt, den geänderten Quelltext vor den Testläufen zu übersetzen.

Der gesamte Zyklus des Optimierungsprozesses ist zu automatisieren. Hierzu ist, neben dem Übersetzen und Starten einer Anwendung aus OPAL, auch eine Möglichkeit notwendig, den Quelltext ohne Verlassen der OPAL -Oberfläche zu ändern. Vor dem Starten der Anwendung ist zu überprüfen, ob das *Executable* aktuell ist. Falls

notwendig ist der Quelltext neu zu übersetzen.

Projektunterstützung

Projekte werden von Zeit zu Zeit wieder neu aufgegriffen, wenn z.B. in anderen Projekten zwischenzeitlich gewonnene Erkenntnisse weitere *Performance*-Steigerungen ermöglichen. Weiterhin werden teilweise mehrere Projekte parallel bearbeitet.

Die Versionsverwaltung muß daher Funktionen zum Anlegen, Pflegen und Löschen von Projekten bereitstellen.

Dokumentation

Beim erneuten Aufgreifen eines seit längerer Zeit ruhenden Projektes ist zum Teil eine lange Einarbeitungszeit aufgrund fehlender Dokumentation der bereits durchgeführten Optimierungen und Messungen notwendig.

Daher ist eine den Optimierungsprozeß begleitende Dokumentation von der Versionsverwaltung zu unterstützen. Hierbei sind für Projekte, Quelltextversionen und Meßkonfigurationen getrennte Kommentare vorzusehen. Die Benutzerführung muß derart gestaltet sein, daß die Funktionen zur Dokumentation genutzt werden.

Speicherplatzrestriktionen

Obwohl durch die inkrementelle Analyse die Größe der *Trace Data Files* minimiert wird, kann ein einzelner Testlauf durchaus mehrere Megabyte an *Performance*-Daten herausschreiben. Auch die Größe eines *Executable* ist etwa ein Megabyte. Selbst bei kleineren Projekten können schnell einige hundert Megabyte an *Performance*-Daten generiert werden. Die *Performance*-Daten, sowie die *Executables* sind reproduzierbare Daten und können somit gelöscht werden, um bestehende Speicherplatzrestriktionen einzuhalten. Zudem sind Eingabedaten zum Teil sehr groß, so daß diese nicht mehrfach gespeichert sein dürfen. Somit muß eine zentrale Kopie der Eingabedaten von allen Programmversionen lesbar sein.

Somit sind Funktionen zu implementieren, die ein selektives Löschen dieser reproduzierbaren Dateien ermöglichen.

Portabilität

Die Versionsverwaltung ist auch in Analysewerkzeuge für *Message Passing* einzubinden. Daneben ist die Versionsverwaltung als eigenständiges Werkzeug denkbar.

Portablität heißt in diesem Fall nicht ausschließlich, daß der Quelltext der Versionsverwaltung auf verschiedenen UNIX-Plattformen übersetzbar und lauffähig ist. Vielmehr ist hier Portabilität im Sinne von Wiederverwendbarkeit des Quelltextes ein wichtiger Punkt und bereits beim Entwurf zu berücksichtigen.

Aufbau von SVM-Fortran Quelltexten

Der SVM-Fortran *Compiler* ist derart implementiert, daß eine getrennte Übersetzung einzelner Programmteile nicht möglich ist. OPAL benötigt daher als für SVM-Fortran entwickeltes Werkzeug den Quelltext in einer Datei, in der *Includes* eingebunden sein dürfen. Somit ist nicht von mehreren Quelltextdateien auszugehen.

Eine spätere Erweiterung auf die Verwaltung mehrere Quelltextdateien, die bei der Nutzung der Versionsverwaltung für *Message-Passing* notwendig ist, muß einfach durchzuführen sein.

Mehrbenutzerunterstützung als optionale Anforderung

Die Unterstützung von Teamarbeit ist eine optionale Anforderung. Dennoch ist diese bereits beim Entwurf zu berücksichtigen, um eine spätere Erweiterung auf Mehrbenutzerunterstützung ohne tiefgreifende Änderungen vornehmen zu können.

3.2 Festlegung des Funktionsumfangs

Aus den im vorigen Kapitel hergeleiteten Anforderungen wird im folgenden der zu implementierende Funktionsumfang festgelegt. Hierzu werden die zur Verwaltung von Projekten, zum Arbeiten mit dem Quelltext und zur Durchführung von Messungen notwendigen Funktionen in getrennten Unterkapiteln behandelt.

3.2.1 Verwaltung von Projekten

Unter einem Projekt ist im Rahmen der Versionsverwaltung die Menge aller Informationen zu verstehen, die bei der Parallelisierung und Optimierung einer Anwendung anfallen. Hierzu gehören neben dem Quelltext, dem *Trace Request File* und den ermittelten *Performance*-Daten auch die vom Entwickler geschriebenen Kommentare und Informationen über die Relationen zwischen den einzelnen Daten.

Die Verwaltung von Projekten erfordert im wesentlichen folgende vier Funktionen:

- Anlegen eines neuen Projektes,
- Laden eines Projektes,
- Pflege eines Projektes,
- Löschen eines Projektes.

Beim Anlegen eines Projektes ist es sinnvoll, den Projektnamen und den Namen des Entwicklers zu erfragen. Bei den, für die Mehrbenutzerunterstützung zu implementierenden, Erweiterungen wird aus dem Eintrag für den Namen des Entwicklers eine Liste der Namen der Entwickler. Bereits beim Anlegen eines Projektes soll der Entwickler die Möglichkeit haben, Angaben zu der zu optimierenden Anwendung einzugeben. Daher muß ein Editorfenster für die Eingabe des Projektkommentares angezeigt werden.

Zum Laden eines Projektes ist in einem Fenster die Liste der schon vorhandenen Projekte anzuzeigen. Danach soll sich OPAL in dem Zustand befinden, in dem es sich bei der letzten Bearbeitung dieses Projektes befand.

Eine weitere wichtige Funktion bei der Pflege eines Projektes ist die laufende Aktualisierung des Projektkommentares. Um den Speicherverbrauch zu minimieren, ist eine Funktion zum Löschen der reproduzierbaren Daten eines Projektes, das längere Zeit nicht mehr weitergeführt wird, vorzusehen.

Schließlich muß eine Funktion zum Löschen von abgeschlossenen Projekten existieren. Diese Funktion ist nur nach Bestätigung durch den Entwickler auszuführen, da sie nicht reversibel ist.

3.2.2 Arbeiten mit dem Quelltext

Für den Umgang mit dem Quelltext ist es notwendig

- den Quelltext zu ändern,
- Quelltextversionen zu archivieren,
- auf einer Quelltextversion basierend neue Versionen zu erstellen und
- die Kommentare der Quelltextversionen zu ergänzen.

Bei der Herleitung der Anforderungen ist bereits festgelegt worden, daß zur Änderung des Quelltextes die OPAL -Umgebung nicht verlassen werden soll. Der Editor muß beim Aufruf den gewünschten Quelltext einlesen und anzeigen. Wird ein externer Editor (z.B. `emacs` [55] oder `vi`) verwendet, ist eine Funktion zur manuellen Aktualisierung des Quelltext-Fensters der OPAL -Entwicklungsumgebung notwendig.

Die Archivierung des Quelltextes ermöglicht einen Rückgriff auf frühere Versionen des Quelltextes. Auf der Basis dieser früheren Versionen soll dann die Möglichkeit gegeben sein, alternative Optimierungen durchführen zu können.

Die bei der Analyse der Programm-*Performance* gewonnenen Erkenntnisse müssen im Kommentar der zugehörigen Quelltextversion festgehalten werden können.

3.2.3 Durchführung von Messungen

Die Durchführung und Analyse von Messungen besteht aus folgenden Teilaufgaben:

- Meßkonfigurationen erstellen,
- Meßkonfigurationen archivieren,
- Kommentare der Meßkonfigurationen ergänzen,
- Testläufe starten,
- *Trace*-Dateien laden.

Nachdem eine Meßkonfiguration erstellt wurde, können Testläufe durchgeführt werden. Nicht archivierte Konfigurationen können wie die Arbeitskopie des Quelltextes laufend geändert werden. Vor dem Start eines Testlaufs und vor dem Einlesen von Meßwerten werden Meßwerte, die älter als die aktuelle Meßkonfiguration sind, gelöscht, um die Integrität der Daten zu wahren. Vor dem Ausführen eines Programms ist zudem das *Executable* neu zu erzeugen, falls dieses älter als der aktuelle Quelltext ist. Desweiteren ist die Anzahl der zu verwendenen Prozessoren vor dem Testlauf abzufragen.

Die Möglichkeit, Messungen mit einer archivierten Konfiguration durchzuführen, muß gegeben sein. Die Auswahl einer Konfiguration soll über eine Liste der verfügbaren Konfigurationen erfolgen. Eine archivierte Konfiguration kann nicht nachträglich verändert werden, daher sind hier keine Maßnahmen zur Wahrung der Integrität der Daten notwendig.

3.3 Eignung von verfügbaren Werkzeugen

In der Entwurfsphase sind UNIX-Werkzeuge zur Verwaltung von Quelltexten und Datenbanksysteme auf ihre Eignung für die Versionsverwaltung untersucht worden. Die Werkzeuge müssen in der Lage sein, die in Abbildung 3.1 gezeigte hierarchische Struktur der anfallenden Daten abzubilden. Ein weiterer wichtiger Punkt ist die effiziente Behandlung großer Binärdateien, wie z.B. *Executables* oder *Trace Data Files*.

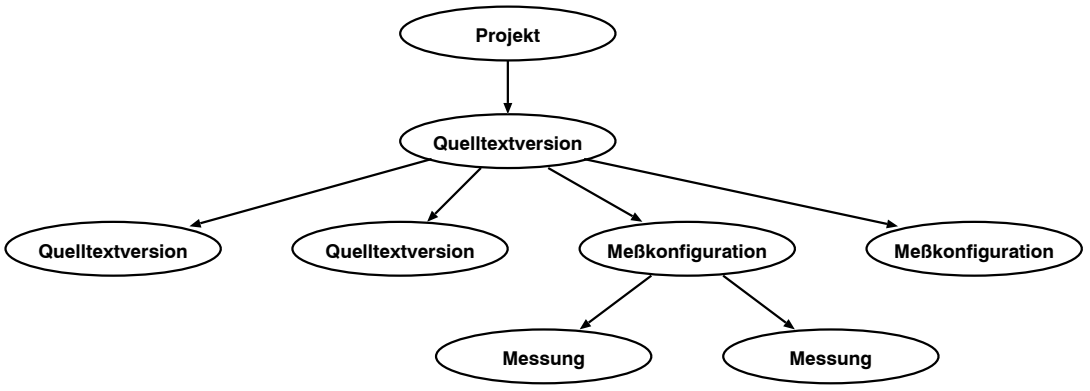


Abbildung 3.1: Hierarchische Struktur der Daten

3.3.1 UNIX-Werkzeuge zur Verwaltung von Quelltexten

Im UNIX-Umfeld sind die Programmpakete `sccs`, `rcs` und `cvs` zur Verwaltung von Quelltexten gebräuchlich. Daneben existieren eine Reihe weiterer derartiger Systeme wie z.B. `Clear/Caster` von IBM, `sdc` von der Carnegie-Mellon Universität oder `cms` von DEC.

Bei vielen kommerziellen UNIX-Systemen ist das von AT&T entwickelte `sccs` (*source code control system*) [36] Bestandteil der Entwicklerversion des Betriebssystems. Es wurde in den *IEEE Transactions on Software Engineering* im Dezember 1975 der Öffentlichkeit vorgestellt und ist das älteste der hier behandelten Systeme zur Verwaltung von Quelltexten.

Unter `sccs` können mehrere Dateien zu einem Projekt, das in ein *Repository* abgebildet wird, zusammengefaßt werden, jedoch wird jede Datei einzeln verwaltet. Bei Projekten, an denen mehrere Entwickler gemeinsam arbeiten, reduziert dieses Vorgehen den Kommunikationsaufwand zwischen den Entwicklern, da jeder Entwickler unabhängig von den anderen neue Versionen seiner Dateien archivieren kann.

Eine Version des Gesamtprojektes wird durch *Tags* (deutsch: Etikett) gebildet, die an die Versionen der einzelnen Dateien angehangen werden. Alle Dateiversionen mit dem gleichen *Tag* bilden dann die Projektversion. Eine Dateiversion kann mehrere solcher *Tags* besitzen, falls diese zwischen zwei oder mehreren aufeinander folgenden Definitionen von Projektversionen nicht verändert wurde.

Eine Versionsnummer besteht aus einer geraden Anzahl natürlicher Zahlen, die durch Punkte abgetrennt sind. Bei rein konsekutiver Programmentwicklung entstehen nur 2-Tupel als Versionsnummern, und bei jeder neuen Version wird die zweite Zahl des Tupels (*minor release number*) um eins erhöht. Die erste Zahl des Tupels (*major release number*) kann vom Entwickler erhöht werden, um z.B. die Einführung wesentlicher Neuerungen zu kennzeichnen. Soll eine Version angelegt werden, die bereits einen Nachfolger besitzt, entsteht ein Zweig, der einen Pfad für die alternativen Nachfolger bereitstellt. Der Zweig erhält die durch eine weitere Zahl erweiterte Nummer der Version, auf die er aufsetzt. So ist `1.2.1` der erste und `1.2.2` der zweite Zweig, der von der Version `1.2` ausgeht. Die Versionen eines Zweiges werden wiederum durchnummeriert, so daß diese ein 2n-Tupel bilden. Abbildung 3.2 illustriert diese Nummerierung, hierbei sind die 2n-Tupel in den Knoten die Dateiversions-

nummern und die (2n+1)-Tupel an den Kanten die Nummer des Zweiges.

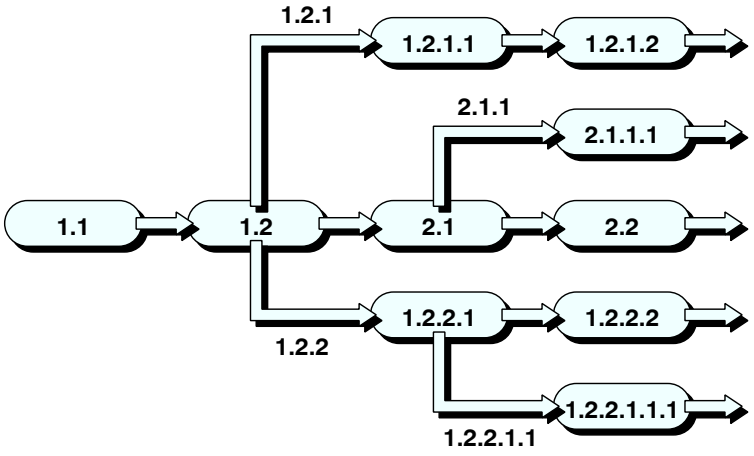


Abbildung 3.2: Aufbau der Versionsnummern

Die Informationen über den Inhalt der Dateien und der vorgenommenen Änderungen werden in einem zentralen Archiv (*repository*) gespeichert. Für jede Quelltext-Datei existiert in diesem Archiv eine Datei, die neben dem Text der initialen Version und einer Beschreibung der Änderungen einer Dateiversion zu ihrem Vorgänger, auch die Kommentare zu den Versionen und die *Tags* zur Festlegung einer Projektversion enthält. Diese Beschreibung der Änderungen wird im folgenden *Deltas* genannt. Eine beliebige Dateiversion wird beim Auslesen aus dem Archiv aus der initialen Version durch sukzessives Ausführen der in den *Deltas* gespeicherten Änderungen rekonstruiert. Diese inkrementelle Art der Speicherung führt zu einer erheblichen Reduktion der Redundanz innerhalb des Archivs und damit auch des Speicherplatzbedarfes. Jeder Entwickler kann sich eine Arbeitskopie der im Archiv gespeicherten Dateien in seinem eigenen Arbeitsverzeichnis durch das Versionsverwaltungssystem erzeugen lassen.

Konflikte, die durch eine gleichzeitige Bearbeitung einer Datei auftreten können, werden dadurch verhindert, daß jeweils nur ein Entwickler zu einer Zeit Schreibrechte auf eine Datei erhält. Will ein Entwickler eine Änderung an einer Datei vornehmen, muß er zunächst für diese Datei Schreibrechte beantragen. Danach kann er die Datei ändern und durch Erzeugen einer neuen Dateiversion wieder freigeben.

Anfang der 80er Jahre startete Walter Tichy an der Purdue Universität die Entwicklung von `rcs` (*revision control system*) [37, 38]. Dieses Programmpaket wurde seitdem von mehreren Entwicklern erweitert und ist bereits seit einigen Jahren für nahezu alle UNIX-Systeme verfügbar. Ab der vierten Version, die 1989 erschien, unterliegt es der GNU General Public Licence.

Der Grund für die Entwicklung von `rcs` war, daß das von AT&T entwickelte `sccs` nicht für BSD UNIX-Systeme zur Verfügung stand, die in dieser Zeit im universitären Bereich dominierten. Es bietet die gleiche Funktionalität wie `sccs`, arbeitet jedoch aufgrund einer effektiveren Implementierung deutlich schneller [38]. So kann z.B. auf die letzte Version ohne sukzessives Ausführen der *Deltas* zugegriffen werden.

Das dritte in diesem Kapitel behandelte System ist `cvs` (*concurrent versions system*) [39, 40]. Urspünglich war `cvs` eine Sammlung von *Shell*-Skripten, die ein *Front-End* für `rcs` bildeten. Die erste Version wurde 1986 freigegeben und stammt von Dick Grune von der Freien Universität Amsterdam. Wie `rcs` unterliegt auch `cvs` der GNU General Public Licence.

Das Sperren einer Datei durch `sccs` oder `rcs` führt zu Wartezeiten, falls mehrere Entwickler gleichzeitig an einer Datei arbeiten müssen. Daher bietet `cvs` als wesentliche Erweiterung gegenüber `sccs` und `rcs` Algorithmen zum Zusammenfassen von parallel ausgeführten Änderungen einer Datei, die ein Sperren von Dateien unnötig machen. Archiviert ein Entwickler unter `cvs` seine geänderte lokale Arbeitskopie einer Datei als neue Dateiversion, werden, falls zwischenzeitlich andere Entwickler neue Versionen dieser Datei erzeugt haben, die Änderungen der anderen Entwickler in seiner lokalen Kopie nachgetragen. Treten hierbei Konflikte auf, d.h. sind Zeilen einer Quelltextdatei von mehreren Entwicklern parallel verändert worden, werden diese gemeldet und müssen manuell beseitigt werden, bevor `cvs` die lokale Arbeitskopie als neue Dateiversion annimmt.

Eine weitere Neuerung gegenüber `sccs` und `rcs` ist die Unterstützung von mehreren Projekten innerhalb eines Archivs. Diese Erweiterung umfaßt auch Befehle, die das gesamte Projekt betreffen. So können mit einem einzelnen Befehl Projekte ausgelesen werden oder alle Dateien einer lokalen Arbeitskopie eines Projektes aktualisiert werden.

Alle in diesem Kapitel besprochenen Werkzeuge sind nicht in der Lage, Abhängigkei-

ten zwischen den von ihnen verwalteten Dateien abzubilden. Darüberhinaus sind sie als Werkzeuge zur Verwaltung von Quelltexten nicht geeignet, große Binärdateien wie Trace-Dateien effizient zu speichern. Die Algorithmen, die `cvs` zur Kombination parallel an einer Datei ausgeführter Änderungen zur Verfügung stellt, sind jedoch für die Versionsverwaltung in Hinblick auf die optionale Anforderung, Teamarbeit zu unterstützen, verwendbar.

3.3.2 Datenbanksysteme

Stellvertretend für die relationalen Datenbanken [41], die die Datenbanksprache SQL [42] verwenden, ist das im Zentralinstitut für Angewandte Mathematik des Forschungszentrum Jülich verwendete `Oracle7` [43] von Oracle Corporation betrachtet worden. Daneben ist `OBST` [44, 45, 46], eine vom Forschungszentrum Informatik (FZI) in Karlsruhe entwickelte C++ Klassenbibliothek analysiert worden, die dem Entwickler Datenbankfunktionen bereitstellt. Dieses Werkzeug basiert auf einem objektorientierten Ansatz.

Das hierarchische Datenbankmodell wäre zur Abbildung der in Abbildung 3.1 dargestellten hierarchischen Abhängigkeiten der Daten geeignet. Es ist jedoch vom relationalen Datenbankmodell weitgehend verdrängt worden.

In relationalen Datenbanken werden die Datensätze gleichen Typs in Mengen zusammengefaßt. Ein Datensatz ist ein Tupel der Eigenschaften eines Datums. Zwei Mengen können durch Relationen verknüpft werden, falls ihren Elementen eine oder mehrere gemeinsame Eigenschaften zugeordnet sind. Diese Art der Abbildung der Daten ist wenig geeignet, um die hierarchischen Abhängigkeiten der Daten wiederzugeben, die innerhalb der Parallelisierung und Optimierung von Anwendungen anfallen. Zudem ist die Wahrung der referentiellen Integrität bei hierarchischen Abhängigkeiten mit einem hohen Aufwand verbunden.

Dieser Nachteil des relationalen Datenbankmodells ist im objektorientierten Datenbankmodell behoben. Hierarchische Abhängigkeiten werden direkt durch Objekthierarchien abgebildet. In der C++ Klassenbibliothek `OBST` hat der Entwickler neben den Objekthierarchien Klassen zur Behandlung von Listen, Bäumen und Mengen zur Verfügung, um die Struktur der Daten nachzubilden.

Ein den relationalen und objektorientierten Datenbanksystemen gemeinsamer Nach-

teil ist, daß sowohl beim Anlegen von Quelltextversionen oder Meßkonfigurationen, als auch bei dem Archivieren von Meßdaten Dateien in Datensätze konvertiert werden müssen. Beim Auslesen sind diese Datensätze wieder in Dateien umzuwandeln. Da es sich gerade bei den Meßdaten um große Dateien handelt, führt dies zu unnötig hohen Wartezeiten.

3.4 Konzepte der Versionsverwaltung

Weder die UNIX-Werkzeuge zur Verwaltung von Quelltexten, noch die Datenbanksysteme können den für die Versionsverwaltung notwendigen Funktionsumfang vollständig abdecken. Dennoch ist das Konzept der Benutzung eines zentralen Archivs, das neben den weiteren Konzepten im folgenden erläutert wird, von diesen Werkzeugen übernommen worden.

3.4.1 Das Work/Version-Konzept

Während der Analyse einer Programmversion darf diese nicht verändert werden, um die Integrität zwischen Quelltext und Meßergebnissen zu wahren, sowie um nachträgliche Messungen zu ermöglichen. Daher exisitiert in einem Projekt genau eine Arbeitskopie des Quelltextes (*work copy*). Änderungen des Quelltextes werden ausschließlich an dieser Arbeitskopie durchgeführt. Momentaufnahmen dieser Arbeitskopie werden als Quelltextversionen (*versions*) des Projektes archiviert.

Der Kern des *Work/Version*-Konzeptes ist die strikte Trennung zwischen der Arbeit mit der Arbeitskopie und dem Arbeiten mit den Quelltextversionen. Daher werden in der Versionsverwaltung die beiden Zustände `Work` und `Version` unterschieden. Alle Aktionen, die auf der Arbeitskopie des Quelltextes basieren, können nur im Zustand `Work` ausgeführt werden. Für die Messungen an einer Quelltextversion muß sich die Versionsverwaltung entsprechend im Zustand `Version` befinden.

Eine Auswirkung dieses Konzeptes ist, daß bei einem Rückgriff auf eine Quelltextversion die aktuelle Arbeitskopie überschrieben wird. Die an der Arbeitskopie seit der letzten Definition einer Quelltextversion durchgeführten Änderungen gehen hierbei verloren.

Für die Arbeitskopie ist, wie bisher, nur eine Meßkonfiguration zu einer Zeit möglich, da hier die Integrität zwischen Quelltext und Meßergebnissen nicht gewahrt ist. Für

Quelltextversionen werden hingegen durch die Versionsverwaltung mehrere parallel existierende Meßkonfigurationen unterstützt. Das wird durch die Unveränderbarkeit eines archivierten Quelltextes ermöglicht. Zur ausführlichen Analyse der Programm-*Performance* ist daher eine Quelltextversion zu erstellen.

3.4.2 Nutzung von cvs zur Archivierung der Quelltexte

Durch die Verwendung von `cvs` zur Archivierung der Quelltexte sind die zuvor näher beschriebenen Funktionen zum Zusammenfassen parallel ausgeführter Quelltextänderungen verfügbar. Damit ist die Versionsverwaltung auf die Erweiterung zur Mehrbenutzerunterstützung vorbereitet.

Ein weiterer Gewinn, der aus der Nutzung von `cvs` resultiert, ist die Speicherverbrauchreduzierung bei inaktiven, bis auf die Quelltexte und Meßkonfiguration reduzierten Projekten aufgrund der redundanzfreien Speicherung der Quelltexte im `cvs`-*Repository*.

3.4.3 Zentrales Archiv für Daten

Das Konzept, ein zentrales Archiv zur Speicherung von Daten zu verwenden, wie es bei den Werkzeugen zur Quelltextverwaltung und den meisten Datenbanksystemen Verwendung findet, wird für die Archivierung der Messungen in der Versionverwaltung übernommen, kommt jedoch erst bei der Erweiterung auf Mehrbenutzerunterstützung zur Geltung.

3.4.4 Die Rolle des Dateisystems

UNIX-Dateisysteme decken alle von der Versionsverwaltung benötigten Datenbankfunktionen zum Anlegen und Löschen einzelner Datensätze, sowie für den Zugriff auf die Dateien, ab. Auch Funktionen zur Vergabe und Kontrolle von Zugriffsrechten stehen zur Verfügung. Weiterhin bilden die Dateisystemfunktionen zum Sperren und Freigeben einzelner Dateien eine Basis zur Implementierung eines *Single Writer, Multiple Reader*-Zugriffs zur Erhaltung der Kohärenz. Zudem wird die hierarchische Abhängigkeit der Daten durch einen Verzeichnisbaum (vgl. Abbildung 3.3) abgebildet, ohne eine Transformation der Struktur durchführen zu müssen.

Gegenüber den Datenbanksystemen entfällt bei der Verwendung des Dateisystems

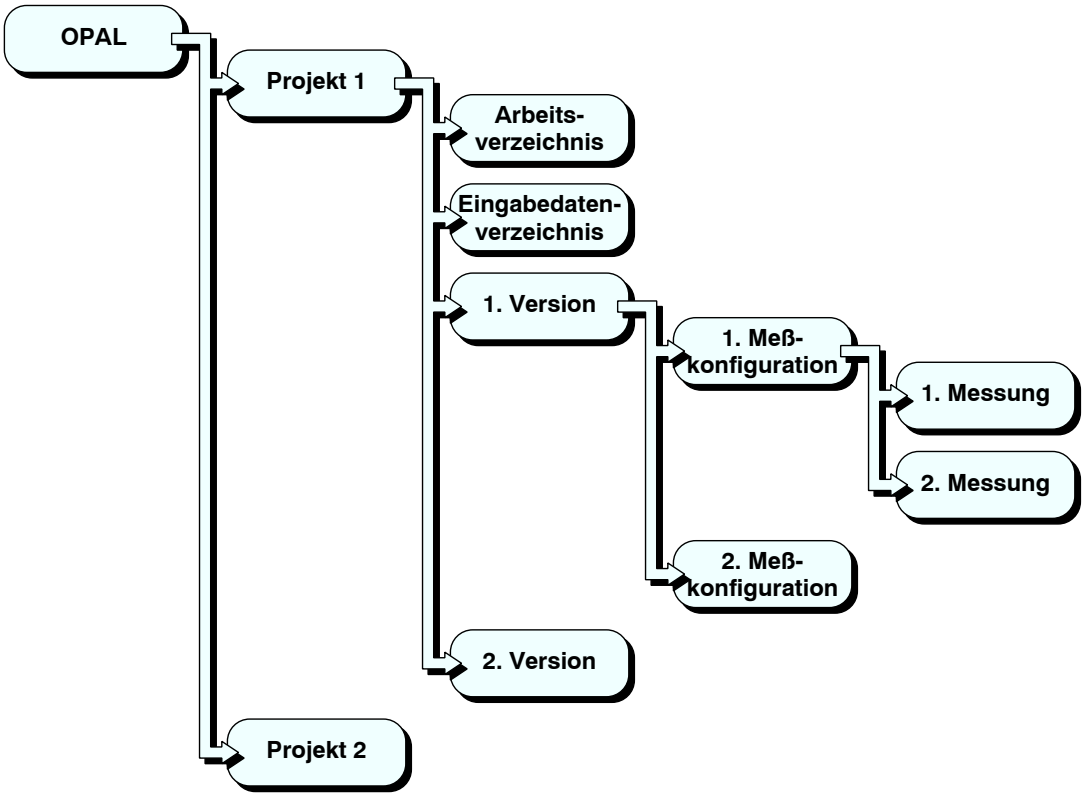


Abbildung 3.3: Verzeichnisstruktur der Versionsverwaltung

als Datenbank die Konvertierung der Dateien zu Datensätzen. Ein weiterer Vorteil ist die Verfügbarkeit auf allen UNIX-Systemen.

3.4.5 Kommentare als Orientierungshilfe

Beim Erzeugen von neuen Quelltextversionen oder Meßkonfigurationen wird der Entwickler aufgefordert, einen Kurzkommentar (*abstract*) und einen ausführlichen Kommentar einzugeben. Diese Kommentare werden durch die von der Versionsverwaltung generierten Informationen, wie Datumsangaben und *Compiler*-Optionen, ergänzt. In der Versionsverwaltung dienen die Kommentare neben der Dokumentation auch zur Orientierung innerhalb des Versionsbaums und den einzelnen Meßkonfigurationen. Zur Auswahl einer Quelltextversion wird eine graphische Darstellung des Versionsbaums generiert, in der die einzelnen Versionen mit dem ersten Wort des Kurzkommentares beschriftet sind. Ergänzend wird eine Liste mit den Kurzkommentaren und ein Fenster mit dem ausführlichen Kommentar der ausgewählten

Version angezeigt. Eine Version wird entweder durch einen „Mausklick“ auf den Versionsbaum oder über die Versionsliste selektiert. Analog wird zur Auswahl einer Meßkonfiguration eine Liste der Kurzkommentare verwendet.

Der Kommentar einer Quelltextversion muß, um zur Orientierung dienen zu können,

- die durchgeführten Änderungen,
- den Grund der Änderungen,
- die beim Testen der Version ermittelten Ergebnisse und
- die aus den Ergebnissen gewonnenen Schlußfolgerungen

stichwortartig charakterisieren.

Der Kommentar für eine Meßkonfiguration beschreibt entsprechend

- die durchgeführten Messungen,
- die Meßwerte und
- die Schlußfolgerungen.

Sowohl für den Kommentar einer Quelltextversion, als auch für den einer Meßkonfiguration existieren „Formblätter“, die vor der Eingabe des Kommentares in den *Editor* geladen werden. Diese „Formblätter“ geben die Struktur des Kommentares vor und vereinheitlichen damit das Aussehen, so daß das Auffinden bestimmter Informationen beschleunigt wird.

3.4.6 Erweiterung auf Mehrbenutzerunterstützung

Bei der Optimierung umfangreicher Anwendungen ist eine Verteilung der Aufgaben auf mehrere Entwickler notwendig. Hierbei wird die Arbeitsverteilung derart gewählt, daß jeder einzelne Entwickler seine Programmteile weitgehend lokal, das heißt ohne viel Kommunikation mit anderen am Projekt beteiligten Entwicklern, implementieren kann. Analog wird auch bei der Optimierung von SVM-Fortran-Anwendungen eine hohe Datenlokalität angestrebt, um die Kommunikation zwischen den Prozessoren zu minimieren. Hierzu ist eine globale Verteilung der Arbeitslast auf die Prozessoren notwendig, was eine Betrachtung der gesamten Anwendung als eine Einheit und somit eine enge Kommunikation zwischen den Entwicklern bedingt. Daher ist bei der Optimierung von SVM-Fortran-Anwendungen eine enge

Kommunikation zwischen allen Entwicklern notwendig. Das Kommunikationsaufkommen unter den Entwicklern und das im Programm ist also im allgemeinen nicht gleichzeitig zu minimieren. Aufgrund der $\mathcal{O}(n^2)$-Komplexität dieses *All-To-All*-Kommunikationsmusters der Entwickler ist die Größe des Entwicklerteams eng begrenzt.

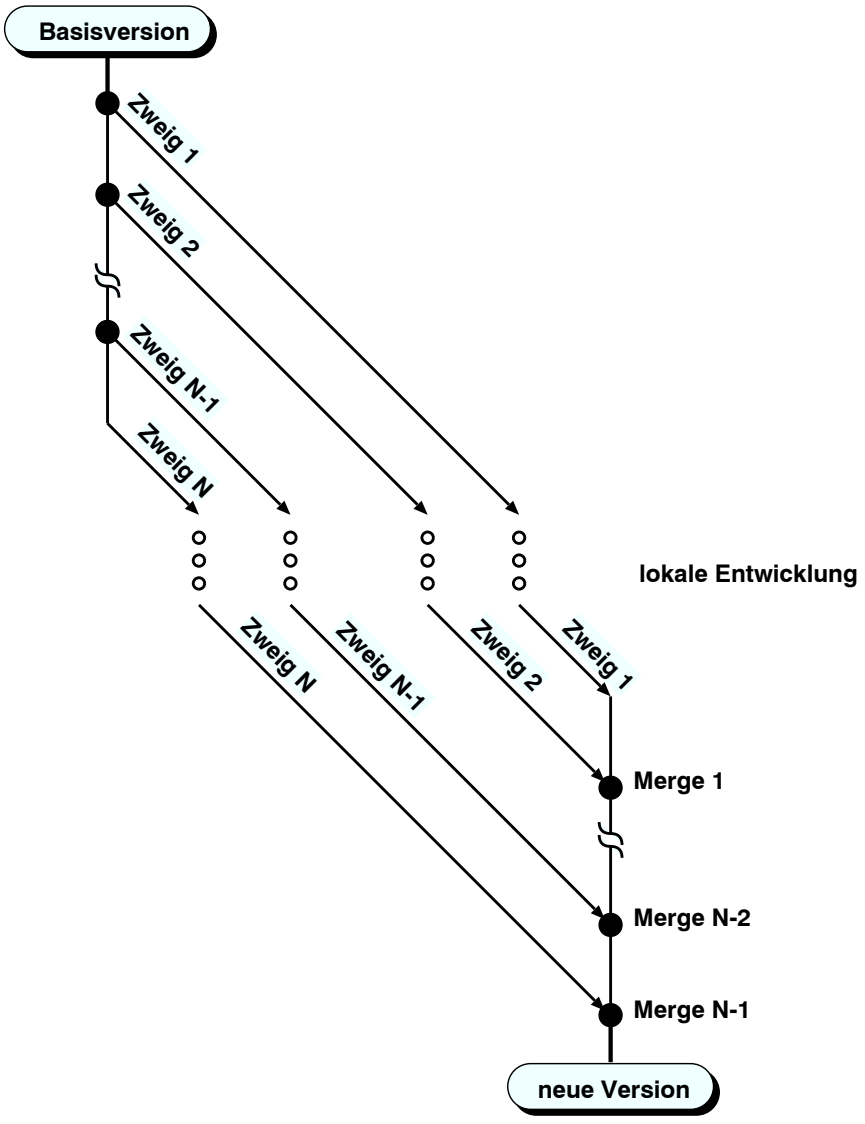


Abbildung 3.4: Vorgehen bei der Optimierung im Team

Die geplante Mehrbenutzerunterstützung der Versionsverwaltung basiert daher auf folgendem Ansatz: Ausgehend von einer Version wird gemeinsam eine Strategie ent-

wickelt. Danach führt jeder Entwickler die ihm zugeteilten lokalen Änderungen aus. Hierbei entstehen, wie in Abbildung 3.4 dargestellt, mehrere parallel laufende Entwicklungspfade, die nach Fertigstellung der Änderungen zu einer Version kombiniert werden, die die Basis für weitere Optimierungen ist. Für das Verschmelzen der Entwicklungspfade sind die *Merging*-Funktionen von `cvs` [39] geeignet.

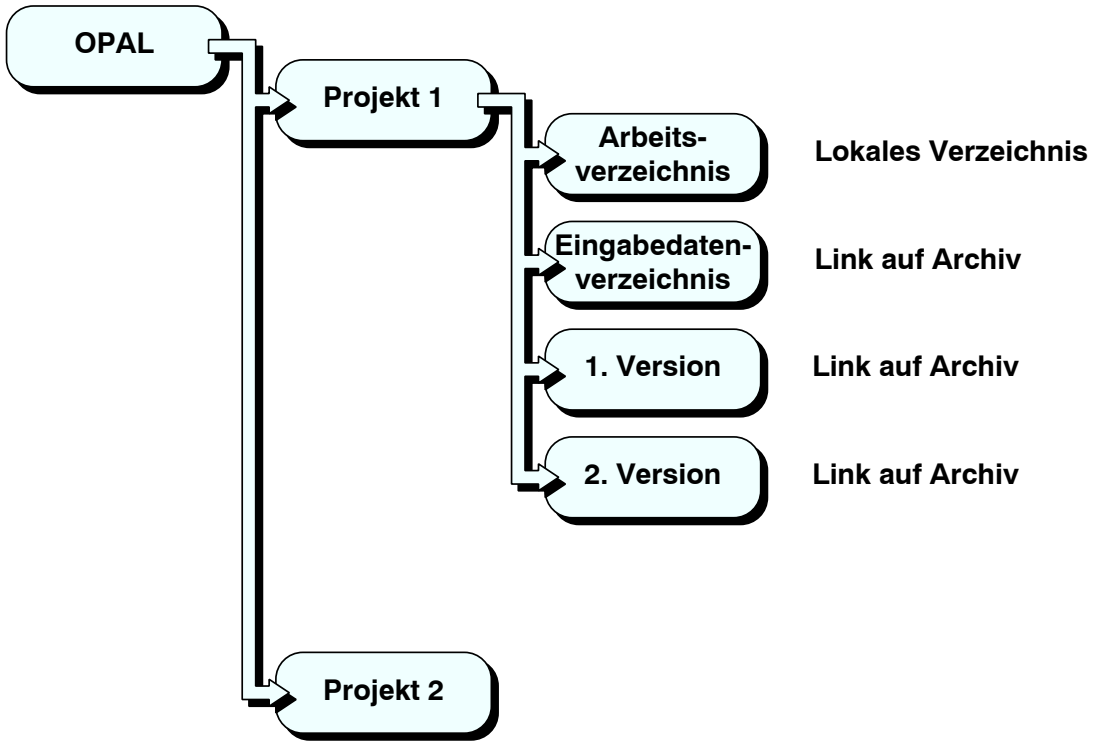


Abbildung 3.5: Verzeichnisstruktur für Mehrbenutzerunterstützung

Nach der Erweiterung auf Mehrbenutzerunterstützung existiert jeweils nur eine globale Instanz der Versionsverzeichnisse, auf die im *Single Writer, Multiple Reader*-Konzept zugegriffen wird. In die lokalen Verzeichnisstrukturen der Entwickler werden diese globalen Versionsverzeichnisse, wie in Abbildung 3.5 dargestellt, durch *Links* eingebunden.

Im Gegensatz zum Quelltext eines Projektes sind die Meßkonfigurationen und Meßdaten WORM-Daten (WORM: *write once, read multiple*), das heißt, sie werden einmal geschrieben und danach gegebenenfalls mehrfach gelesen. Daher ist hier kein *Single Writer, Multiple Reader*-Mechanismus notwendig.

Ein weiterer Aspekt ist die Speicherung des Zustands der Versionsverwaltung. Nach dem Aufruf befindet sich OPAL in dem Zustand, in dem es zuvor verlassen wurde. Hierzu speichert OPAL seinen Zustand beim Beenden in einer Datei ab. Diese Datei muß oberhalb der Projektverzeichnisse abgelegt werden und ist daher lokal, so daß hier keine besonderen Zusatzmaßnahmen zu treffen sind.

Kapitel 4

Die Implementierung

Dieses Kapitel beschreibt ausgewählte Aspekte der Implementierung, die zum Verständnis notwendig erscheinen. Hierzu wird zunächst die Einbindung der Versionsverwaltung in OPAL erläutert, da diese eine Referenz für die Verwendung der Versionsverwaltung innerhalb einer Programmanalyse-Umgebung darstellt. Danach wird auf den von der Versionsverwaltung aufgebauten Verzeichnisbaum und im Anschluß auf die zur Verwaltung der Verzeichnisse verwendeten *Shell*-Skripte detailiert eingegangen. Nachdem damit das Umfeld beschrieben ist, wird mit der internen Darstellung eines Projektes der Kern der Versionsverwaltung besprochen. In den das Kapitel abschließenden Anmerkungen zur Implementierung werden bislang noch nicht erwähnte Hintergrundinformationen zu einigen bereits beim Entwurf getroffenen Entscheidungen, die die Implementierung wesentlich beeinflußten, zusammengefaßt.

4.1 Einbindung in OPAL

4.1.1 Erweiterung der Bedienoberfläche

Vor der Festlegung des Funktionsumfangs war die Frage, ob die Versionsverwaltung besser als externes Werkzeug oder in OPAL integriert zu implementieren sei, offen. Im Lastenheft geforderte Funktionalität, wie z.B. das Ausführen von Testläufen oder das Ändern des Quelltextes ohne Verlassen der OPAL -Umgebung, erfordert eine Integration der Versionsverwaltung in OPAL und ist der Grund der Entscheidung für diese Variante der Implementierung. Die Bedienoberfläche von OPAL basiert auf der Motif-Bibliothek, so daß durch diese Wahl die Verwendung von Motif auch für die Benutzerführung innerhalb der Versionsverwaltung notwendig ist.

Zur Einbindung der Versionsverwaltung in OPAL wurde die Menüleiste von OPAL durch die Menüpunkte `Work` und `Version` erweitert und der Menüpunkt `System` durch `Project` sowie der Menüpunkt `Information` durch `Source` ersetzt.

Im `Project`-Menü werden alle Funktionen zum Verwalten von Projekten zusammengefaßt. Es wird mit

New Project
: ein neues Projekt angelegt,

Visit Project
: ein Projekt besucht,

Save Project
: der aktuelle Zustand des Projektes und der Versionsverwaltung in zwei Dateien gesichert,

Maintain Project
: im Unterpunkt `Edit Comment` Projektkommentar ergänzt oder mit dem Unterpunkt `Clean` das Projekt durch Löschen aller reproduzierbaren Dateien komprimiert,

Delete Project
: nach Rückfrage das gesamte Projekt gelöscht.

Die beiden nächsten Punkte der Menüleiste `Work` und `Version` sind mit zwei gleichnamigen, sich gegenseitig ausschließenden Zuständen der Bedienoberfläche verbunden, die sich aus der konsequenten Fortführung des *Work/Version*-Konzeptes in der Benutzerführung ergeben. Im Zustand `Work` sind die Funktionen zum Arbeiten mit der Arbeitskopie des Quelltextes, jedoch nicht die für das Arbeiten mit den Quelltextversionen verfügbar. Umgekehrt sind im Zustand `Version` z.B. *Performance*-Messungen an einer Quelltextversion möglich, aber die Funktionen, die auf den Quelltext im Arbeitsverzeichnis zugreifen, sind gesperrt. Funktionen für einen Zustandswechsel werden in den `Work` und `Version` Menüs über die abgetrennten, an letzter Position stehenden Menüpunkte `Switch To Version` bzw. `Switch To Work` angeboten. Darüberhinaus wechselt beim Archivieren des Quelltextes im Arbeitsverzeichnis als neue Quelltextversion der Zustand von `Work` auf `Version`, so daß ein expliziter Zustandswechsel für die Durchführung der anschließenden Messungen entfällt.

Das `Work`-Menü enthält die Funktionen zum Arbeiten mit der Arbeitskopie des Quelltextes. Hier wird mit dem Menüpunkt

Archive Source
: eine neue Quelltextversion aus der Arbeitskopie des Quelltextes erstellt;

Retrieve Version
: der Quelltext im Arbeitsverzeichnis durch den Quelltext einer Quelltextversion überschrieben.

Wie bereits beschrieben dient der Menüpunkt

Switch To Version
: zum Wechseln vom `Work`-Zustand in den `Version`-Zustand.

Im `Version`-Menü werden Funktionen zum Arbeiten mit den Quelltextversionen angeboten. Diese sind in drei Gruppen unterteilt, wobei in der ersten Gruppe mit

Visit Other Version
: zwischen zwei Quelltextversionen gewechselt und mit

Maintain Version
: analog zum Menüpunkt `Maintain Projekt` des `Project`-Menüs im Unterpunkt `Edit Comment` der Versionskommentar ergänzt und mit dem Unterpunkt `Clean` die angezeigte Version durch Löschen aller reproduzierbaren Dateien komprimiert wird.

In der zweiten Gruppe sind die Funktionen zum Erstellen und Verwalten von Meßkonfigurationen zusammengefaßt. Hier dient

Visit Trace Config
: zum Besuchen einer Meßkonfiguration,

Archive Trace Config
: zum Archivieren der aktuellen Instrumentierung als neue Meßkonfiguration,

Maintain Trace Config
: wie bereits bei den Projekten und Versionen zum Ergänzen des Kommentares bzw. zum Löschen reproduzierbarer Dateien, also in diesem Fall der Meßwerte.

Symmetrisch zum Menüpunkt `Switch To Version` im `Work`-Menü existiert hier der Menüpunkt

Switch To Work
zum Wechseln vom `Version`-Zustand in den `Work`-Zustand.

Das `Source`-Menü enthält, neben den vom `Information`-Menü übernommenen Menüpunkten zur Beeinflußung der Quelltextdarstellung im Quelltextfenster die beiden neuen Menüpunkte

Launch Program
zum Starten eines Testlaufs und

Edit Source
zum Bearbeiten der Arbeitskopie des Quelltextes.

Neben der Implementierung der neuen Funktionen wurden existierende Funktionen umbenannt, um dem Anwender durch eine einheitliche, kompakte Nomenklatur das Erlernen der Menüstruktur zu erleichtern.

Die Möglichkeit, einfach zwischen Projekten, Quelltextversionen und Meßkonfigurationen wechseln zu können, bedingt eine Anzeige was momentan bearbeitet wird. Daher ist unterhalb des Quelltextfensters eine Statuszeile eingefügt worden. In vier Feldern wird dem Entwickler das Projekt, die Quelltextversion, die Meßkonfiguration und die Anzahl der beim Testlauf verwendeten Prozessoren angezeigt. Im Versions- und Meßkonfigurationsfeld wird wie in den Auswahllisten der vom Entwickler eingegebene, einzeilige Kurzkommentar verwendet. Befindet sich die Versionsverwaltung im Zustand `Work`, wird im Versionsfeld das Schlüsselwort `work`, gefolgt vom Kurzkommentar der Quelltextversion, auf der die Arbeitskopie des Quelltextes basiert, angezeigt.

4.1.2 Schnittstelle zum Parser und zum Trace Analyzer

Bei der Herleitung der Anforderungen an die Versionsverwaltung aus den mit der Arbeit mit OPAL gewonnenen Erfahrungen ist bereits beschrieben worden, daß OPAL bis auf die *Trace Data Files* alle Dateien in dem Verzeichnis erwartet, in dem es gestartet wurde. Die Versionsverwaltung legt jedoch für jede einzelne Version des Programms ein eigenes Verzeichnis an und speichert die Meßkonfigurationen in Unterverzeichnisse dieses Verzeichnisses ab. Hier existieren zwei Wege das Problem zu lösen. Eine Lösung ist, OPAL im Arbeitsverzeichnis des gewählten Projektes zu starten und für jede besuchte Meßkonfiguration eine neue Instanz von OPAL zu erzeugen. Diese Methode bietet den Vorteil, daß alle Meßkonfigurationen gleichzeitig

sichtbar sind und damit ein Vergleich zweier Versionen direkt möglich ist. Nachteilig ist der hohe Hauptspeicherverbrauch.

Die zweite Variante ist, die Schnittstellen vom *Parser* und vom *Trace Analyzer* so zu erweitern, daß die Unterverzeichnisse adressiert werden. Hierdurch ist nur eine Instanz von OPAL notwendig. Bei einem Wechsel zwischen zwei Versionen eines Programms oder zwei Messungen einer Programmversion muß jedoch der Quelltext bzw. die Meßwerte neu eingelesen und dargestellt werden. Der geringere Hauptspeicherverbrauch wird daher durch einen höheren Prozessor- und Datentransferaufwand erkauft, der sich in einer längeren Wartezeit niederschlägt.

In der Versionsverwaltung ist die zweite Variante implementiert. Hierzu ist anzumerken, daß in OPAL der, auch im SVM-Fortran-*Compiler* verwendete, PAFF Fortran *Parser* [17] zum Einlesen des Quelltextes genutzt wird. OPAL wandelt vor der Quelltextanalyse den vom PAFF-*Parser* erzeugten Syntaxbaum in seine interne Datenstruktur um. Neben dieser Datenstruktur werden zur Darstellung der in einem Testlauf ermittelten *Performance*-Daten die *Trace Data Files* benötigt.

Im Verlauf der Änderungen an den Schnittstellen wurde die relative Adressierung sämtlicher Dateien durch eine absolute Adressierung ersetzt. Der Hauptvorteil dieser Umstellung ist, daß OPAL jetzt in einem beliebigen Verzeichnis gestartet werden kann. Insbesondere kann hierdurch OPAL über ein Menü des *Window Managers* aufgerufen werden.

4.1.3 Initialisierung und Terminierung

Die Initialisierung der internen Datenstruktur beim Starten und die Sicherung dieser beim Verlassen der Versionsverwaltung wird durch den Aufruf der Funktion `loadState` bzw. `saveState` veranlaßt. Beide Funktionen benötigen keine Motif-Umgebung, so daß sie vor der Initialisierung bzw. nach der Terminierung des eigentlichen Analyse-Werkzeugs aufgerufen werden können.

4.2 Der Verzeichnisbaum

Die Struktur des Verzeichnisbaums ist bereits bei der Vorstellung der Konzepte beschrieben worden. Im folgenden werden daher die Inhalte der einzelnen Dateien im Vordergrund stehen. In der Darstellung des Verzeichnisbaums in Abbildung 4.1

ist daher die Struktur aus Abbildung 3.3 um die Dateien erweitert.

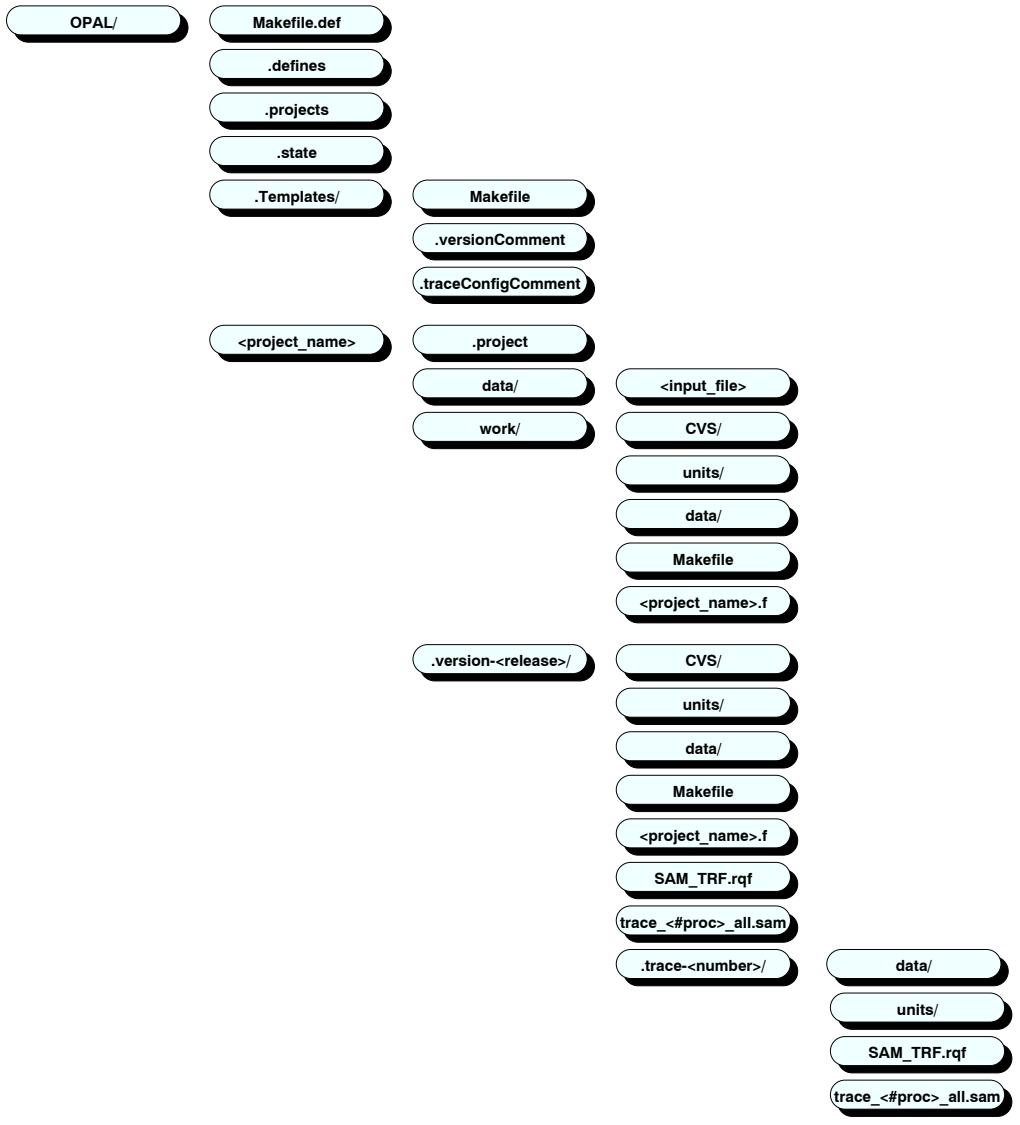


Abbildung 4.1: Die Verzeichnisstruktur

Ein *Link* mit dem Namen `.opal` im *Home*-Verzeichnis der *Workstation*, auf der OPAL abläuft und dem Parallelrechner des Entwicklers zeigt auf das Basisverzeichnis der Versionsverwaltung und ermöglicht eine absolute Adressierung der Dateien, bei gleichzeitig freier Wahl der Position des Basisverzeichnisses. Hierzu wird vorausgesetzt, daß der Parallelrechner sowohl die Möglichkeit des lesenden, als auch des schreibenden Zugriffs auf das Dateisystem der *Workstation* hat. Auf dem verwende-

ten Referenzsystem ist dieser Zugriff durch das *Network File System* (`NFS`) realisiert. Zudem wird eine Möglichkeit benötigt, von der *Workstation* aus Programme auf dem Parallelrechner zu starten, um auf dem Parallelrechner den `.opal`-*Link* zu erzeugen oder einen Testlauf zu starten. Dies ist auf dem Referenzsystem durch *Remote Shell* Aufrufe implementiert.

4.2.1 Das Basisverzeichnis der Versionsverwaltung

Das Basisverzeichnis enthält von allen Projekten benötigte Informationen. Hierzu gehört die Beschreibung der Konfiguration (`.defines`), die Liste der Projekte (`.projects`), der Zustand der Versionsverwaltung (`.state`) und eine *Include*-Datei für das *Makefile* mit dem Namen `Makefile.def`. Auch das Verzeichnis `.Templates/`, in dem die Musterdateien (*Templates*) gespeichert sind, ist im Basisverzeichnis enthalten.

Die Beschreibung der Rechnerkonfiguration (Abbildung 4.2) wird in *Shell*-Skripte eingebunden und entspricht daher der Bourne-Shell-Syntax. Um Namensgleichheiten mit anderen Umgebungsvariablen zu vermeiden, beginnen alle Namen der hier definierten Variablen dem Präfix `OPAL_`.

```
OPAL_CVS_REPOSITORY_DIR=/home/zdvex/zdv172/opal/cvs-repository
OPAL_SVM_HOST_LOGIN=zdv172
OPAL_SVM_HOST=zam358.zam.kfa-juelich.de
```

Abbildung 4.2: Beispiel einer Rechnerkonfiguration

Die Variable `OPAL_CVS_REPOSITORY_DIR` enthält den absoluten Pfad zum Basisverzeichnis des `cvs`-*Repository*. Die Versionsverwaltung kann somit ein bereits existierendes sowie ein eigenes `cvs`-*Repository* nutzen.

Mit den Variablen `OPAL_SVM_HOST_LOGIN` und `OPAL_SVM_HOST` wird der *Login* des Entwicklers auf dem Parallelrechner und die Adresse dieses Rechners festgelegt. Durch die Definition des *Login* innerhalb dieser Datei wird der Tatsache, daß externe Anwender von Parallelrechnern zumeist auf ihrer privaten *Workstation* einen anderen *Login* als auf dem Parallelrechner besitzen, Rechnung getragen.

Bei der Erweiterung der Versionsverwaltung auf Mehrbenutzerunterstützung kann

entweder die Limitierung auf einen *Login* beibehalten werden, d.h. alle Entwickler haben einen gemeinsamen *Login* auf dem Parallelrechner, oder es kann die Variable `OPAL_SVM_HOST_LOGIN` durch eine Tabelle ersetzt werden. Notwendig für diese Erweiterung ist jedoch das Hinzufügen einer Variablen, die den absoluten Pfad zum Basisverzeichnis des dann existierenden zentralen Archivs enthält.

Die Datei `.projects` enthält eine Liste der Projektnamen. Die Versionsverwaltung verwendet für die Benennung von Projektverzeichnissen diesen Namen.

Abbildung 4.3 zeigt eine `.state`-Datei, in der die Versionsverwaltung beim Beenden einer *Session* eine Beschreibung ihres Zustands abspeichert. Sie enthält als ersten Eintrag ein *Flag*, das angibt, ob sich die Versionsverwaltung zuletzt im `Work`- oder im `Version`-Zustand befand. Im nächsten Eintrag wird der Projektname gespeichert. Darauf folgen die Versionsnummern der im Quelltextfenster angezeigten, der für die Arbeitskopie zugrundeliegenden und der zuletzt besuchten Quelltextversion. Zur Spezifikation der Quelltextversionen wird eine von der Versionsverwaltung generierte, eindeutige Versionsnummer verwendet, die auch für die Kennzeichnung der Quelltextversionen innerhalb des `cvs`-Archivs genutzt wird. Im letzten Eintrag ist die zuletzt angezeigte Meßkonfiguration gesichert. Falls im Projekt noch keine Quelltextversion oder Meßkonfiguration besucht wurde, wird das Schlüsselwort `none` anstelle der Nummer ausgegeben.

```
CurrentWorkVersionFlag version
CurrentProject QMR-Freund
CurrentVersion 2.1.1
ActualVersion 4
VisitedVersion 2.1.1
CurrentTraceConfig 3
```

Abbildung 4.3: Darstellung des Zustands

Die zum Übersetzen von SVM-Fortran-Programmen verwendeten `Makefiles` enthalten größtenteils vom einzelnen Programm unabhängige Informationen, wie etwa die Pfade zum Übersetzer und Binder. Diese Angaben sind in der Datei `Makefile.def`, die vom jeweiligen `Makefile` gelesen wird, zentral abgelegt.

Im *Template*-Verzeichnis befinden sich ein Muster für die *Makefiles* und „Form-

blätter“ für den Quelltext- und den Meßkonfigurationskommentar. Beim Aufruf von `make` wird der Projektname übergeben und als Präfix für die Namen des Hauptquelltextes und des *Executable* verwendet. Die *Include*-Dateien werden von der Versionsverwaltung automatisch ermittelt, so daß der Entwickler nur noch projektspezifische Optimierungsoptionen eintragen muß.

4.2.2 Die Projektverzeichnisse

In einem Projektverzeichnis befindet sich neben den Verzeichnissen für die Arbeitskopien des Quelltextes, den Eingabedaten und den Quelltextversionen nur noch die Datei `.project`, in der alle das jeweilige Projekt betreffenden Informationen gesichert werden.

```
...

ProjectBegin QMR-Freund
ProjectNameLength 10
ProjectName QMR-Freund
ProjectAuthorNameLength 12
ProjectAuthorName Carsten Pitz
ProjectCommentLength 42
ProjectCommentBegin
QMR Algorithmus nach Freund und Nachtigal
ProjectCommentEnd

VersionBegin initial
...
TraceConfigBegin overall performance
...
TraceConfigEnd overall performance

VersionEnd initial

...

ProjectEnd QMR-Freund
```

Abbildung 4.4: Schema der Speicherung der Projektinformationen

Zur Verdeutlichung des Aufbaus dieser Datei ist in Abbildung 4.4 das Schema der Speicherung dargestellt. Diese Datei enthält neben den Verwaltungsversionen die Kurzkommentare und ausführlichen Kommentare. Die `.project`-Datei ist eine Abbildung der internen Darstellung eines Projektes. Daher wird erst bei der Beschreibung der internen Darstellung näher auf die Vorgehensweise bei der Speicherung und auf die gespeicherten Informationen eingegangen.

4.2.3 Das Arbeitsverzeichnis eines Projektes

Das Arbeitsverzeichnis ist aufgrund des *Work/Version*-Konzeptes der einzige Ort im Verzeichnisbaum eines Projektes, in dem der Quelltext verändert werden kann. Somit befindet sich hier die Arbeitskopie des Quelltextes.

Der Quelltext besteht aus einer Hauptdatei, deren Namen aus dem Namen des Projektes durch Anhängen des Suffix `.f` gebildet wird. In diese Datei können jedoch *Include*-Dateien mit beliebigen Namen eingebunden werden.

OPAL legt, nachdem es den Quelltext eingelesen hat, für jedes Unterprogramm des Quelltextes eine Datei an, in der die interne Darstellung der Unterprogramme binär abgespeichert wird. In einer weiteren Datei mit dem Namen `global.all` wird die interne Darstellung globaler Quelltextinformationen gesichert. Diese Dateien werden im `units`-Unterverzeichnis abgelegt. Zweck dieser Dateien ist ein beschleunigtes Einlesen des Programms durch direktes Lesen der internen Darstellung.

Im Arbeitsverzeichnis unterliegt der Quelltext ständiger Veränderung, daher wird hier nur eine Meßkonfiguration ermöglicht. Es können jedoch mit dieser Meßkonfiguration mehrere Testläufe mit verschiedenen Prozessorzahlen durchgeführt werden, so daß geprüft werden kann, ob die Archivierung des aktuellen Quelltextes als neue Quelltextversion sinnvoll ist. Das für diese Testläufe benötigte *Executable* wird bei Bedarf, d.h. falls kein *Executable* existiert oder es älter ist als der aktuelle Quelltext, beim Aufsetzen der Testläufe automatisch erzeugt und im Arbeitsverzeichnis abgelegt. Analog zur Hauptdatei des Quelltextes wird der Name des *Executable* durch Anhängen des Suffix `.exe` an den Projektnamen abgeleitet.

Verwaltungsinformationen legt `cvs` im `CVS`-Verzeichnis ab. Die Datei `Entries` ist eine, um die Versionsnummern und ggf. `tags` der zuletzt ausgelesenen Versionen erweiterte, Liste der verwalteten Quelltextdateien. Die Dateien `Root` und `Repository`

enthalten den absoluten Pfad zum verwendeten `cvs`-Repository bzw. des Unterverzeichnisses des Projektes innerhalb des `cvs`-Repository.

4.2.4 Das Eingabedatenverzeichnis eines Projektes

Neben den *Trace Data Files* und den *Executables* sind zum Teil auch die Eingabedateien beim Plattenspeicherverbrauch dominant. Da diese nur gelesen, jedoch während der gesamten Programmoptimierung nicht verändert werden, ist es ausreichend, nur eine Kopie innerhalb des Verzeichnisbaums eines Projektes abzulegen.

In jedem Verzeichnis, in dem *Executables* ausgeführt werden, existiert ein *Link* mit dem Namen `data` auf das Eingabedatenverzeichnis, um eine relative Adressierung zu ermöglichen. Dies ist notwendig, da in SVM-Fortran, wie auch in Fortran77 oder Fortran90, keine standardisierte Methode zum Lesen von UNIX-Umgebungsvariablen existiert.

4.2.5 Die Versionsverzeichnisse eines Projektes

Im Gegensatz zu den Arbeitsverzeichnissen müssen in einem Versionsverzeichnis mehrere Meßkonfigurationen koexistieren können. Durch die Anwendung des *Work/Version*-Konzeptes bei den Meßkonfigurationen existiert eine Meßkonfiguration, die analog der Arbeitskopie des Quelltextes veränderbar ist und die Basis für die archivierten, nicht veränderbaren Meßkonfigurationen bildet. Diese transiente Meßkonfiguration befindet sich im Versionsverzeichnis. Für jede archivierte Meßkonfiguration existiert im Versionsverzeichnis ein eigenes Unterverzeichnis.

Der weitere Aufbau des Versionsverzeichnis entspricht dem des Arbeitsverzeichnis, bis auf die Tatsache, daß hier der Quelltext nicht verändert werden darf. Die Dateien im `CVS`-Unterverzeichnis eines Versionsverzeichnisses enthalten lokal benötigte Informationen und werden von `cvs` unabhängig von den Dateien der `CVS`-Unterverzeichnisse, der anderen Versionsverzeichnisse und des Arbeitsverzeichnisses geändert. Ein Ersetzen dieses Verzeichnisses durch einen *Link* auf das `CVS`-Unterverzeichnis des Arbeitsverzeichnisses ist somit nicht möglich.

Eine weitere Analogie zum Arbeitsverzeichnis, das den Ort der Quelltextentwicklung darstellt, ist, daß die Versionsverzeichnisse die Orte sind, an denen eine Quelltextversion analysiert wird. Die im Anschluß beschriebenen Meßkonfigurationsverzeichnisse

dienen ausschließlich der Speicherung von Meßkonfigurationen und der zugehörigen Meßwerte.

4.2.6 Die Meßkonfigurationsverzeichnisse einer Version

Wie im vorigen Unterkapitel beschrieben, befinden sich die archivierten Meßkonfigurationen einer Quelltextversion in eigenen Unterverzeichnissen. Hier sind neben dem *Trace Request File* als Bescheibung der Meßkonfiguration auch die *Trace Data Files* abgelegt, die die ermittelten Meßergebnisse enthalten.

Es können Meßergebnisse mehrerer Testläufe mit unterschiedlichen Prozessorzahlen verwaltet werden. Die Archivierung der Ergebnisse mehrerer Testläufe mit gleicher Prozessorzahl wird jedoch nicht unterstützt.

4.3 Shell-Skripte

In der Versionsverwaltung sind sowohl Dateisystemoperationen, wie das Anlegen von Verzeichnissen und das Duplizieren von Dateien, als auch die Schnittstelle zu `cvs` durch *Shell*-Skripte realisiert. Diese Skripte werden von der Versionsverwaltung mit der ANSI C89 Standardbibliotheksfunktion `system` aufgerufen. Als Skript-Interpreter wird die Bourne Shell [47, 59] verwendet, da diese auf allen UNIX-Systemen verfügbar ist, und somit größtmögliche Portabilität gewährleistet ist. Alternativ können diese Skripte auch mit `bash` [48, 49, 50] oder `ksh` [51] ausgeführt werden, falls diese eine bessere *Performance* bieten. Neuere Skript-Sprachen wie z.B. die mit BSD UNIX eingeführte `csh` [52] oder die in Plan 9 verwendete `rc` [53] sind nicht derart verbreitet.

Im folgenden sind alle für die Versionsverwaltung entwickelten *Shell*-Skripte einzeln beschrieben.

4.3.1 opalConfigure

Aufgabe

Konfiguration der Versionsverwaltung

Syntax

```
opalConfigure
```

Parameter

keine

Beschreibung

Das *Shell*-Skript `opalConfigure` ist ein interaktives Hilfsprogramm zur Konfiguration der Versionsverwaltung. Aus den Benutzereingaben wird

- die Verzeichnisstruktur generiert,
- die *Links* mit dem Namen `.opal` im *Home*-Verzeichnis der *Workstation* auf der OPAL abläuft und dem Parallelrechner erzeugt und auf die Basis der Verzeichnisstruktur gesetzt,
- die Musterdateien in das *Template*-Verzeichnis kopiert,
- die Datei `Makefile.def` in das Basisverzeichnis der Verzeichnisstruktur kopiert,
- das *cvs* Repository ggf. erzeugt und initialisiert.

4.3.2 opalCreateProject

Aufgabe

Anlegen eines neuen Projektes

Syntax

`opalCreateProject` *project*

Parameter

Name	Bedeutung
project	Name des neuen Projektes

Beschreibung

Das *Shell*-Skript `opalCreateProject` legt ein neues Projekt innerhalb der Verzeichnisstruktur der Versionsverwaltung an. Hierzu werden folgende Schritte ausgeführt:

- Anlegen des Projektverzeichnisses,
- Anlegen des Arbeits- und des Eingabedatenverzeichnisses im neu erstellten Projektverzeichnis,
- Anlegen der Unterverzeichnisse `CVS/` und `units/` sowie des *Links* `data` im neuen Arbeitsverzeichnis,

- Kopieren des Muster-*Makefile* und der „Formblätter“ für den Quelltext-versions- und Meßkonfigurationskommentar in das Arbeitsverzeichnis.

4.3.3 opalCreateVersion

Aufgabe

Anlegen einer neuen Quelltextversion

Syntax

- Archivierung des Prototyps als initiale Version:
  `opalCreateVersion -p` *project* `-r 1`
- Anlegen des ersten Nachfolgers einer Version:
  `opalCreateVersion -p` *project* `-r` *release* `-m` *message*
- Anlegen eines weiteren Nachfolgers zu einer Version:
  `opalCreateVersion -p` *project* `-b` *branch* `-r` *release* `-m` *message*

Parameter

Name	Bedeutung
project	Name des Projektes
release	`cvs`-Tag der Quelltextversion
message	Kurzkommentar der Quelltextversion

Beschreibung

Das *Shell*-Skript `opalCreateVersion` legt eine neue Quelltextversion innerhalb eines Projektes an. Hierbei können drei Fälle auftreten, die sich in der Behandlung durch `cvs` unterscheiden.

1. Archivierung des Prototyps als initiale Version. Hierzu muß der Quelltext in das `cvs`-Archiv importiert werden. Daher wird
   - im Projektverzeichnis ein *Link* mit dem Projektnamen erzeugt, der auf das Arbeisverzeichnis zeigt,
   - mit `cvs import` der Quelltext in das `cvs`-Archiv importiert,
   - nach dem Importieren gelöscht und
   - mit `cvs checkout` aus dem `cvs`-Archiv ausgelesen. Hierbei legt `cvs` im Arbeitsverzeichnis das `CVS`-Verzeichnis an.
   - Der Import ist damit abgeschlossen und der zuvor hierzu angelegte *Link* wird gelöscht.

2. Zum Anlegen des ersten Nachfolgers einer Version wird der Quelltext mit `cvs commit` in das `cvs`-Archiv kopiert.
3. Vor dem Anlegen eines weiteren Nachfolgers einer Version muß der neue alternative Entwicklungspfad `cvs` mitgeteilt werden. Hierzu wird
   - mit `cvs update` dem `cvs`-System der zu verwendende Entwicklungspfad mitgeteilt.
   - Danach wird der Quelltext mit `cvs commit` in das `cvs`-Archiv kopiert.

Die folgenden, nicht durch `cvs` ausgeführten Schritte sind für alle drei Fälle gleich. Es folgt das

- Erzeugen des Unterverzeichnisses für die neue Quelltextversion,
- Anlegen der Unterverzeichnisse `CVS/` und `units/` sowie des *Links* `data` im neuen Versionsverzeichnis,
- Löschen des *Trace Request File* und aller *Trace Data Files* im Arbeitsverzeichnis, falls das *Trace Request File* älter als der Quelltext ist.
- Falls notwendig: Löschen aller *Trace Data Files* im Arbeitsverzeichnis, die älter als das zugehörige `Trace Request File` sind.
- Falls notwendig: Kopieren der *Trace Data Files* und des `Trace Request File` vom Arbeitsverzeichnis in das Unterverzeichnis der neuen Quelltextversion.

4.3.4 opalRetrieveVersion

Aufgabe

Auslesen einer Quelltextversion aus dem `cvs`-Archiv

Syntax

- Auslesen einer Version ohne Nachfolger aus dem Hauptzweig
  `opalRetrieveVersion -d` *dest* `-p` *project* `-r` *release*
- Auslesen einer Version ohne Nachfolger eines Nebenzweigs
  `opalRetrieveVersion -d` *dest* `-p` *project* `-r` *release* `-b` *branch*
- Auslesen einer Version mit Nachfolger
  `opalRetrieveVersion -d` *dest* `-p` *project* `-r` *release* `-nb` *new_branch*

Parameter

Name	Bedeutung
dest	`work` oder `version`
project	Name des Projektes
release	`cvs`-Tag der Quelltextversion
branch	`cvs`-Tag des Zweigs
new_branch	`cvs`-Tag des neu anzulegenden Zweigs

Beschreibung

Das *Shell*-Skript `opalRetrieveVersion` kopiert eine Quelltextversion aus einem Versionsverzeichnis in das Arbeitsverzeichnis. Weitgehend analog zum Skript `opalCreateVersion` müssen hierzu für `cvs` drei Fälle unterschieden werden.

1. Zum Auslesen einer Version ohne Nachfolger aus dem Hauptzweig wird diese mit `cvs update -r` *release* aus dem `cvs`-Archiv in das Arbeitsverzeichnis geladen.
2. Das Auslesen einer Version ohne Nachfolger eines Nebenzweigs erfordert zudem ein Setzen des alternativen Entwicklungszweigs durch `cvs update -r` *branch*.
3. Beim Auslesen einer Version mit Nachfolger wird nach dem Kopieren der gewünschten Quelltextversion mit `cvs update -r` *release*
   - mit `cvs tag` der neue *Tag* angelegt und
   - mit `cvs update -r` *new_branch* in die im `CVS`-Unterverzeichnis des Arbeitsverzeichnisses abgelegten lokalen Versionsinformationen eingetragen.

Im Anschluß werden das *Trace Request File* und die *Trace Data Files* im Arbeitsverzeichnis gelöscht.

4.3.5 opalCreateTraceConfig

Aufgabe

Anlegen einer neuen Meßkonfiguration

Syntax

`opalCreateTraceConfig -p` *project* `-r` *release* `-t` *trace*

Parameter

Name	Bedeutung
project	Name des Projektes
release	`cvs`-Tag der Quelltextversion
trace	Nummer der Meßkonfiguration

Beschreibung

Das *Shell*-Skript `opalCreateTraceConfig` erzeugt eine neue Meßkonfiguration für eine Quelltextversion. Hierzu werden die folgenden Schritte ausgeführt:

- Löschen aller `Trace Data Files` im Verzeichnis der Quelltextversion, die älter als das zugehörige `Trace Request File` sind,
- Erzeugen des Unterverzeichnisses für die neue Meßkonfiguration im Verzeichnis der Quelltextversion,
- Kopieren der `Trace Data Files` und des `Trace Request File` vom Verzeichnis der Quelltextversion in das Unterverzeichnis der neuen Meßkonfiguration,
- Anlegen eines *Link* im Verzeichnis der neuen Meßkonfiguration auf das `units`-Verzeichnis des Versionsverzeichnisses.
- Anlegen eines *Link* im Verzeichnis der neuen Meßkonfiguration auf das Eingabedatenverzeichnis.

4.3.6 opalRetrieveTraceConfig

Aufgabe

Zugriff auf eine archivierte Meßkonfiguration

Syntax

`opalRetrieveTraceConfig -p` *project* `-r` *release* `-t` *trace*

Parameter

Name	Bedeutung
project	Name des Projektes
release	`cvs`-Tag der Quelltextversion
trace	Nummer der Meßkonfiguration

Beschreibung

Das *Shell*-Skript `opalRetrieveTraceConfig` kopiert eine Meßkonfiguration

aus einem Meßkonfigurationsverzeichnis in das Versionsverzeichnis. Hierzu werden die folgenden Schritte ausgeführt:

- die Meßkonfiguration im Versionsverzeichnis wird gelöscht,
- die spezifizierte Meßkonfiguration wird aus dem Meßkonfigurationsverzeichnis in das Versionsverzeichnis kopiert.

4.3.7 opalLaunchApplication

Aufgabe

Starten eines Testlaufs auf dem Parallelrechner

Syntax

- Programmstart ohne *Tracing*
  `opalLaunchApplication -d` *dest* `-p` *project* `-r` *release* `-n` *#proc*
- Programmstart mit *Tracing*
  `opalLaunchApplication -t -d` *dest* `-p` *project* `-r` *release* `-n` *#proc*

Parameter

Name	Bedeutung
dest	`work` oder `version`
project	Name des Projektes
release	`cvs`-Tag der Quelltextversion
#proc	Anzahl der zu verwendenen Prozessoren

Beschreibung

Das *Shell*-Skript `opalLaunchApplication` startet einen Testlauf mit der angegebenen Anzahl Prozessoren auf dem Parallelrechner. Mit der `-d` Option wird spezifiziert, ob das *Executable* im Arbeitsverzeichnis (`work`), oder in einem der Versionsverzeichnisse (`version`) ausgeführt wird. Falls das *Executable* einer Quelltextversion ausgeführt werden soll, wird diese falls notwendig aus dem `cvs`-Archiv zuvor ausgelesen. Ebenfalls wird überprüft, ob ein aktuelles *Executable* existiert und ggf. der Quelltext übersetzt. Mit der Option `-t` wird das *Tracing* eingeschaltet. Ein Sonderfall ist das Starten von Testläufen vor dem Anlegen der initialen Version. Hierzu muß als Quelltextversion das Schlüsselwort `prototype` angegeben werden.

4.4 Interne Darstellung eines Projektes

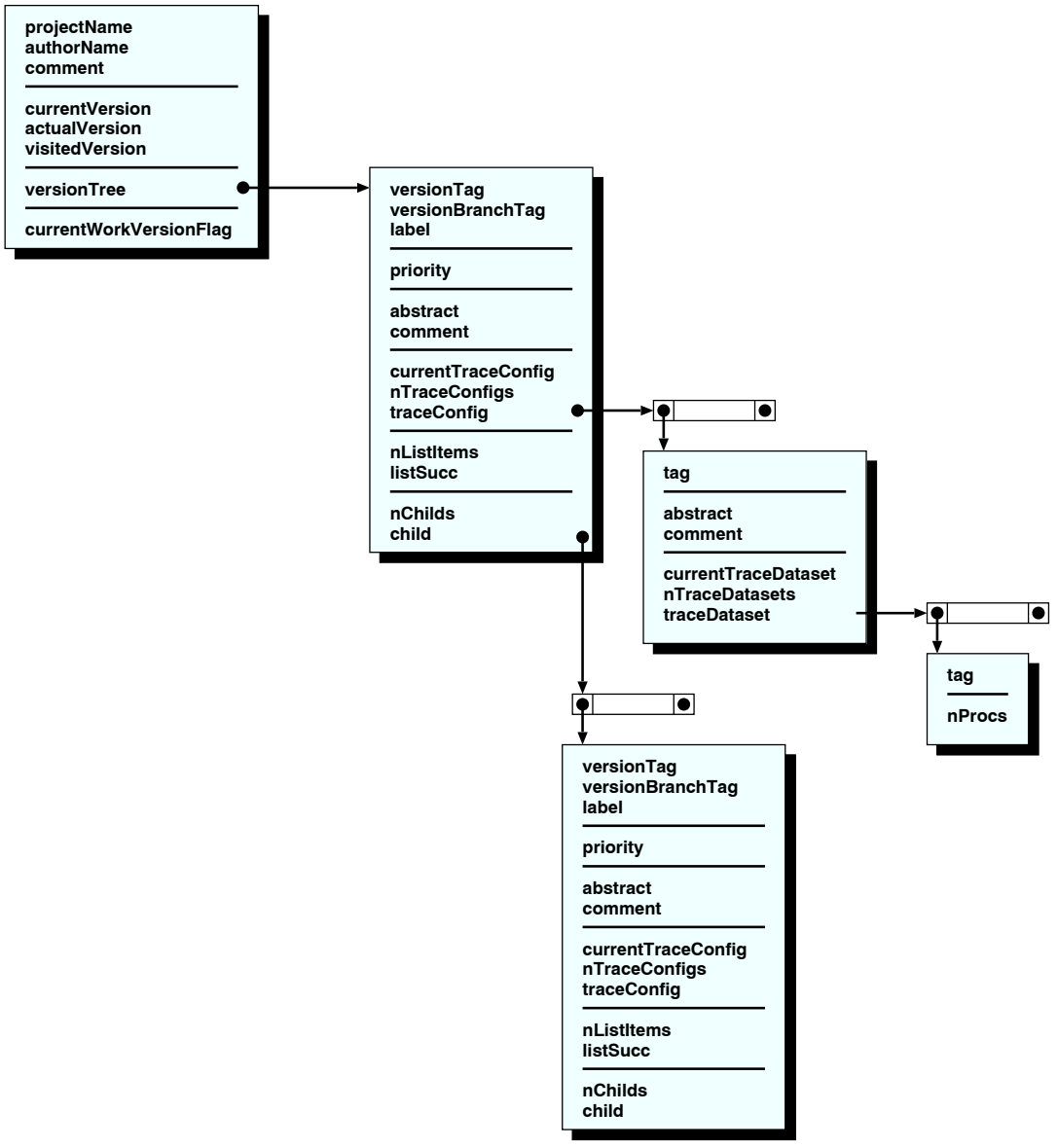


Abbildung 4.5: Interne Darstellung des Versionsbaums

Die Versionsverwaltung verwendet zur internen Darstellung eines Projektes getrennte Datenstrukturen zur Abbildung von Projekt-, Quelltextversions-, Meßkonfigurations- und Testlaufinformationen. Diese sind zu einer Baumstruktur (vgl. Abbil-

dung 4.5), in der die Projektdatenstruktur die Wurzel bildet, verbunden. Anders als bei der Verzeichnisstruktur, bei der alle Versionsverzeichnisse auf einer Ebene liegen, wird bei der internen Darstellung die Topologie des Versionsbaums nachgebildet.

Bei der Vorstellung der `.project`-Datei wurde diese bereits als Abbildung der internen Darstellung auf eine Datei beschrieben. Wie eine `.project`-Datei (vgl. Abbildung 4.4) enthält die interne Darstellung jeweils nur ein Projekt, so daß vor dem Wechsel zwischen Projekten innerhalb der Versionsverwaltung, wie vor dem Beenden, die interne Darstellung des geladenen Projektes in die korrespondierende `.project`-Datei gesichert werden muß.

Beim Sichern wird zunächst die Projektdatenstruktur gespeichert. Anschließend wird die interne Darstellung des Versionsbaums rekursiv durchlaufen. Hierbei wird für jede Quelltextversion der zugehörigen Version die Versionsdatenstruktur, gefolgt von den Meßkonfigurationsstrukturen, abgelegt. An jede Meßkonfigurationsstruktur werden die Testlaufdatenstrukturen angehangen. Durch dieses geschachtelte Speicherschema ist die hierarchische Abhängigkeit der Daten durch ihre relative Position wiedergegeben. Eine weitere Verbesserung der Lesbarkeit wird durch Schlüsselwörter wie z.B. `ProjectAuthorName` oder `VersionNumOfTraceConfigs`, Umrahmen der Blöcke (Datenstrukturen) mit `Begin`- und `End`-Klammern sowie durch Einfügen von Leerzeilen erreicht. Die Schlüsselwörter werden durch Anfügen des Feldnamens der entsprechenden Datenstruktur an den Präfix `Project`, `Version`, `TraceConfig` oder `TraceDataset` gebildet. Diese Präfixe werden auch den `Begin`- und `End`-Klammern vorangestellt. Damit kann ein Ausdruck der `.project`-Datei als Dokumentation des Projektstatus verwendet werden.

Zum Einlesen der `.project`-Datei wurde zunächst ein mit `lex` und `yacc` [58] generierter *Parser* in Betracht gezogen. Diese Idee wurde verworfen, da dieser Ansatz nicht mit der später im Unterkapitel „Anmerkungen zur Implementierung“ beschriebenen Strukturierung des Quelltextes vereinbar war.

4.4.1 Die Projektdatenstruktur

Die interne Darstellung des Versionsbaums gibt die Hierarchie von Projekt-, Versions- und Meßkonfigurationsverzeichnis direkt wieder. Als Information über das Arbeitsverzeichnis ist nur die Quelltextversion, auf der die Arbeitskopie des Quelltextes basiert, zu speichern. Sie ist daher als Zeiger auf eine Versionsdatenstruktur im Feld

`actualVersion` abgelegt.

Obwohl sich die zuletzt bzw. momentan angezeigte Quelltextversion `currentVersion` aus der `actualVersion` und der zuletzt besuchten Version `visitedVersion` und dem Zustand `currentWorkVersionFlag` ergibt, wird diese Information mitgeführt, um den Zugriff zu beschleunigen.

Der Zeiger `versionTree` verweist auf die Versionsdatenstruktur der initialen Version, der Wurzel des Versionsbaums.

```
struct project_struct
{
  char *projectName;
  char *authorName;
  char *comment;

  version_t currentVersion;
  version_t actualVersion;
  version_t visitedVersion;

  version_t versionTree;

  work_version_flag_t currentWorkVersionFlag;
}
```

Abbildung 4.6: Die Projektdatenstruktur

4.4.2 Die Versionsdatenstruktur

Die ersten beiden Felder `versionTag` und `versionBranchTag` enthalten die zur Kennzeichnung der Quelltextversion bzw. zur Vorbereitung eines neuen Zweigs verwendeten `cvs`-*Tags*. Mit dem Feld `label`, das die Beschriftung der *Buttons* enthält, die die Quelltextversionen repräsentieren, sind in der ersten Gruppe alle Informationen zur Benennung und zum Zugriff auf die Quelltextversionen zusammengefaßt. Alle drei Felder sind redundant und dienen ausschließlich der Beschleunigung des Zugriffs und der Protokollierung der `cvs`-*Tags* innerhalb der `.project`-Datei.

```
struct version_struct
{
  char *versionTag;
  char *versionBranchTag;
  char *label;

  int priority;

  char *abstract;
  char *comment;

  trace_config_t currentTraceConfig;
  int nTraceConfigs;
  trace_config_t *traceConfig;

  int nListItems;
  version_t listSucc;

  int nChilds;
  version_t *child;
};
```

Abbildung 4.7: Die Versionsdatenstruktur

Das Feld `priority` wird zur Zeit nicht verwendet. Es ist vorgesehen zur Implementation von Quelltextversion-Prioritäten zum Ausblenden wenig erfolgversprechender Pfade im Versionsbaum. Dieses Feld wird bei der Initialisierung der Datenstruktur mit 0 belegt und in der `.project`-Datei ausgegeben, um bei einer Erweiterung das Dateiformat beibehalten zu können.

In der folgenden Gruppe mit dem Feld `abstract` wird der Kurzkommentar und mit `comment` der ausführliche Kommentar referenziert.

Der Zeiger `currentTraceConfig` verweist auf die Datenstruktur der angezeigten Meßkonfiguration. Mit dem Zeigerfeld `traceConfig`, dessen `nTraceConfigs` Elemente die Datenstrukturen der für Quelltextversionen angelegten Meßkonfigurationen adressieren, sind in dieser Gruppe alle zur Verwaltung der Meßkonfigurationen notwendigen Informationen zusammengefaßt.

Möchte der Anwender eine Quelltextversion besuchen oder als Basis seines weiteren Vorgehens bei der Optimierung verwenden, wird ihm zur Spezifikation der Version neben einer graphischen Darstellung des Versionsbaums eine Versionsliste angezeigt. Diese Liste enthält zunächst die zuletzt besuchte Quelltextversion und alle hierauf basierenden Versionen. Wird ein *Button* im Versionsbaum gedrückt, bildet die hierdurch gewählte Version den Kopf der Liste. Die Felder `nListItems` und `listSucc` enthalten Informationen zum schnellen Aufbau dieser Liste.

Nach dem Einlesen der `.project`-Datei und Aufbau des Versionsbaums wird dieser rekursiv nach dem in Abbildung 4.8 gezeigten Schema durchlaufen. Hierbei wird im Feld `listSucc` der Datenstruktur der zuvor „besuchten“ Version die Adresse der Datenstruktur der momentan „besuchten“ Version eingetragen und so die Versionen zur einer Liste verkettet.

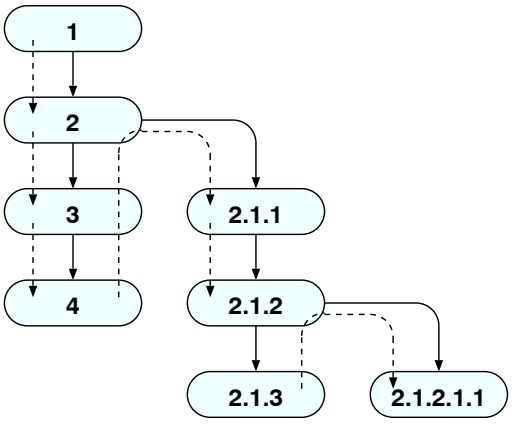


Abbildung 4.8: Verkettung der Versionen zu einer Auswahlliste

Die Anzahl der Listenelemente, die angezeigt werden, falls die korrespondierende Version den Kopf der Liste bildet, wird im Feld `nListItems` gespeichert. Bildet eine Quelltextversion, die ein Blatt des Versionsbaums ist, den Kopf der Auswahlliste, so hat diese Liste ein Element. Die Anzahl der Listenelemente, wenn eine „innere“ Version den Kopf der Liste bildet, ist die Summe der Listenelemente, die ihre direkten Nachfolger benötigen, erhöht um eins, da für die Version selbst ein weiteres Listenelement benötigt wird. Daher übergibt jede Version, wie in Abbildung 4.9 verdeutlicht, ihren `nListItems`-Wert an ihren Vorgänger. Hier werden diese Rück-

gabewerte summiert, um eins erhöht, im Feld `nListItems` abgelegt und wiederum übergeben.

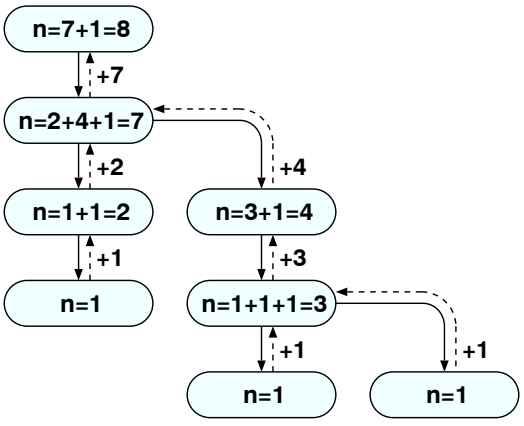


Abbildung 4.9: Ermittlung der Anzahl der Listenelemente

Zur Anzeige der Auswahlliste als Motif-*Widget* werden, ausgehend von der gewählten Version, die Kurzkommentare der `nListItems` Listenelemente in eine Motif-Datenstruktur vom Typ `XmStringTable` kopiert.

4.4.3 Die Meßkonfigurationsdatenstruktur

```
struct trace_config_struct
{   int tag;

  char *abstract;
  char *comment;

  trace_dataset_t currentTraceDataset;
  int nTraceDatasets;
  trace_dataset_t *traceDataset;
};
```

Abbildung 4.10: Die Meßkonfigurationsdatenstruktur

Die Felder der Meßkonfigurationsdatenstruktur entsprechen den korrespondierenden Felder der Versionsdatenstruktur. Das *Tag* `tag` enthält die Position des Zei-

gers im Zeigerfeld `traceConfig` der Datenstuktur der zugrundeliegenden Quelltextversion, die die Meßkonfigurationsdatenstruktur adressiert.

4.4.4 Die Testlaufdatenstruktur

Die Testlaufdatenstruktur enthält neben dem, analog dem in der Meßkonfigurationsdatenstruktur verwendeten, *Tag* `tag`, nur noch die im Feld `nProcs` gespeicherte Anzahl der für den Testlauf angeforderten Prozessoren.

```
struct trace_dataset_struct
{
  int tag;

  int nProcs;
};
```

Abbildung 4.11: Die Testlaufdatenstruktur

4.5 Anmerkungen zur Implementierung

Obwohl OPAL im ANSI C++ Standard implementiert ist, wurde für die Versionsverwaltung ausschließlich der in ANSI C++ als Untermenge enthaltene ANSI C89 Standard [56] verwendet. Der Grund hierfür ist die bessere Wiederverwendbarkeit des Quelltextes in anderen Analysewerkzeugen. So ist z.B. auch das im Zentralinstitut für Angewandte Mathematik des Forschungszentrum Jülich entwickelte Werkzeug PARvis zur Analyse von *Message-Passing*-Programmen in ANSI C89 geschrieben. Für eine Anbindung von in Klassen eingebetteten ANSI C++ Funktionen müßten *Wrapper* in diesem geschrieben werden.

Dennoch wurden Prinzipien der objektorientierten Programmentwicklung angewendet. Datenstrukturen und Funktionen bilden Einheiten, die in Klassen eingebettet werden können. Für die Datenstrukturen der internen Darstellung eines Projektes existieren jeweils eigene Funktionen zum Erzeugen (*constructor*), Sichern und Abbauen (*destructor*) durch Freigeben des allokierten Speichers. Diese werden im Sprachgebrauch der objektorientierten Programmentwicklung Methoden genannt.

Weiterhin sind die Verwaltungsunktionen für Projekte, Versionen, Meßkonfigurationen und Testläufe gruppiert. Durch die verwendete globale Namenskonvention werden gleichartige Funktionen verschiedener Gruppen analog benannt. So sind `cbEditProjectComment`, `cbEditVersionComment` und `cbEditTraceConfigComment` die Funktionen zum Ändern der Kommentare von Projekten, Versionen bzw. Meßkonfigurationen.

In auf Motif [57] basierenden Bedienoberflächen werden *Widgets Callback*-Funktionen zugeordnet, die bei bestimmten Ereignissen (z.B. Mausaktionen) ausgeführt werden, um die mit dem *Widget* verbundene Funktionalität zu realisieren. In OPAL werden die *Callback*-Funktionen über globale Variablen mit den notwendigen Parametern versorgt. In der Versionsverwaltung wird hingegen ein anderer Mechanismus, der dem in PARvis verwendeten ähnlich ist, angewendet. Bei der Zuordnung einer *Callback*-Funktion zu einem *Widget* kann ein Zeiger auf eine Datenstruktur angegeben werden, die der *Callback*-Funktion beim Aufruf übergeben wird. Die Versionsverwaltung nutzt diese Möglichkeit, den *Callback*-Funktionen Parameter zu übergeben. Hierbei ist jeder *Callback*-Funktion, die Parameter benötigt, eine eigene Datenstruktur zugeordnet. Diese werden zur Laufzeit beim Erzeugen eines *Callback* angelegt (*constructor*) und nach dem Löschen des zugeordneten *Widget* selbst gelöscht (*destructor*). Im Gegensatz zur Implementierung in PARvis, in der die *Widgets* über globale Variablen referenziert werden, sind in der Versionsverwaltung die Zeiger auf die *Widgets* in den, den *Callback*-Funktionen übergebenen, Datenstrukturen enthalten.

Hauptvorteil dieser Methode gegenüber der Verwendung globaler Variablen ist die vereinfachte Wiederverwendbarkeit durch Kapselung des Codes und der Daten einer Funktionalität in eine geschlossene Einheit. Dieser Vorteil wird jedoch durch die zum Anlegen und Löschen der Datenstrukturen benötigte Prozessorzeit erkauft.

Ein weiterer Vorteil ist, daß mehrere Instanzen einer *Callback*-Funktion koexistieren können, d.h. die *Callback*-Funktionen sind reentrant. Somit entfällt die Notwendigkeit, einen Schutzmechanismus, der einen mehrfachen Aufruf verhindert, zu implementieren.

Die Beschränkung auf eine Hauptquelltextdatei, in der *Includes* eingebunden sein können, ist in anderen Anwendungen evtl. nicht haltbar. Diese wirkt sich auf das *Makefile*, sowie auf die Skripte `opalCreateVersion` und `opalRetrieveVersion` aus.

Das *Makefile* ist speziell auf SVM-Fortran zugeschnitten und ist daher nicht für andere Anwendungen zu übernehmen. Die Skripte sind jedoch gegebenenfalls entspechend zu ändern.

Kapitel 5

Die Versionsverwaltung in der Anwendung

Nach einem kurzen Rückblick auf die Optimierung des iterativen Gleichungssystemlösers und einer Zusammenfassung der Auswirkungen des *Work/Version*-Konzeptes auf die Bedienung der Versionsverwaltung, wird in vier weiteren Unterkapiteln eine Einführung in die Anwendung der Versionverwaltung mit dem Gleichungssystemlöser als Beispiel gegeben. Zunächst wird dabei auf die Initialisierung der Versionsverwaltung und das Anlegen von Projekten eingegangen. Im darauffolgenden Unterkapitel wird das Erstellen der initialen Quelltextversion beschrieben und zum Schluß das Vorgehen bei der Parallelisierung und Optimierung der Anwendung dargestellt.

5.1 Rückblick auf die Optimierung des iterativen Gleichungssystemlösers

Bei der Optimierung des iterativen Gleichungssystemlösers war die Entwicklung neuer Quelltextversionen nicht konsekutiv. In zwei Fällen wurde auf eine ältere Quelltextversion zurückgegriffen. Zur Veranschaulichung ist daher in Abbildung 5.1 die zeitliche Abfolge der Entwicklung und die Anordnung der Quelltextversionen im Versionsbaum dargestellt.

Ausgehend von einer sequentiellen Implementierung des Gleichungssystemlösers wurde die erste parallele Version erstellt. Hierbei wurden parallel auszuführende Schleifen und Reduktionen durch Direktiven gekennzeichnet. Die erreichte *Performance* war hierbei noch unwichtig. Das Ziel dieses Schrittes war es, eine parallele Version

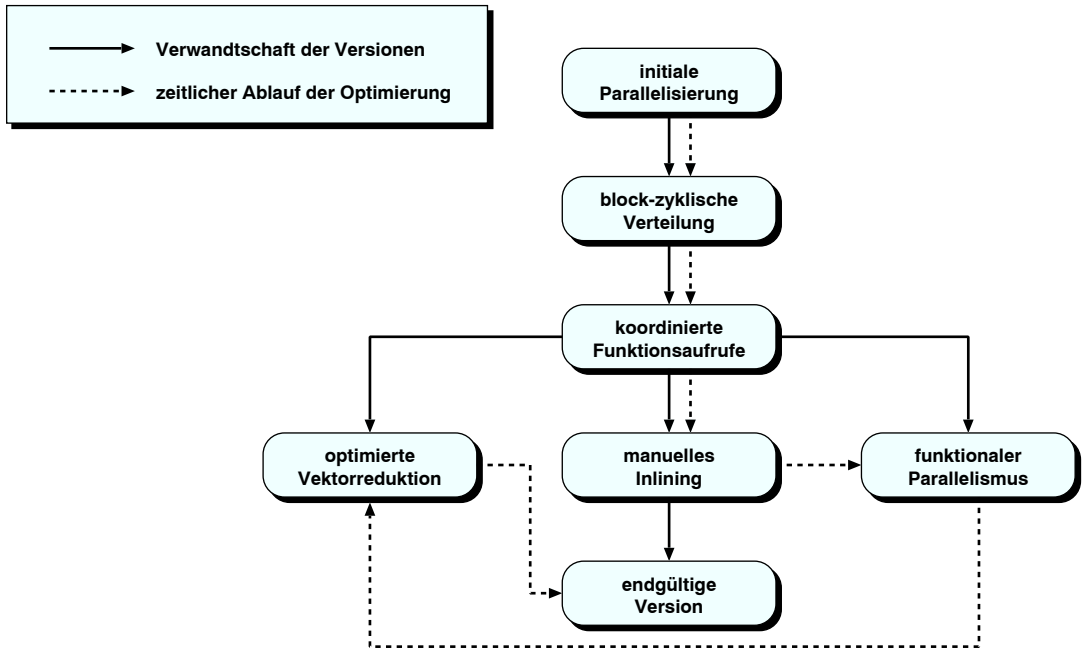


Abbildung 5.1: Nicht konsekutive Entwicklung der Versionen

des Programms zu erstellen, die korrekte Ergebnisse liefert.

Im zweiten Schritt wurde durch das Ausrichten der Vektoren auf Seitengrenzen und Anwendung einer block-zyklischen Arbeitsverteilung *False Sharing* reduziert und damit der erste Schritt zur *Performance*-Optimierung durchgeführt.

In zwei weiteren Schritten wurde zunächst durch das Einführen koordinierter Funktionsaufrufe (*coordinated calls*) und dann durch manuelles *Inlining* sowie weiteren lokalen Optimierungen die *Performance* weiter verbessert.

Die Nutzung des im verwendeten Algorithmus enthaltenen funktionalen Parallelismus wäre in der Version mit manuellem *Inlining* nur unter hohem Aufwand zu realisieren gewesen. Daher wurde auf die vorletzte Version zurückgegriffen und ein neuer Entwicklungspfad erzeugt. Wie bereits beschrieben führte dieser Weg aufgrund schlechter Lastbalance nicht zur erwarteten Reduzierung der Ausführungszeit und wurde deswegen nicht weiterverfolgt.

Ein gutes Skalieren des Programms wurde immer noch durch die nahezu von der

Prozessorzahl unabhängige Ausführungszeit des Produktes mit der transponierten Matrix verhindert. Um dieses Problem zu lösen, wurde ein zweites mal auf die letzte Version des Hauptzweigs ohne manuellem *Inlining* zurückgegriffen. Aufgrund der Existenz der Unterprogramme blieben hier die Auswirkungen der notwendigen Änderungen im Unterprogramm `TMatVec` lokal.

Die im vorigen Schritt erarbeiteten Optimierungen wurden im Hauptzweig, durch Einfügen in die Version mit manuellem *Inlining*, übernommen. Mit diesem Schritt wurde die Parallelisierung und Optimierung des Gleichungssystemlösers beendet.

5.2 Auswirkungen des Work/Version-Konzeptes auf die Bedienung

Die grundlegende Idee des *Work/Version*-Konzeptes besteht darin, daß nur eine laufend weiterentwickelte Arbeitskopie des Quelltextes eines Projektes existiert. Momentaufnahmen dieser Arbeitskopie werden als Quelltextversionen des Projektes archiviert. Um die Integrität zwischen Quelltext und Meßergebnissen zu wahren, sowie um nachträgliche Messungen zu ermöglichen, sind archivierte Quelltexte nicht veränderbar.

Als Folge des *Work/Version*-Konzeptes befindet sich die Versionsverwaltung entweder im Zustand `Work` oder im Zustand `Version`. Alle Aktionen, die auf der Arbeitskopie des Quelltextes basieren, können nur im Zustand `Work` ausgeführt werden. Für die Messungen an einem archivierten Quelltext muß sich die Versionsverwaltung entsprechend im Zustand `Version` befinden.

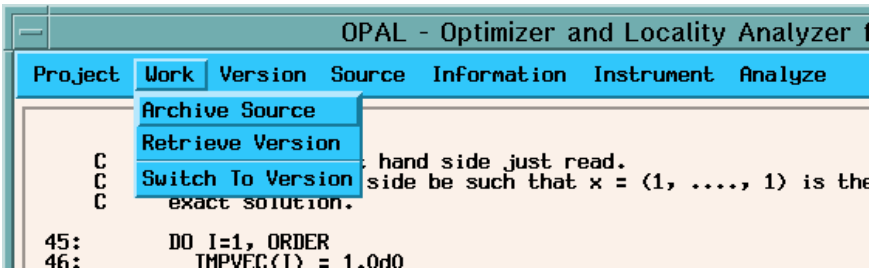


Abbildung 5.2: Das Work-Menü

Die Trennung zwischen `Work` und `Version` wirkt sich auch auf die Menü-Struktur aus. Im `Work`-Menü (siehe Abbildung 5.2) sind die Funktionen der Versionsverwaltung, die die Arbeitskopie des Quelltextes betreffen, zusammengefaßt.

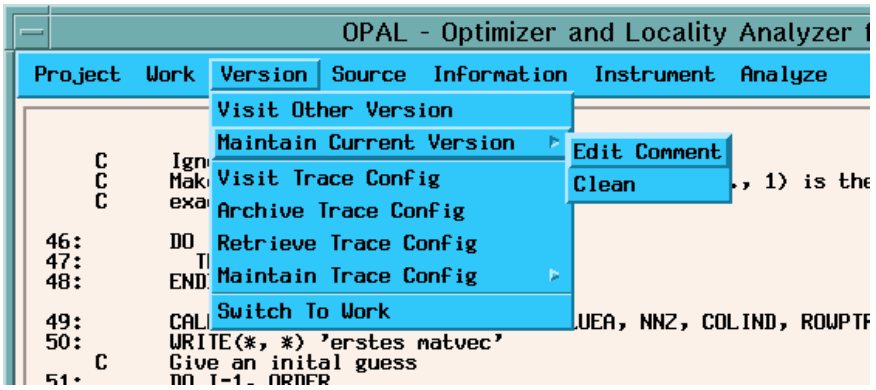


Abbildung 5.3: Das Version-Menü

Entsprechend sind im in Abbildung 5.3 gezeigten `Version`-Menü die Funktionen zum Arbeiten mit den Quelltextversionen enthalten.

5.3 Initialisierung der Versionsverwaltung

Vor der Initialisierung der Versionsverwaltung mit dem Skript `opalConfigure` muß sowohl die *Workstation* als auch der Parallelrechner manuell auf die Versionsverwaltung vorbereitet werden.

Um einen Zugriff auf den Parallelrechner (*SVM Host*) über eine *Remote-Shell* zu ermöglichen, muß in der `.rhosts`-Datei im *Home*-Verzeichnis des Entwicklers auf dem Parallelrechner die Adresse der *Workstation*, auf der OPAL ablaufen soll, und der korrespondierende *Login*-Name eingetragen werden. Eine weitere notwendige Vorarbeit ist die Einbindung des Dateisystems der *Workstation* in das Dateisystem des Parallelrechners. In der zur Entwicklung und zum Testen der Versionsverwaltung verwendeten Rechnerkonfiguration wird dies durch das *Network File System* (`NFS`) realisiert. Dieser Schritt benötigt `root`-Rechte sowohl auf dem Parallelrechner als auch auf der *Workstation* und ist daher von den jeweiligen Administratoren durchzuführen. Die Versionsverwaltung verwendet die Umgebungsvariablen `EDITOR` und `HOME`, die auf der *Workstation* mit gültigen Werten belegt sein müssen. Die Variable

`EDITOR` spezifiziert den zu verwendenden externen *Editor*. Die Variable `HOME` enthält den absoluten Pfad des *Home*-Verzeichnisses und wird zur Bildung der absoluten Pfade zu den Dateien innerhalb der Verzeichnisstruktur notwendig.

Die weiteren Arbeitsschritte werden vom *Shell*-Skript `opalConfigure` durchgeführt. Im folgenden Dialog werden die Konfigurationsdetails abgefragt.

„please enter base directory (absolut path)“
: Der Anwender wird gebeten, den absoluten Pfad zum Basisverzeichnis der Verzeichnisstruktur der Versionverwaltung einzugeben. Existiert das angegebene Verzeichnis, oder kann es nicht angelegt werden, wird die Abfrage nach Anzeige einer Fehlermeldung wiederholt. Als *Default* wird `$HOME/opal` vorgeschlagen.

„please enter repository directory (absolut path)“
: Hiermit wird analog zur vorigen Abfrage der absolute Pfad zum Basisverzeichnis des `cvs`-*Repository* erfragt. Der *Default* ist hier ein Unterverzeichnis mit dem Namen `repository` im Basisverzeichnis der Verzeichnisstruktur der Versionverwaltung. Bei dieser Abfrage wird festgelegt, ob ein bereits existierendes Archiv verwendet oder ein neues Archiv angelegt werden soll.

„please enter SVM host“
: Die Adresse des Parallelrechners (*SVM host*) muß bei dieser Frage eingegeben werden. Als *Default* wird hier ein im Skript festgelegter Rechner verwendet. Mit dem UNIX-Kommando `ping` wird die Existenz des Rechners überprüft. Falls der Rechner nicht gefunden wurde, wird die Abfrage nach Anzeige einer Fehlermeldung wiederholt.

„please enter SVM host login“
: Hierbei wird der beim Zugriff auf den Parallelrechner zu verwendene *Login*-Name erfragt. Durch Absetzen eines `echo`-Befehls auf dem Parallelrechner und anschließender Analyse der Ausgabe wird die Zugriffsmöglichkeit überprüft. Falls kein Zugriff möglich ist, wird nach Anzeige einer Fehlermeldung die Abfrage wiederholt. Der *Default* ist der *Login*-Name auf der *Workstation*.

„please enter SVM host's access path to the project directory“
: Hiermit wird der Zugriffspfad des Parallelrechners auf die im Dateisystem der *Workstation* abgelegte Verzeichnisstruktur der Versionsverwaltung erfragt. Auch in diesem Fall wird die Eingabe überprüft und ggf. die Abfrage wiederholt. Der *Default* ist von der Verzeichnisstruktur auf der *Workstation* abhängig

und entspricht den Konventionen, die für die vom Zentralinstitut für Angewandte Mathematik betriebenen Rechner gelten.

5.4 Anlegen von Projekten

Das Anlegen von neuen Projekten wird innerhalb der Analyse-Umgebung durchgeführt. Mit dem Unterpunkt `New Project` des `Project`-Menüs wird die, in Abbildung 5.4 dargestellte, Eingabemaske angezeigt, in der der Name des Projektes, der Name des Autors und eine Beschreibung des Projekts eingegeben werden. Diesen Kommentar kann der Anwender später mit dem Menüpunkt `Project` → `Maintain Project` → `Edit Comment` ändern oder erweitern.

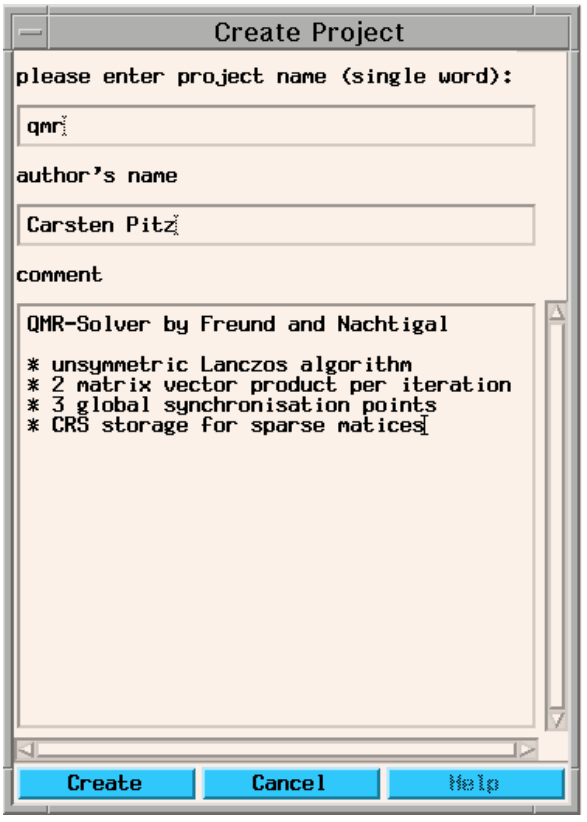


Abbildung 5.4: Dialogfenster zum Anlegen eines neuen Projektes

Auf Basis dieser Informationen legt die Versionsverwaltung, nachdem die Eingaben durch Drücken des `Create`-*Buttons* bestätigt wurden, die Projektverzeichnisstruktur

durch Aufruf des *Shell*-Skriptes `opalCreateProject` an.

Wenn das neue Projekt angelegt ist, werden die Statusfelder unterhalb des Quelltext-*Widget* (siehe Abbildung 5.5) aktualisiert. Im ersten Feld wird der Projektname und im zweiten, das den angezeigten Quelltext spezifizierende Feld, das Schlüsselwort `work` eingetragen. Das dritte Feld spezifiziert das verwendete *Trace Request File*. Durch das Schlüsselwort `editable` wird angegeben, daß das *Trace Request File* änderbar ist. Das vierte Feld gibt die Anzahl der im Testlauf verwendeten Prozessoren an und bleibt leer.

Abbildung 5.5: Die Statusfelder nach Anlegen eines Projektes

Danach werden die Quelltexte im Arbeitsverzeichnis und die Eingabedateien im Eingabedatenverzeichnis abgelegt. Dieser Arbeitsschritt ist nicht automatisiert und muß vom Benutzer manuell ausgeführt werden.

5.5 Erstellen der initialen Version

Der Import eines Projektes in die Versionsverwaltung umfaßt, neben dem Kopieren der Dateien in die Verzeichnisstruktur, zumeist eine Umstrukturierung der Quelltextdateien und Quelltextänderungen. Die Durchführung dieser Änderungen und die zum Überprüfen der Änderungen notwendigen Testläufe sind, ebenso wie die abschließende Archivierung des Quelltextes als initiale Version, Thema dieses Unterkapitels.

5.5.1 Notwendige Quelltextänderungen

Die Implementierung der Versionsverwaltung als Erweiterung von OPAL benötigt den Quelltext in einer Hauptdatei, in der jedoch weitere Dateien als *Includes* eingebunden sein können. Daher sind Quelltextdateien entweder zu einer zusammenzufassen oder durch *Includes* zu verbinden.

Alle Eingabedateien befinden sich im Eingabedatenverzeichnis. Zudem werden manuelle Eingaben nicht unterstützt. Dies bedeutet, daß im Quelltext Fortran-Befehle, wie z.B. `open` oder `inquire`, die den Pfad zu einer Datei als Parameter benötigen, angepaßt werden müssen. In jedem Verzeichnis, in dem *Executables* ausgeführt werden, existiert ein *Link* mit dem Namen `data` auf das Eingabedatenverzeichnis, so daß die Eingabedateien mit `data/<filename>` relativ adressiert werden können. Manuelle Eingaben müssen entweder durch Zuweisungen oder durch Lesen der Eingaben aus einer Eingabedatei ersetzt werden.

```
USER_FFLAGS = -O4 -Mvect -Knoieee
USER_LIBS =

include ../../Makefile.def
```

Abbildung 5.6: Aufbau des lokalen Makefile

Im in Abbildung 5.6 gezeigten *Makefile* sind nur die zu verwendenden *Compiler*-Optionen und spezielle Bibliotheken einzutragen.

Die `USER_FFLAGS` des Fortran77-*Compiler* sind mit `-O4 -Mvect -Knoieee` vorbelegt, können jedoch vom Entwickler geändert werden. Diese Optionen sind für den auf der INTEL Paragon, der Referenzplattform für SVM-Fortran, installierten INTEL Fortran77-*Compiler* vorgesehen. Mit `-O4` werden neben Standardoptimierungen auch aggressive Optimierungen und *Loop Pipeling* durchgeführt. Die `Mvect`-Option aktiviert den Vektorisierer, und durch die `Knoieee`-Option wird das Abfangen von *Underflows* verhindert und das Unterprogramm zur Berechnung einer IEEE konformen Fließkommadivision durch einen deutlich schnelleren Algorithmus ersetzt.

Werden für das Programm spezielle Bibliotheken benötigt, sind diese in der Zeile `USER_LIBS` einzutragen. Die INTEL NX-Bibliothek, die Unterprogramme zur Interprozessorkommunikation bereitstellt, sowie die SVM-Fortran-Laufzeitbibliothek werden standardmäßig angebunden. Ebenso wird die Bibliothek zur Ermittlung von Laufzeidaten SAM bei Bedarf automatisch angebunden.

5.5.2 Testen des Prototyps

Zum Austesten der durchgeführten Anpassungen kann der, im folgenden Prototyp genannte, Quelltext mit der Funktion `Source -> Launch Program` übersetzt und ausgeführt werden.

Dieser Prototyp ist ein an die im vorigen Unterkapitel beschriebenen Konventionen der Versionsverwaltung angepaßtes sequentielles Programm. Daher erscheint vor der Ausführung des Programms kein Dialogfenster zur Eingabe der für den Testlauf zu verwendenen Prozessorzahl.

Zur Übersetzung wird jedoch, wie bei den zu entwickelnden parallelen Programmversionen, der Quelltext des Prototyps zunächst mit dem SVM-Fortran-*Compiler* nach Fortran77 und danach mit dem Fortran77-*Compiler* des Parallelrechners in ein *Executable* übersetzt.

5.5.3 Archivieren des Prototyps als initiale Version

Nachdem alle erforderlichen Anpassungen durchgeführt und getestet sind, wird der Prototyp als initiale Version archiviert, um als Basis der weiteren Entwicklung zu dienen. Beim Anlegen der initialen Version wird kein Eingabefenster für einen Kommentar angezeigt. Als Kurzkommentar wird `initial version` verwendet, der ausführliche Kommentar enthält nur den Text des „Formblattes“. Der ausführliche Kommentar kann jedoch später mit dem Menüpunkt `Version` → `Maintain Current Version` → `Edit Comment` geändert werden.

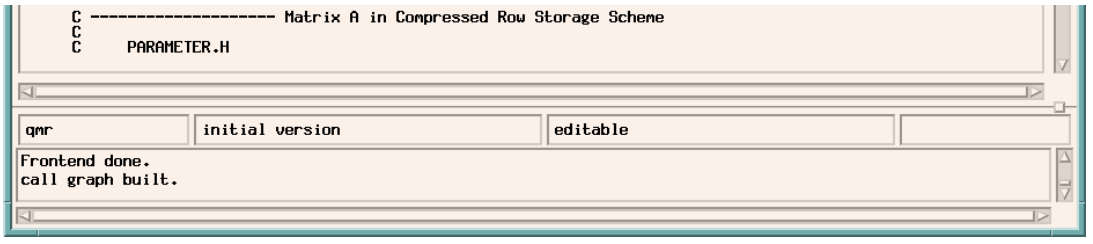


Abbildung 5.7: Die Statusfelder nach Archivierung des Prototyps

Nachdem die initiale Version archiviert ist, wird, wie in Abbildung 5.7 gezeigt, im zweiten Feld der Statusanzeige der Kurzkommentar zur Version, also hier `initial`

`version` eingetragen. Die Versionsverwaltung verbleibt, im Gegensatz zum Verhalten beim Archivieren der folgenden Quelltextversionen, im Zustand `Work`, in dem der Benutzer mit der Parallelisierung beginnen kann.

5.6 Ablauf der Programmentwicklung

Das Thema dieses Unterkapitels ist der Ablauf der Parallelisierung und Optimierung einer Anwendung. Die Beschreibung der einzelnen Arbeitsabläufe gibt einen Überblick über die neue Funktionalität von OPAL. Daneben wird hierduch zusätzlich eine Einführung in die empfohlene Vorgehensweise bei der Anwendung der Versionsverwaltung gegeben, die stark durch das *Work/Version*-Konzept geprägt ist.

5.6.1 Ändern des Quelltextes

Die Änderungen des Quelltextes werden an der Arbeitskopie des Quelltextes durchgeführt. Folglich ist die mit dem Menüpunkt `Source` → `Edit Source` aufzurufende *Editor*-Funktion nur im Zustand `Work` verfügbar.

Die Verwendung eines externen *Editor* hat für den Anwender den Vorteil, daß die Einarbeitung in einen weiteren *Editor* entfällt. Auf der anderen Seite hat dies den Nachteil, daß die Versionsverwaltung ein Abspeichern des bearbeiteten Quelltextes oder das Verlassen des *Editor* nicht bemerkt. Obwohl vor der Übersetzung und Ausführung der Arbeitskopie des Quelltextes der in OPAL angezeigte Quelltext aktualisiert wird, ist durch die Funktion `Source` → `Read Source` die Möglichkeit gegeben, den in OPAL angezeigten Quelltext manuell zu aktualisieren.

5.6.2 Testen des Quelltextes im Arbeitsverzeichnis

Befindet sich die Versionsverwaltung im Zustand `Work`, wird durch die Funktion `Source` → `Launch Program` die Arbeitskopie des Quelltextes in OPAL neu eingelesen und dargestellt, falls notwendig neu übersetzt und danach ausgeführt. Hierzu erscheint das in Abbildung 5.8 dargestellte Dialogfenster.

Im Feld `number of processors` muß die Anzahl der für den Testlauf zu verwendenden Prozessoren angegeben werden. Weiterhin kann mit der Option `enable compilation for tracing` angegeben werden, ob das Programm beim Übersetzen auf *Performance*-Messungen vorbereitet werden soll. Dabei ist zu beachten,

daß diese Option nur beim Übersetzen verwendet wird. Mit der Option `force new compilation` kann jedoch eine Übersetzung erzwungen werden.

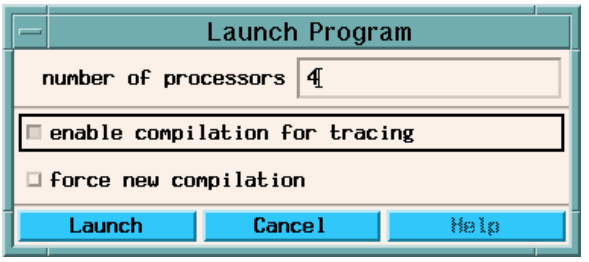


Abbildung 5.8: Dialogfenster zum Ausführen eines Testlaufs

Nach Beendigung des Testlaufs wird die Anzahl der verwendeten Prozessoren im vierten Feld der Statusanzeige eingetragen. Darüberhinaus werden, falls eine Messung durchgeführt wurde, die Meßergebnisse automatisch eingelesen und dargestellt.

5.6.3 Anlegen einer neuen Quelltextversion

Zum Anlegen einer neuen Quelltextversion muß sich die Versionsverwaltung im Zustand `Work` befinden. Beim Archivieren wird neben dem Quelltext auch, falls existierend, die aktuelle Meßkonfiguration mit den zugehörigen Meßwerten in das automatisch angelegte Versionsverzeichnis dupliziert.

Die neue Quelltextversion wird mit der Funktion `Work` → `Archive Source` angelegt. Hierbei muß zuerst das Feld `abstract` des, in Abbildung 5.9 gezeigten, Dialogfensters ausgefüllt werden. Das Schreiben des ausführlichen Kommentares ist optional und kann auch später mit der Funktion `Version` → `Maintain Current Version` → `Edit Comment` erfolgen. Diese Version dient jedoch primär zur laufenden Aktualisierung des Kommentares.

Der Kurzkommentar (*abstract*) dient zur Orientierung im Versionsbaum. Er erscheint vollständig in der zur Auswahl einer Quelltextversion gezeigten Liste. Das erste Wort des Kurzkommentares wird zur Bezeichnung einer Quelltextversion in der graphischen Darstellung des Versionsbaums verwendet und sollte somit bereits eine grobe Vorstellung vom Inhalt der Quelltextversion vermitteln.

Die Aufgabe des ausführlichen Kommentares (*comment*) ist die Dokumentation der

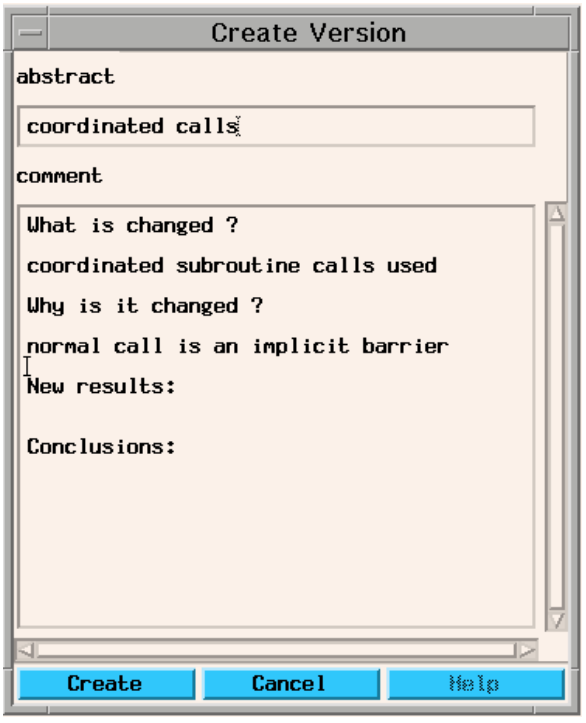


Abbildung 5.9: Dialogfenster zum Anlegen einer neuen Version

durchgeführten Quelltextänderungen sowie der gewonnenen Meßergebnisse und Erkenntnisse. Zur Gliederung des Kommentares werden Überschriften (Abbildung 5.9) vorgegeben. Diese werden aus der Datei `.opal/.Templates/.versionComment` gelesen und können als Bestandteil des Kommentartextes geändert werden. Zudem kann die *Template*-Datei selbst geändert werden.

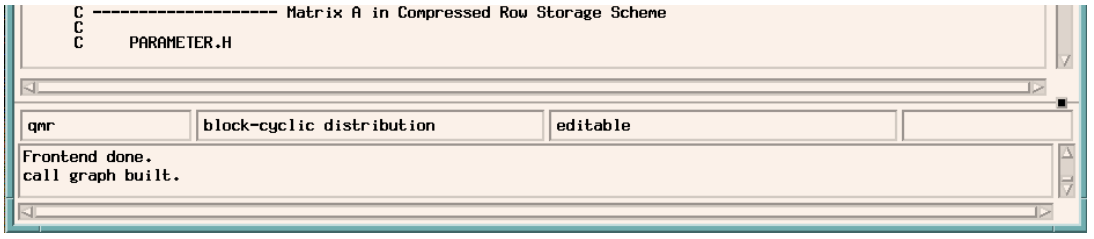


Abbildung 5.10: Die Statusfelder nach Archivierung des Quelltextes

Nachdem die neue Quelltextversion erstellt ist, wird wie in Abbildung 5.10 gezeigt im zweiten Feld der Statusanzeige der Kurzkommentar der Version eingetragen und

die Versionsverwaltung wechselt vom Zustand `Work` in den Zustand `Version`, so daß im Anschluß, ohne mit der Funktion `Work` → `Switch To Version` explizit wechseln zu müssen, mit den *Performance*-Messungen begonnen werden kann.

5.6.4 Performance-Messung an einer Quelltextversion

Die Möglichkeit, einen Testlauf der Arbeitskopie des Quelltextes ausführen zu können, soll nur zur Überprüfung der Korrektheit des Quelltextes genutzt werden. Die eigentliche Messung des Laufzeitverhaltens sollte an einer archivierten und somit fixierten Quelltextversion durchgeführt werden. Im Gegensatz zum Arbeitsverzeichnis können im Versionsverzeichnis mehrere Meßkonfigurationen koexistieren und diese zudem kommentiert werden.

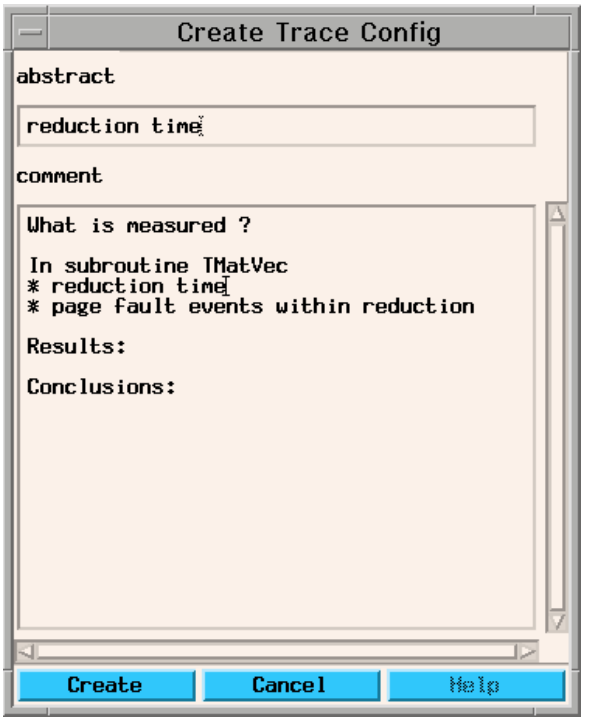


Abbildung 5.11: Dialogfenster zum Anlegen einer neuen Meßkonfiguration

Bei der Erstellung der Meßkonfigurationen für eine Quelltextversion wird, wie beim Anlegen der Quelltextversionen selbst, das *Work/Version*-Konzept angewendet. Hierbei entspricht das *Trace Request File* im Versionsverzeichnis dem Quelltext im Arbeitsverzeichnis und dient als Basis für die archivierten Meßkonfigurationen.

Das in Abbildung 5.11 gezeigte Dialogfenster zum Anlegen einer neuen Meßkonfiguration ist daher mit dem in Abbildung 5.9 dargestellten Dialogfenster zum Anlegen einer neuen Quelltextversion, bis auf die Titel des Fensters und den im Eingabefeld `comment` enthaltenen Überschriften zur Gliederung des Kommentars, identisch. Der im Eingabefeld `comment` gezeigte Text wird aus der Datei `.traceConfigComment` gelesen und ist, wie die *Template*-Datei selbst, änderbar.

Wie beim Anlegen einer neuen Quelltextversion, muß der zur Orientierung dienende Kurzkommentar angegeben werden, wohingegen das Schreiben des ausführlichen Kommentares optional ist. Im Verlauf der Messungen kann der ausführliche Kommentar mit der Funktion `Version` → `Maintain Trace Config` → `Edit Comment` aktualisiert werden.

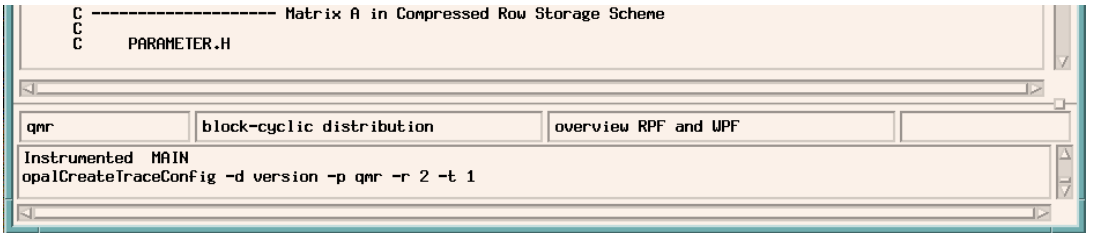


Abbildung 5.12: Die Statusfelder nach Archivierung einer Meßkonfiguration

Im dritten Feld der Statusanzeige wird im Anschluß, wie in Abbildung 5.12 dargestellt, der Kurzkommentar der Meßkonfiguration eingetragen.

Nach der Instrumentierung wird der Testlauf mit der Funktion `Source` → `Launch Program` gestartet. Es erscheint das in Abbildung 5.8 gezeigte Dialogfenster, in dem jedoch die Schalter zum Abschalten des *Tracing* und zum Erzwingen einer Übersetzung inaktiv sind.

Zum Besuchen einer Meßkonfiguration mit `Version` → `Visit Trace Config` erscheint das in Abbildung 5.13 gezeigte Dialogfenster. Im obersten Feld werden die Kurzkommentare (*abstacts*) der existierenden Meßkonfigurationen aufgelistet. Durch Drücken des *Button* `use transient trace configuration` wird die transiente Meßkonfiguration im Versionsverzeichnis verwendet. In der folgenden Zeile wird der Kurzkommentar und in einem zweiten, nicht in Abbildung 5.13 dargestell-

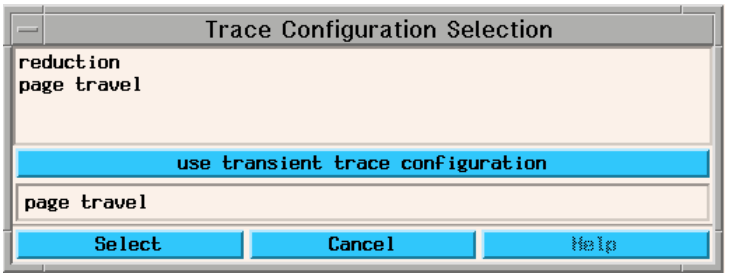


Abbildung 5.13: Dialogfenster zur Auswahl einer Meßkonfiguration

ten Fenster, der ausführliche Kommentar der aktuellen Wahl angezeigt. Die Wahl wird durch Drücken des `Select`-*Button* ausgeführt oder mit dem `Cancel`-*Button* abgebrochen. Nach erfolgter Wahl wird im dritten Feld der Statusanzeige der Text aktualisiert.

Eine archivierte Meßkonfiguration kann mit `Version` → `Retrieve Trace Config` in das Versionsverzeichnis geladen werden. Die existierende Meßkonfiguration wird dabei überschrieben. Zur Wahrung der Integrität der Daten werden im Versionsverzeichnis die *Trace Data Files* gelöscht. Auch hier wird nach erfolgter Aktion das Statusfeld für die Meßkonfiguration aktualisiert.

5.6.5 Wechseln zwischen Quelltextversionen

Abbildung 5.14 zeigt den zum Wechseln zwischen Quelltextversionen aufgebauten, aus drei Fenstern bestehenden Dialog. Im `VersionTree` Fenster, das bereits alle zur Auswahl notwendigen Informationen und Funktionen enthält, nimmt die graphische Darstellung des Versionsbaums den größten Raum ein. Die Knoten dieses Baums sind *Push Buttons*, die mit dem ersten Wort des Kurzkommentares beschriftet sind. Durch Drücken einer dieser *Buttons* wird eine Version ausgewählt. Unterhalb der Baumdarstellung wird die aktuelle Wahl angezeigt. Wie bei der Wahl einer Meßkonfiguration wird die Wahl durch Drücken des `Select`-*Button* ausgeführt, oder mit dem `Cancel`-*Button* abgebrochen.

Eine Liste der Kurzkommentare der aktuell gewählten Version und ihrer Nachfolgeversionen wird im `Version Scroll List` Fenster angezeigt. Diese Liste wird beim Auswählen einer Version im Versionsbaum neu aufgebaut. Ebenso wird bei der Wahl einer Version in der Liste die Anzeige der aktuell gewählten Version im `VersionTree`

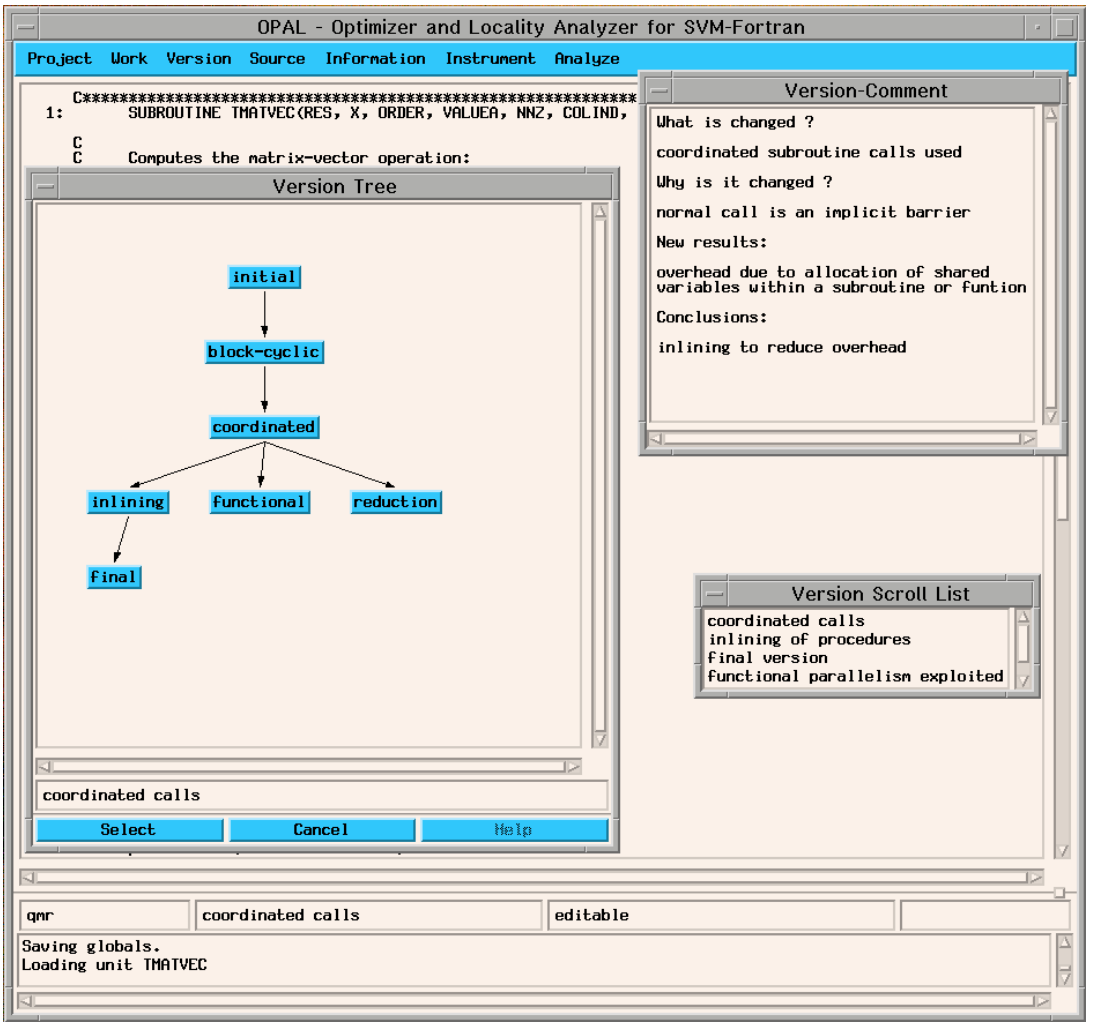


Abbildung 5.14: Dialogfenster zur Auswahl einer Version

Fenster aktualisiert.

Im dritten Fenster, das den Namen `Version Comment` trägt, wird der ausführliche Kommentar der aktuell gewählten Version angezeigt.

5.6.6 Wechseln von Work nach Version

Mit der Funktion `Version` → `Switch To Work` wird vom Zustand `Work` in den Zustand `Version` gewechselt.

In einem Projekt existiert genau eine Arbeitskopie des Quelltextes. Daher benötigt der Wechsel von einer Quelltextversion zur Arbeitskopie des Quelltextes mit `Version` → `Switch To Work` keinen Dialog.

Nach erfolgtem Wechsel zur Arbeitskopie des Quelltextes wird im zweiten Feld der Statusanzeige das Schlüsselwort `work`, gefolgt vom in Klammern gefaßten Kurzkommentar der Quelltextversion, auf der die Arbeitskopie basiert, angezeigt.

5.6.7 Rückgriff auf eine Quelltextversion

Bei der Parallelisierung und Optimierung des Gleichungssystemlösers ist zweimal auf die letzte Quelltextversion des Hauptzweigs ohne *Inlining* zurückgegriffen worden. Ein solcher Rückgriff auf eine Quelltextversion wird in der Versionsverwaltung durch die Funktion `Work` → `Retrieve Version` ausgeführt. Beim Kopieren der Quelltextversion in das Arbeitsverzeichnis wird die alte Arbeitskopie überschrieben. Zur Auswahl der Quelltextversion wird der in Abbildung 5.14 gezeigte Dialog ein weiteres Mal verwendet. Beim Archivieren der neuen, veränderten Arbeitskopie wird die neue Quelltextversion als Nachfolgeversion der Quelltextversion, auf die zurückgegriffen wurde, in den Versionsbaum eingehangen, so daß ein neuer Entwicklungspfad entsteht.

5.6.8 Wechseln von Work nach Version

Beim Wechsel von der Arbeitskopie des Quelltextes zu einer Quelltextversion mit `Work` → `Switch To Version` wird der gleiche Dialog (siehe Abbildung 5.14) wie beim Wechsel zwischen zwei Quelltextversionen angezeigt. Als aktuell gewählte Version wird beim Aufbau des Dialogs die zuletzt besuchte Quelltextversion eingetragen, so daß beim mehrfachen Wechsel zwischen der Arbeitskopie des Quelltextes und einer Quelltextversion nur der `Select`-*Button* gedrückt werden muß.

Nach dem Wechsel wird die Statusanzeige aktualisiert und im zweiten Feld wird der Kurzkommentar eingetragen.

Kapitel 6

Zusammenfassung und Ausblick

Das Ziel der vorliegenden Arbeit war es, dem Entwickler die Verwaltung der im Optimierungsprozeß anfallenden Quelltextversionen, Meßkonfigurationen und Meßergebnisse zu erleichtern.

Bisher mußte der Entwickler diese Dateien selbst verwalten. Neben dem damit verbundenen Zeitaufwand, hatte dies den Nachteil, daß die manuelle Verwaltung fehlerträchtig war. Durch eine falsche Zuordnung einer Quelltextversion mit einem *Trace Request File* konnte die Zuordnung von Meßergebnissen zu den Regionen verschoben werden.

Die Implementierung der Versionsverwaltung ist von zwei Aspekten geprägt. Der erste Aspekt ist der Konflikt zwischen Automatisierung von Aufgaben zur Vermeidung von Fehlern durch Fehlbedienung auf der einen Seite, und die strikte Festlegung des Vorgehens mit der damit verbundenen Beschränkung der Freiheit des Entwicklers bei seiner Wahl der Vorgehensweise auf der anderen Seite.

Bei der vorliegenden Implementierung der Versionsverwaltung ist ein von SVM-Fortran Anwendern als sinnvoll und wenig einschränkend empfundenes Vorgehensmodell umgesetzt worden. Dieses führt zum *Work/Version*-Konzept, das den Entwurf und die Realisierung der Versionsverwaltung wie ein roter Faden durchzieht.

Der zweite die Arbeit prägende Aspekt ist die Portabilität im Sinne der Wiederverwendbarkeit der im Rahmen der Arbeit entwickelten *Software*-Komponenten. Dieser Aspekt hat sowohl die Wahl der verwendeten Programmiersprachen, als auch die Strukturierung des Quelltextes maßgeblich beeinflußt. Obwohl beim Entwurf Me-

thoden des objektorientierten Designs verwendet wurden, ist die Versionsverwaltung in ANSI C89 und nicht in ANSI C++ kodiert. Durch diese Wahl können *Software*-Komponenten sowohl in Analysewerkzeuge, die in ANSI C89 geschrieben sind, als auch solche in denen ANSI C++ Verwendung findet, eingebunden werden.

Bei der Festlegung des zu realisierenden Funktionsumfangs wurde die ursprüngliche Aufgabe, die Verwaltung der Dateien, erweitert. Zur Verwaltung der Quelltextversionen kam eine Verwaltung von Projekten. Weiterhin wurde durch, in die Analyseumgebung integrierte Funktionen zum Ändern des Quelltextes und zur Übersetzung und Ausführung des Programms, das Analysewerkzeug OPAL in eine integrierte Entwicklungsumgebung zur Parallelisierung und Optimierung von SVM-Fortran Programmen umgewandelt. Funktionen zur die Entwicklung begleitenden Dokumentation des Optimierungsprozesses runden die gebotene Funktionalität ab.

Die Vorteile für den Entwickler sind effektiveres Arbeiten durch:

- flexiblen und schnellen Zugriff auf Informationen,
- die Entwicklung begleitende Dokumentation,
- die Integration aller zur Programmentwicklung notwendigen Funktionen in eine Bedienoberfläche.

Die Integration aller zur Optimierung einer Anwendung notwendigen Funktionen in eine Bedienoberfläche ist nicht üblich. Andere Analysewerkzeuge wie z.B. Pablo (University of Illinois) [28, 29], ParaDyn (University of Wisconsin) [30] oder PARvis (Forschungszentrum Jülich) [26, 27] bieten nur Funktionen zur Analyse des Laufzeitverhaltens, einige auch Funktionen zum Starten der Testläufe. Integrierte Entwicklungsumgebungen bieten Funktionen zum Ändern und Übersetzen des Quelltextes. Funktionen zur Analyse des Laufzeitverhaltens sind hier zumeist nicht und wenn nur rudimentär vorhanden. Eine Verwaltung von Programmversionen und Funktionen zur Dokumentation des Vorgehens im Verlauf der Entwicklung sind in beiden Fällen nicht vorhanden.

Eine bereits in der Arbeit angesprochene Erweiterung ist die Mehrbenutzerunterstützung. Diese Erweiterung wird bereits im Entwurf vorgesehen und ist mit den anderen Konzepten beschrieben worden.

Eine Wiederverwendung der in dieser Arbeit entwickelten *Software*-Komponenten in anderen Analysewerkzeugen oder in einem eigenständigen Programm ist ebenfalls im Entwurf berücksichtigt. Durch Kapselung von Programmcode und Daten in abgeschlossene Einheiten wird die spätere Wiederverwendung erleichtert. Eine Einbindung der Versionsverwaltung in PARvis [26, 27] – ein Analysewerkzeug für *Message-Passing*-Programme – benötigt lediglich eine Anpassung der *Shell*-Skripte und des *Makefile*. Der in ANSI C89 geschriebene Kern kann nach dem in Kapitel 4.1 beschriebenen Schema eingebunden werden.

Bei der Parallelisierung des iterativen Gleichungssystemlösers sind zwei verschiedene Datensätze verwendet worden. Ein kleineres, schnell konvergierendes Gleichungssystem diente zur Überprüfung des Konvergenzverhaltens. Die Laufzeitmessungen wurden jedoch an einem größeren System durchgeführt. Eine Verwaltung mehrerer Eingabedatensätze ist somit in einigen Projekten wünschenswert.

Als Erweiterung der Funktionen zur Dokumentation ist eine automatische Generierung von *Reports* aus den ausführlichen Kommentaren denkbar. Wird hierzu ein gängiges Dateiformat wie z.B. LaTeX oder `RTF` (*rich text format*) verwendet, können diese *Reports* einfach in Berichte eingebunden werden.

Durch das Konzept der inkrementellen Entwicklung und Analyse wird in SVM-Fortran die Entwicklungszeit gegenüber dem *Message-Passing*-Konzept erheblich reduziert. Die neuartige Kombination von Programmentwicklung und Analyse in einer Bedienoberfläche beschleunigt die Portierung von Anwendungen auf Parallelrechner um ein weiteres Stück.

Literaturverzeichnis

[1] G. Wolf und R. Krahl, *Stand und Perspektiven des Parallelen Höchstleistungsrechnens und seiner Anwendungen*, Statustagung des BMBF (HPSC'95) in Jülich, 11.-14. September 95, herausgegeben durch den Projektträger des BMBF für Informationstechnik bei der DLR e.V., Jülich, 1995.

[2] F. Hoßfeld und W. E. Nagel, *Per Aspera ad Astra: On the Way to Parallel Processing*, Interner Bericht, KFA-ZAM-IB-9507, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich, 1995.

[3] F. Hoßfeld, *On „Retarded Potentials" in High-Performance Scientific Computing*, Interner Bericht, KFA-ZAM-IB-9426, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich, 1994.

[4] F. Hoßfeld, *On the Super-computional Backgound of the Research Centre Jülich*, Interner Bericht, KFA-ZAM-IB-9502, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich, 1995.

— Arbeiten zum Thema SVM (shared virtual memory) —

[5] K. Li, *Shared Virtual Memory on Loosely Coupled Microprocessors*, Ph. D. Thesis, Yale University, 1986.

[6] St. Zeisset, *Evaluation and Enhancement of the Paragon Multiprocessor's Shared Virtual Memory System*, Diplomarbeit, TU München, Lehrstuhl für Rechnertechnik und Rechnerorganisation, 1993.

[7] J. Göbel, *Evaluation of Shared Virtual Memory on the Paragon Supercomputer*, Diplomarbeit, TU München, Lehrstuhl für Rechnertechnik und Rechnerorganisation, 1995.

— Performance Analyse bei SVM-Systemen —

[8] M. Gerndt, *Performance Analysis Environment for SVM-Fortran Programs*, Interner Bericht, KFA-ZAM-IB-9417, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich 1994.

[9] M. Mairandres, *ASVM Monitoring Support Specification*, to be published by Intel Corporation, Rev. 0.1, 1994/1995.

[10] S. N. Özmen, *SAM: Performance-Analyse-Monitor für SVM-Fortran*, Jül-Bericht, Jül-3116, Forschungszentrum Jülich, Jülich, 1995.

[11] M. Gerndt, A. Krumme und S. Özmen, *Performance Analysis for SVM-Fortran with OPAL*, Int. Conference on Parallel Processing Techniques and Applications (PDPTA'95), Athens, Georgia, USA, Seite 561-570, 1995.

[12] M. Gerndt und A. Krumme, *Program Optimization for Shared Virtual Memory*, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich, 1996.

[13] M. Gerndt und A. Krumme, *Automatic Performance Analalysis for Shared Virtual Memory Systems*, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, interner Bericht, KFA-ZAM-IB-9631, Jülich 1996.

— SVM-Fortran Dokumentation —

[14] R. Berrendorf, M. Gerndt, W. Nagel und J. Prümmer, *SVM-Fortran*, Interner Bericht, KFA-ZAM-IB-9322, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich 1993.

[15] R. Berrendorf und M. Gerndt, *SVM-Fortran Reference Manual — Version 1.4*, Interner Bericht, KFA-ZAM-IB-9510, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich 1995.

[16] R. Berrendorf und M. Gerndt, *Compiling SVM-Fortran for the Intel Paragon XP/S*, Proceedings of the Working Conference on Massively Parallel Programming Models (MPPM'95), IEEE Press, Los Alamitos, 1995.

[17] R. Berrendorf, *Der FORTRAN-Parser PAFF als wiederverwendbares Werkzeug für Programmier-Tools*, Interner Bericht, Jül-Spez-537, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich 1989.

— Mit SVM-Fortran parallelisierte Anwendungen —

[18] M. Gerndt, *Parallelization of the AVL FIRE Benchmark with SVM-Fortran*, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, interner Bericht, KFA-ZAM-IB-9520, Jülich, 1995.

[19] M. Gerndt und R. Berrendorf, *Parallelizing Applications with SVM-Fortran*, International Conference on High- Performance Computing and Networking (HPCN '95), Mailand, LNCS 919, pp. 793-798, 1995

[20] R. Berrendorf, M. Gerndt, A. Krumme und S. Özmen, *SVM-Fortran: Eine Programmierumgebung für massiv-parallele Rechner*, PIK, Vol. 19, No. 3, pp. 142 -147, 1996.

[21] O. Weiss, *Partitionierung Unstrukturierter Gitter für Shared-Virtual-Memory-Rechner*, Diplomarbeit RWTH Aachen, Bericht des Forschungszentrum Jülich No. Jül-3336, 1997.

— Weitere Fortran-Dialekte für Parallelrechner —

[22] HPFF, *High Performance Fortran Language Specification 1.1*, November 1994, Rice University, Houston, Texas, 1994.

[23] P. Brezany, B. M. Chapman, P. Mehrotra, A. Schwald und H. P. Zima, *VIENNA FORTRAN - A Language Specification Version 1.1*, Technical Report ACPC/TR 92-4, Institut für Softwaretechnik und parallele Systeme, Universität Wien, 1992.

[24] Kendall Square Research Corp., *KSR/Series Fortran Programming*, KSR/Series Manuals, Waltham, 1993.

[25] F. Bodin, L. Kervella und T. Priol, *Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures*, In Proceedings of *Supercomputing'93*, Portland, Oregon, November 19–22, 1993.

— Analysewerkzeuge für Message Passing Programme —

[26] A. Arnold, U. Detert und W. Nagel, *Performance Optimization of Parallel Programs - Tracing, Zooming, Understanding*, Interner Bericht, KFA-ZAM-IB-9508, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich 1995.

[27] A. Arnold, *Performance-Analyse mit PARvis* , Paragon Workshop für Fortgeschrittene, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich 1995.

[28] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields und B. W. Schwartz, *Pablo: An Extensible Performance Analysis Environment for Parallel Systems*, The University of Illinois Pablo Research Group, University of Illinois, Illinois, 1992.

[29] M. Voss, *Untersuchung der Performance Analyse Umgebung PABLO Release 4.0*, Studienarbeit, Lehrstuhl für Technische Informatik und Computerwissenschaften, RWTH-Aachen, Aachen, 1995.

[30] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam und T. Newhall, *The Paradyn Parallel Performance Measurement Tools*, IEEE Computer 28, 11, 1995.

— Intel Paragon Message Passing Bibliothek —

[31] P. Mitra, D. G. Payne, L. Shuler, R. van de Gejin und J. Watts, *Fast Collective Communication Libraries, Please*, InterCom Project, University of Texas at Austin, 1995.

— Parallelisierte QMR Algorithmen —

[32] R. W. Freund und W. N. Nachtigal, *QMR: A Quasi-Minimal-Residual Method for Non-Hermitian Linear Systems*, Numerische Mathematik, 60(3), pp 315-339, 1991.

[33] R. W. Freund und W. N. Nachtigal, *An implementation of the QMR Method Based on Coupled Two-Term Recurrences*, SIAM Journal on Scientific Computing, 15(2), pp 313-337, 1994.

[34] A. Basermann, *Iterative Verfahren für dünnbesetzte Matrizen zur Lösung technischer Probleme auf massiv-parallelen Systemen*, KFA-ZAM-JÜL-3015, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich GmbH, Jülich, 1995.

[35] H. M. Bücker und M. Sauren, *A Parallel Version of the Unsymmetric Lanczos Algorithm and its Application to QMR*, Interner Bericht, KFA-ZAM-IB-9605, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich GmbH, Jülich, 1996.

— Werkzeuge zur Verwaltung von Quelltexten —

[36] M. J. Rochkind, *The Source Code Control System*, Dezember 1975, IEEE Transactions on Software Engineering, vol. SE-1, no. 4, pp 364-370 , 1975.

[37] W. F. Tichy, *Design, Implementation and Evaluation of a Revision Control System*, in Proceedings of the 6th International Conference on Software Engineering, Seite 58-67, ACM, IEEE, IPS, NBS, 1982.

[38] W. F. Tichy, *RCS — A System for Version Control*, Department of Comuter Sciences, Purdue University, West Lafayette, Indiana, 1991.

[39] P. Cederqvist et al., *Version Managment with CVS*, Signum Support AB, Schweden, 1996

[40] B. Berliner, *CVS II: Parallelizing Software Development*, Prisma Inc., Colorado Springs, Colorado, 1989.

— Datenbanksysteme —

[41] H. R. Hansen, *Witschaftsinformatik*, 6. Auflage, Gustav Fischer Verlag, Stuttgart - Jena, 1992.

[42] DIN, *Datenbanksprache SQL mit erweiterten Integritätsregeln*, DIN Deutsches Institut für Normung e.V., Berlin, Juli 1990.

[43] Oracle Corporation, *Oracle7™ Server SQL Reference*, Februar 1996, Oracle Corporation, Redwood City, CA, 1996.

[44] J. Uhl, D. Theobald, B. Schiefer, M. Ranft, W. Zimmer und J.Alt , *The Object Managment System of Stone*, 5. Mai 1994, Forschungszentrum Informatik (FZI), Karlsruhe, 1994.

[45] B. Schiefer, D. Theobald, J. Uhl, E. Casais und A. Freyberg, *User's Guide – OBST Release 3.4* , 14. Juni 1994, Forschungszentrum Informatik (FZI), Karlsruhe, 1994.

[46] Xcc Software, OBST+ Das objektorientierte Datenbanksystem für C++ Entwickler, Mai 1995, Xcc Software, Karlsruhe, 1995.

— UNIX Shells —

[47] S. R. Bourne, *UNIX Time-Sharing System: The UNIX Shell*, Bell Systems Technical Journal, 57(6), pp 1971-1990, Juli-August 1978.

[48] IEEE, *IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Part 2: Shell and Utilities*, 1992.

[49] C. Ramey, *Bash – The GNU Shell*, Case Western Reserve University, 1992.

[50] B. Fox und C. Ramey, *Bash Features*, Free Software Foundation, August 1994.

[51] M. Bolsky und D. Korn, *The Korn Shell Command and Programming Language*, Prentince Hall, 1989.

[52] B. Joy, *An Introduction to the C Shell*, UNIX User's Supplementary Documents, University of California at Berkeley, 1986.

[53] T. Duff, *Rc – A Shell for Plan 9 and UNIX systems*, Proc. of the Summer 1990 EUUG Conference, pp 21-33, London, Juli 1990.

— UNIX Programmierumgebung —

[54] R. M. Stallman und R. McGrath, *GNU Make – A Program for Directing Recompilation*, Free Software Foundation, Cambridge, Massachusetts, 1985.

[55] R. M. Stallman, *GNU Emacs Manual*, Free Software Foundation, August 1993.

[56] B. W. Kernighan und D. M. Ritchie, *The C Programming Language*, Second Edition, ANSI C, Bell Telephone Laboratories Inc., Prentice-Hall International, Englewood Cliffs, NJ, 1988.

[57] D. Heller, *Motif Programming Manual*, 1. Auflage, September 1991, O'Reilley & Associates Inc., Sebastopol, CA, 1991.

[58] J. R. Levine, T. Mason und D. Brown, *lex & yacc*, 2. Auflage, Februar 1995, O'Reilley & Associates Inc., Sebastopol, CA, 1995.

[59] D. Gilly et al., *UNIX in a Nutshell*, 2. Auflage, August 1994, O'Reilley & Associates Inc., Sebastopol, CA, 1994.

Danksagung

Die vorliegende Arbeit wurde als Diplomarbeit in Elektrotechnik am Lehrstuhl für Technische Informatik und Computerwissenschaften der Fakultät für Elektrotechnik der Rheinisch-Westfälischen Technischen Hochschule Aachen (RWTH) angefertigt. Dem Lehrstuhlinhaber Herrn Prof. Dr. F. Hoßfeld danke ich für die Möglichkeit, die Arbeit am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrum Jülich erstellen zu können.

Mein außerordentlicher Dank gilt Herrn Dipl. Ing. A. Krumme, der die Betreuung dieser Arbeit übernommen hat und der meinen Ideen stets aufgeschlossen gegenüberstand. Bei Herrn Dipl. Inform. R. Berrendorf und Herrn Dr. H. M. Gerndt bedanke ich mich für die Beantwortung unzähliger Detailfragen zum SVMF-Compiler und zum Analysewerkzeug OPAL. An dieser Stelle möchte ich mich auch bei Herrn Dipl. Ing. Dipl. Inform. M. Bücker und Herrn Dr. B. Steffen bedanken, die mir in zahlreichen Diskussionen halfen, die Probleme der Numerik besser zu verstehen. Nicht vergessen sind auch die Systembetreuer, insbesondere Herrn Dipl. Inform. H. Bast und Frau Dipl. Inform. J. Docter (Intel Paragon), Herrn Dipl. Inform. V. Sander (SUN Workstations) sowie Herrn T. Eickermann (IBM SP2), die durch ihre Arbeit für einen störungsarmen Betrieb der von mir verwendeten Rechner sorgten. Herrn Dipl. Inform. H. Bast kommt hier besonderer Dank zu, da er sich falls eben möglich Zeit für meine Fragen nahm und diese äußerst detailliert beantwortete.

Dank kommt auch Herrn W. Frings zu, der alle meine Fragen zu LaTeX bereitwillig beantwortet hat. Weiterhin danke ich meiner Freundin für die zahlreichen Stunden, die sie zum Korrekturlesen dieses Textes geopfert hat. Zudem bedanke ich mich bei den Operateuren von ZAM und TIA-G für das Drucken und Binden dieser Arbeit.

Weiterhin möchte ich die sehr gute kameradschaftliche Atmosphäre unter den Doktoranden und Diplomanden hervorheben, die außerordentlich zum guten Gelingen dieser Arbeit beigetragen hat.

Die Arbeit stellt den Abschluß meines Studiums dar, und nicht zuletzt deshalb möchte ich mich insbesondere bei meinen Eltern und nochmals bei meiner Freundin bedanken, die mich stets auf dem Wege durch mein Studium begleitet und nach Kräften unterstützt haben.