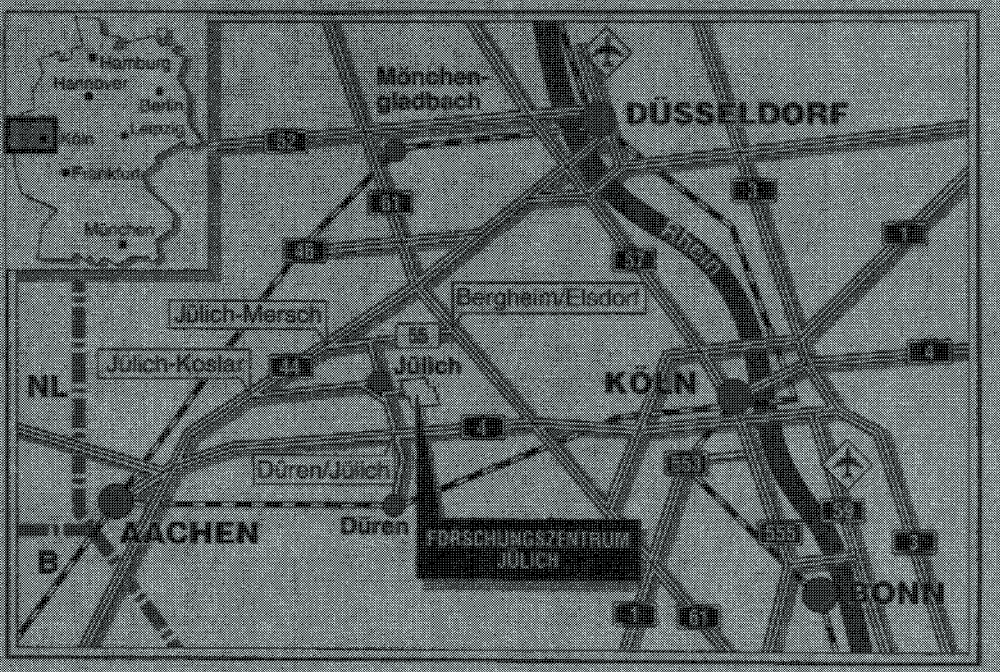
FORSCHUNGSZENTRUM JÜLICH GmbH
KFA

Zentralinstitut für Angewandte Mathematik

Neue PARvis-Komponenten zur effizienten Performance-Analyse von SVM-Fortran-Programmen

Jost Berner



Berichte des Forschungszentrums Jülich ; 3158
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3158

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
D-52425 Jülich · Bundesrepublik Deutschland
Telefon: 02461/61-6102 · Telefax: 02461/61-6103 · Telex: 833556-70 kfa d

Neue PARvis-Komponenten zur effizienten Performance-Analyse von SVM-Fortran-Programmen

Jost Bernert

Zusammenfassung

Der von den klassischen Rechnerarchitekturen bekannte gemeinsame Adreßraum wird bei massiv-parallelen Systemen mit verteiltem Speicher durch das Programmiermodell des gemeinsamen virtuellen Speichers (SVM) realisiert. Auf diesem Modell baut die Programmiersprache SVM-Fortran auf. Um ein besseres Verständnis der Vorgänge im Speicher während der Programmausführung von SVM-Fortran-Programmen zu erhalten, werden daher bestimmte Ereignisse in einer Trace-Datei protokolliert. Die Auswertung der gewonnenen Daten erfolgt mit dem Visualisierungswerkzeug PARvis, welches vielfältige Möglichkeiten zur Analyse und Darstellung bietet. Zur Nutzung von PARvis für SVM-Fortran wird in dieser Arbeit zunächst die Entwicklung eines Treibers für die im SDDF-Format vorliegenden Trace-Daten beschrieben. Im nächsten Schritt erfolgt die Erweiterung der Darstellungsmöglichkeiten von PARvis für Trace-Informationen, die SVM-Fortran spezifisch sind. Eine Ergänzung der Performance-Analyse um den Quelltextbezug bietet sich durch das Werkzeug OPAL an. Aufbauend auf der gemeinsamen Datenbasis (Trace-Daten) bildet die Konzeption und Entwicklung einer Schnittstelle zwischen diesen beiden Werkzeugen den Abschluß dieser Arbeit.

Abstract

For massively parallel systems, the well-known global address space is realized by the programming model Shared Virtual Memory (SVM). This model is the base for the SVM-Fortran programming language. To achieve a better understanding of what happens in the memory during a SVM-Fortran program run, specific events are traced in a file. The analysis of the collected data is done with the visualization tool PARvis, which offers many well-designed features for this purpose. To allow the use of PARvis in the SVM-Fortran environment, this thesis first describes the development of a driver for the trace data given in a SDDF representation. The next step adds some new visualization options to PARvis to make SVM-Fortran specific information accessible to the user. Additional performance analysis is available through the source code related view, given for SVM-Fortran by the tool OPAL. Based on the common data base (trace data) conception and development of an interface between these two tools completes this thesis.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Bedeutung von Parallelrechnern bei der numerischen Lösung von Problemen aus den Bereichen Forschung, Wissenschaft und Technik, beispielsweise in den Gebieten Simulation oder Bildverarbeitung, hat während der letzten Jahre beständig zugenommen. Das Spektrum dieser Rechner reicht dabei von Graphik-Workstations mit mehreren Prozessoren und gemeinsamem Speicher bis hin zu massiv-parallelen Architekturen mit verteiltem Speicher. Gerade die Entwicklung von Mehrprozessorsystemen mit einem gemeinsamen Hauptspeicher hat die Verbreitung dieser Rechner sehr gefördert, da bei ihnen mit dem von klassischen Rechnerarchitekturen bekannten Speichermodell des globalen Adreßraumes gearbeitet werden kann. Aus technologischen und finanziellen Gründen werden jedoch massiv-parallele Systeme ausschließlich mit verteiltem Speicher gefertigt. Für diese Systeme wurde das Programmiermodell des Message-Passing entwickelt, bei dem der Programmierer durch entsprechende Anweisungen im Programm sicherstellen muß, daß die benötigten Daten zur richtigen Zeit am richtigen Ort (= Prozessor) zur Verarbeitung bereitstehen. Die Erstellung von Message-Passing-Programmen führt zwar meist zu sehr effizientem Code, die Programmierung selbst wird aber allgemein als sehr fehlerträchtig angesehen. Standardisierungsbemühungen *(Message Passing Interface, MPI)* führen in diesem Bereich zumindest zu einer Bibliothek von genormter Funktionalität.

Um effiziente Message-Passing-Programme schreiben zu können, ist häufig ein tiefgehendes Verständnis der Abläufe in einem Parallelrechner während der Programmausführung notwendig. Zur Performance-Analyse bietet sich, neben den klassischen Methoden wie Profiling oder Sampling, die Protokollierung von Ereignissen an. Da die bei der Programmausführung erzeugten Trace-Daten typischerweise einen Umfang von mehreren MByte aufweisen, sind neue Konzepte notwendig, um die diesen Daten inhärenten Informationen auszuwerten. Dazu wurde am Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich das X-basierte Visualisierungswerkzeug *PARvis* [Arn93, NaAr94] entwickelt. Mit ihm ist die Darstellung von Zustandswechseln auf vielfältige Weise möglich. Visualisierungstechniken wie Zooming erlauben es dem Anwender, die vorliegenden Trace-Daten schnell und präzise auszuwerten.

Daneben wurden in den letzten Jahren verschiedene Ansätze entwickelt, um die geschilderten Schwierigkeiten im Umgang mit dem Message-Passing-Programmiermodell zu umgehen. Sie alle zielen darauf, dem Programmierer soweit wie möglich die Verteilung der Daten abzunehmen. Zu den am weitesten entwickelten Ansätzen gehören die Programmiermodelle des

- *Compiler-Bridged Shared Memory* und des
- gemeinsamen virtuellen Adreßraumes *(Shared Virtual Memory)*.

So bildet bei dem Programmiermodell des *Compiler-Bridged Shared Memory (High Performance Fortran)* [HPF93, HPF94] der Compiler durch den Einsatz von Message-Passing-Befehlen den gemeinsamen Adreßraum auf den verteilten Speicher ab. Bei diesem Programmiermodell ist die Datenverteilung eines Programms entscheidend für seine Performance. Deshalb wird versucht, durch Methoden und Verfahren der Performance-Vorhersage für verschiedene Datenverteilungen a priori Informationen über den Programmablauf zu erhalten. Auf der Basis der gewonnenen Ergebnisse sollen dem Programmierer bzw. Compiler Hinweise für eine Optimierung der Datenverteilung gegeben werden. Dazu wird eine Analyse des Quelltextes an Hand verschiedener zugrundeliegender Modelle durchgeführt. Bisher ungelöste Probleme werfen hier irreguläre Strukturen auf, da die dynamisch während der Programmausführung auftretenden Änderungen in der Speichernutzung durch den Compiler kaum bzw. gar nicht berücksichtigt werden können. Das Vorgehen bei der Performance-Vorhersage steht im Gegensatz zur Performance-Analyse mit Hilfe von Trace-Daten, da hierbei eine Post Mortem-Analyse des Programmablaufes durchgeführt wird.

Zur Realisierung eines gemeinsamen virtuellen Adreßraumes *(Shared Virtual Memory)* bei Parallelrechnern mit verteiltem Speicher bieten sich ebenfalls mehrere Möglichkeiten. Sie unterscheiden sich dabei in der Ebene, in der ein „Speichermanager“ einen gemeinsamen Speicher realisiert. Bei den massiv-parallelen Systemen der Firma Cray (Cray T3D und deren Nachfolgemodell) wird der Speichermanager auf Hardware-Ebene durch eine Adreßlogik unterstützt, die nach Zugriff des Betriebssystems auf eine Speicheradresse für die Lokalisierung des Datums innerhalb des verteilten Speichers sorgt. Eine Hardware-Logik sorgt ebenfalls für die Kohärenz der Daten im Speicher. Diese Hardware-basierte Realisierung eines gemeinsamen virtuellen Speichers wird noch Gegenstand der Forschung am Zentralinstitut für Angewandte Mathematik sein, da ein entsprechender Rechner der Firma Cray angeschafft wird.

Der zur Zeit im Zentralinstitut für Angewandte Mathematik installierte Parallelrechner vom Typ Intel Paragon unterstützt das Speicherkonzept des *Shared Virtual Memory* mit einer Software-basierten Lösung durch den aus Mach bekannten externen Memory Manager XMM. Dieser wird in einer ersten Version einer Neuimplementierung durch den ASVM-Kernel zur Verfügung gestellt. Für dieses Speicherkonzept wurde die Programmiersprache *SVM-Fortran* [BGNP93] entwickelt und ein

Compiler für die Intel Paragon unter ASVM implementiert. In *SVM-Fortran* wird durch Direktiven die Arbeitsverteilung gesteuert, so daß die Daten während der Programmausführung an ihre „Plätze“ migrieren. Durch die Unterteilung des Speichers in Seiten kann ein *SVM-Fortran*-Programm nur dann effizient ablaufen, wenn die Zahl der während der Programmausführung aufgetretenen Seitenfehler möglichst gering ist. Daraus resultiert die Forderung, daß die Datenlokalität möglichst groß sein muß. Zur Untersuchung der Datenlokalität bei *SVM-Fortran*-Programmen ist das Quelltext-basierte Analyse-Werkzeug *OPAL* entwickelt worden [GKÖ95]. Es unterstützt die gezielte Instrumentierung des Quelltextes durch sogenannte Hook-Aufrufe. Zur Laufzeit erfolgt die Protokollierung von Ereignissen durch Verzweigung an den Hook-Aufrufen in den *SVM-Fortran Application Monitor (SAM)* [Özm95]. Entsprechend der Vorgaben einer Steuerungsdatei *(Trace Request-Datei)* werden Ereignisse verschiedener Typen generiert. *SAM* erzeugt Trace-Daten, die in dem maschinenunabhängigen SDDF-Format vorliegen. Durch die Auswertung dieser Trace-Daten in *PARvis* und *OPAL* sollen Erfahrungen gesammelt werden, um so dem Programmierer (oder dem Compiler) in Zukunft automatisch Hinweise für eine Verbesserung des Programms geben zu können. Dazu ist die Kopplung der beiden Werkzeuge *PARvis* und *OPAL* über eine gemeinsame Schnittstelle ein neuer Ansatz zur Performance-Analyse. Ein Ziel dieser Arbeit ist daher die Konzeption und Realisierung dieser gemeinsamen Schnittstelle.

PARvis ist im Zuge der Implementierung eines SVM-Simulators zur Untersuchung von Scheduling-Problemen um Komponenten zur Visualisierung von SVM-Aktivitäten erweitert worden [Mül94]. Mit der Bereitstellung von *SVM-Fortran* als einer Programmierschnittstelle zur Nutzung von Parallelrechnern mit gemeinsamem virtuellem Speicher entsteht nun die Notwendigkeit, die *PARvis*-Visualisierungsumgebung auch für die Trace-Daten von *SVM-Fortran*-Programmen zu nutzen.

Nach einer kurzen Einführung in die Thematik der Performance-Vorhersage in Kapitel 2 dieser Arbeit wird im Kapitel 3 die Realisierung eines Treibers für die im SDDF-Format vorliegenden Trace-Daten von *SVM-Fortran*-Programmen vorgestellt. Die durchgeführte Angleichung in der Funktionalität bei der Visualisierung dieser Trace-Daten an den *PARvis*-Standard sowie die statistische Auswertung bestimmter Ereignisse durch neu entwickelte *PARvis*-Komponenten wird in Kapitel 4 beschrieben. Kapitel 5 schließlich befaßt sich mit der Konzeption und Realisierung der Schnittstelle zwischen den beiden Werkzeugen *PARvis* und *OPAL*, um den Quelltextbezug bei der Performance-Analyse mit der Hilfe von Trace-Daten zu optimieren. Abgeschlossen wird diese Arbeit in Kapitel 6 mit einer Zusammenfassung der erzielten Ergebnisse und einem Ausblick auf mögliche und wünschenswerte Ergänzungen.

Kapitel 2

Performance-Vorhersage für Parallelrechner mit verteiltem Speicher

Die Performance von parallelen Programmen auf modernen Parallelrechnern mit verteiltem Speicher bleibt häufig deutlich hinter den Erwartungen zurück. Wegen der hochdynamischen Vorgänge in diesen Parallelrechnern (z.B. im Kommunikationsnetzwerk) sind Aussagen über den genauen Programmablauf nur schwer möglich. Die bisher von Parallelrechnern mit gemeinsamem Speicher bekannten und erprobten Analysemethoden lassen sich durch die grundsätzlich andere Rechnerarchitektur nicht mehr verwenden. Daher versucht man mit neuen Werkzeugen (z.B. *PARvis* [ADN95, NaAr93], *ParaGraph* [HeFi94] oder *Pablo* [RAMN92]) Ursachen für Performance-Engpässe zu entdecken. Basierend auf Trace-Daten, die den Programmablauf protokollieren, ist eine Post Mortem-Analyse beispielsweise der Kommunikationsmuster des Programms möglich. Ergänzt werden diese Werkzeuge durch neue Methoden zur Performance-Vorhersage. Ziel dieser Methoden ist es, den Programmierer bereits bei der Entwicklung seiner Programme auf mögliche Performance-Engpässe hinzuweisen und ihn nach Möglichkeit bei der Suche nach besseren Lösungen zu unterstützen. Schwerpunkt dabei ist die Suche nach einer optimalen Datenverteilung für ein paralleles Programm auf einem Parallelrechner mit verteiltem Speicher.

Der größte Teil von Programmen für Parallelrechner mit verteiltem Speicher wird heute unter Verwendung des Message-Passing-Programmiermodells erstellt. Unter *Message-Passing* versteht man das explizite Programmieren von Kommunikationsanweisungen für einzelne Prozessorelemente, was einer Programmierung auf niedriger Ebene entspricht. Benötigt ein Prozessorelement ein in seinem lokalen Speicher nicht vorhandenes Datum, so ist es die Aufgabe des Programmierers, dieses Datum durch entsprechende Kommunikationsbefehle über das Verbindungsnetzwerk aus dem lokalen Speicher eines anderen Prozessorelementes zu beschaffen [Lov93]. Der Programmierer muß also stets die aktuelle Verteilung der Daten auf den lokalen

Speichern kennen, damit die Anforderung eines Datums von dem angesprochenen Prozessorelement auch befriedigt werden kann.

Mit Hilfe des Message-Passing-Programmiermodells ist es möglich, daß parallele Programme auf Parallelrechnern mit verteiltem Speicher eine maximale Performance erreichen. Diese Performance wird aber mit dem Nachteil erkauft, daß das Erstellen von Message-Passing-Programmen eine schwierige Aufgabe ist [GeBe95]. So fehlt dem Programmierer zum Beispiel jede Unterstützung bei der Programmierung einer möglichst optimalen Datenverteilung für eine bestimmte Zielarchitektur [BFKK90]. Es gibt verschiedene Ansätze, den Programmierer von dieser Aufgabe zu befreien. Allen Ansätzen gemeinsam ist, daß ein globaler Adreßraum durch Software emuliert wird [GeBe95]. Geschieht dies zum Beispiel auf Compiler-Ebene, wird das so entstandene Programmiermodell als *Compiler-Bridged Shared Memory* bezeichnet [Lin94]. Der Compiler bildet dabei den logisch gemeinsamen Speicher durch den Einsatz von Message-Passing-Sprachprimitiven auf dem real verteilten Speicher ab.

Das Programmiermodell *Compiler-Bridged Shared Memory* wird unter anderem in *High Performance FORTRAN (HPF)* [HPF93, HPF94], *Fortran D* [FHKK90] und *Vienna Fortran* [ZBCM92] realisiert. In *Fortran D* und *Vienna Fortran* arbeitet der Programmierer in einer integrierten Entwicklungsumgebung, die neben Editor und Compiler auch ein Werkzeug zur Performance-Vorhersage bzw. -Schätzung enthält. Eine Beschreibung dieser Werkzeuge zur Performance-Vorhersage findet sich in Abschnitt 2.1.2 *(Fortran D)* und Abschnitt 2.1.3 *(Vienna Fortran)*.

Die Auswahl einer Datenverteilung ist der zentrale Schritt bei der datenparallelen Programmierung. Zum einen wird durch sie bestimmt, wieviel von der in dem Programm vorhandenen Datenparallelität ausgenutzt werden kann [BFKK91]. Zum anderen bestimmt sie die Zahl der Datenzugriffe auf den lokalen Speicher. Gelingt es, möglichst viele Datenzugriffe lokal auszuführen, ist eine gute Performance erreichbar, da der Unterschied in der Zugriffszeit zwischen lokalen und nicht-lokalen Daten um mehrere Größenordnungen differieren kann. Werkzeuge zur Performance-Vorhersage können verschiedene Datenverteilungen für ein paralleles Programm „ausprobieren“, ohne daß das komplette Programm übersetzt, gebunden und ausgeführt werden muß. Für den Compiler reicht dabei eine Aussage über die wahrscheinlich günstigste Datenverteilung aus. Ein Programmierer wird dagegen meist nicht an einem relativen, sondern an einem absoluten Wert für die zu erwartende Performance (Ausführungsdauer in Sekunden oder Speedup bei Verwendung von p Prozessoren) interessiert sein [Cro94]. Die Phase der Programmoptimierung kann mit Hilfe dieser Angaben erheblich verkürzt werden.

Compiler für neue Sprachkonzepte auf Parallelrechnern mit verteiltem Speicher erfordern also die Entwicklung von Werkzeugen zur Performance-Vorhersage. Wird die Datenverteilung nicht vom Compiler, sondern vom Programmierer vorgenommen, so profitiert auch er von den Hinweisen, die ihm ein solches Werkzeug geben kann.

2.1 Methoden der Performance-Vorhersage

Die Zahl der Methoden zur Performance-Vorhersage ist fast so groß wie die der einzelnen Implementierungen. Insgesamt sind drei große Gruppen zu unterscheiden:

- analytische Modelle
  - statistische Modelle
  - deterministische Modelle
  - Hybrid-Modelle
- Simulationssysteme und -sprachen
- Benchmark-Kernel-Libraries

Die größte Zahl von Implementierungen weist die Gruppe der analytischen Modelle auf; innerhalb dieser Gruppe wiederum die der statistischen Modelle. Statistische Größen haben einen erheblichen Einfluß auf die Ausführungszeiten von parallelen Programmen, deshalb ist eine Modellbildung mit Hilfe dieser Größen naheliegend.

Eine weitere mögliche Klassifizierung der verschiedenen Methoden findet sich in [Gem93]. Dort wird zwischen (deterministischer) Komplexitätsanalyse und (wahrscheinlichkeitstheoretischer) Warteschlangenanalyse unterschieden. Die Modellparameter lassen sich bei Verwendung des deterministischen Ansatzes leichter gewinnen als bei Verwendung des wahrscheinlichkeitstheoretischen Ansatzes [Adv93, Gem93]. Der deterministische Ansatz versagt allerdings bei der Modellierung der statistischen Vorgänge für die Vergabe von Betriebsmitteln. Dadurch wird der oben geschilderte Vorteil der einfachen Gewinnung der Modellparameter eingeschränkt.

Alle Modelle lassen sich in verschiedene Ebenen aufteilen. In [Adv93] sind zwei Ebenen genannt. In der unteren Ebene des Modells werden die individuellen Task-Ausführungszeiten berechnet. Hier werden systemspezifische Kosten wie für die Kommunikation (z.B. Verzögerungen durch konkurrierenden Zugriff auf das Verbindungsnetzwerk) einbezogen. Dies geschieht üblicherweise durch ein Warteschlangenmodell des Systems. Die obere Ebene beschreibt das Verhalten des Programms auf der Task-Ebene: Task-Ausführung und -Beendung sowie Prozeßsynchronisation. Aus den in der unteren Ebene berechneten individuellen Task-Ausführungszeiten werden in dieser Ebene die Ausführungszeiten des Programms berechnet. Bei der Entwicklung eines Modells verursacht die Berechnung dieser Zeiten meistens die größten Schwierigkeiten. Insbesondere bei den statistischen Modellen ist der Aufwand zur Berechnung der Kosten für die Synchronisation hoch.

Welche Faktoren beeinflussen die Performance eines parallelen Programms und damit auch die Performance-Vorhersage? In [Adv93] wird dazu die reine Rechenarbeit auf dem Prozessor, die Kommunikation zwischen Prozessen, die Konkurrenz um gemeinsame Betriebsmittel für Rechenarbeit und Kommunikation sowie Verzögerungen durch Synchronisationen gezählt, die dann auftreten, wenn ein Prozeß auf einen

anderen Prozeß an einem Synchronisationspunkt warten muß. Auf Rechnerarchitekturen mit physikalisch verteiltem, logisch jedoch gemeinsamem Speicher *(Shared Virtual Memory)*, kommen weitere Verzögerungen hinzu, bedingt durch das „Wandern“ von Speicherseiten von einem Prozessor zu einem anderen.

Ein weiterer Ansatz, die Einflüsse auf die Performance von parallelen Programmen zu charakterisieren, ist eine hierarchische Einteilung in Einflußfaktoren vom parallelen Programm bis hin zum Parallelrechner, auf dem das Programm ausgeführt wird [Fah93]:

- das parallele Programm
- der parallelisierende Source-to-Source Compiler
- der Compiler des Zielrechners
- das Betriebssystem des Zielrechners
- die Architektur des Zielrechners

Der Entwurf eines Werkzeuges zur Performance-Vorhersage sollte unter der Beachtung der folgenden Aspekte erfolgen [Fah93]:

- Welche Art von Performance-Informationen soll geschätzt werden:
  - Abschätzung der Ausführungszeit eines Programms
  - Abschätzung einer Kostenfunktion für synchrone Kommunikationsbefehle
  - ...
- Welche Ebenen der oben erwähnten hierarchischen Einteilung sollen modelliert werden?
- Welche Methoden sind zur Performance-Vorhersage einsetzbar?

Aber nicht nur an ein Werkzeug zur Performance-Vorhersage sind Anforderungen zu stellen. Auch die verwendeten Methoden für die Vorhersage müssen bestimmte Bedingungen erfüllen [CBLM92]:

- Die Methoden sollen bereits Vorhersagen durch Messungen an unvollständigen Programmen wie auch an seriell ausgeführten Programmteilen liefern.
- Die Methoden sind einfach genug, um in ein Werkzeug zur Performance-Vorhersage eingebunden zu werden.
- Die Methoden liefern bei möglichst vielen verschiedenen Programmtypen möglichst genaue Vorhersagen.

Gerade der letzte Punkt ist schwer zu erfüllen. Je genauer die Vorhersage werden soll, desto einschränkender sind derzeit die Bedingungen an das zu untersuchende Programm.

In den folgenden Abschnitten werden einige erfolgversprechende Methoden zur Performance-Vorhersage ausführlicher vorgestellt.

2.1.1 Der Benchmark-Kernel-Library-Ansatz

Die Laufzeit eines Programms wird durch dynamische und statische Programmparameter bestimmt. Wenn diese Parameter bekannt sind, kann darauf aufbauend eine Performance-Vorhersage für das Programm durchgeführt werden.

In [Fah94] werden die dynamischen Programmparameter eines parallelen Programms während eines sequentiellen Programmlaufes bestimmt. Mit Hilfe des Werkzeuges *Weight Finder* wird ein Ausführungsprofil erstellt. Der *Weight Finder* ist ein verbesserter Fortran-Profiler und ist Bestandteil des *Vienna Fortran Compiling Systems (VFCS)*. Der Profiler ermittelt Werte für Sprungwahrscheinlichkeiten, obere Schleifengrenzen usw. Da nach dem *Single Program Multiple Data (SPMD)* Modell [Kar87] der Kontrollfluß auf allen Prozessoren während eines parallelen Programmlaufes gleich ist, lassen sich die ermittelten Werte auf das parallele Programm übertragen.

Die statischen Programmparameter erhält man, indem das parallele Programm von einem Werkzeug nach bestimmten Code-Mustern durchsucht wird. Diese Code-Muster repräsentieren Programmeinheiten („Programm-Kerne"), deren sequentielle Ausführungszeiten in einem Benchmark-Verfahren gemessen werden. Aus den gewonnenen Ergebnissen wird eine Benchmark-Kernel-Library aufgebaut. Für jeden von dem Werkzeug in dem parallelen Programm gefundenen Programm-Kern werden die in der Library gespeicherten Ausführungszeiten aufsummiert.

Die Auswahl der in der Library gespeicherten Programm-Kerne und die Fähigkeit des Werkzeuges, diese in dem parallelen Programm wiederzuerkennen, bestimmen die Genauigkeit der Performance-Vorhersage (siehe dazu auch [NaLi93]).

Abbildung 2.1 zeigt den prinzipiellen Ablauf der Performance-Vorhersage mit einer Benchmark-Kernel-Library.

Programm-Kerne, nach denen das parallele Programm durchsucht wird, werden bei [Fah94] in vier Klassen eingeteilt:

1. Einfache Operationen: Grundrechenarten, logische Operationen, einfache Zugriffe auf Felder (z.B. `A(I+1)`) usw.
2. Einfache Befehle: z.B. `DO`-Schleifenköpfe, `SUBROUTINE`- und `FUNCTION`-Aufrufe, Bedingungs-Anweisungen, Zuweisungen, `GOTO`-Anweisungen, atomare Kommunikationsanweisungen (`send` und `receive`).

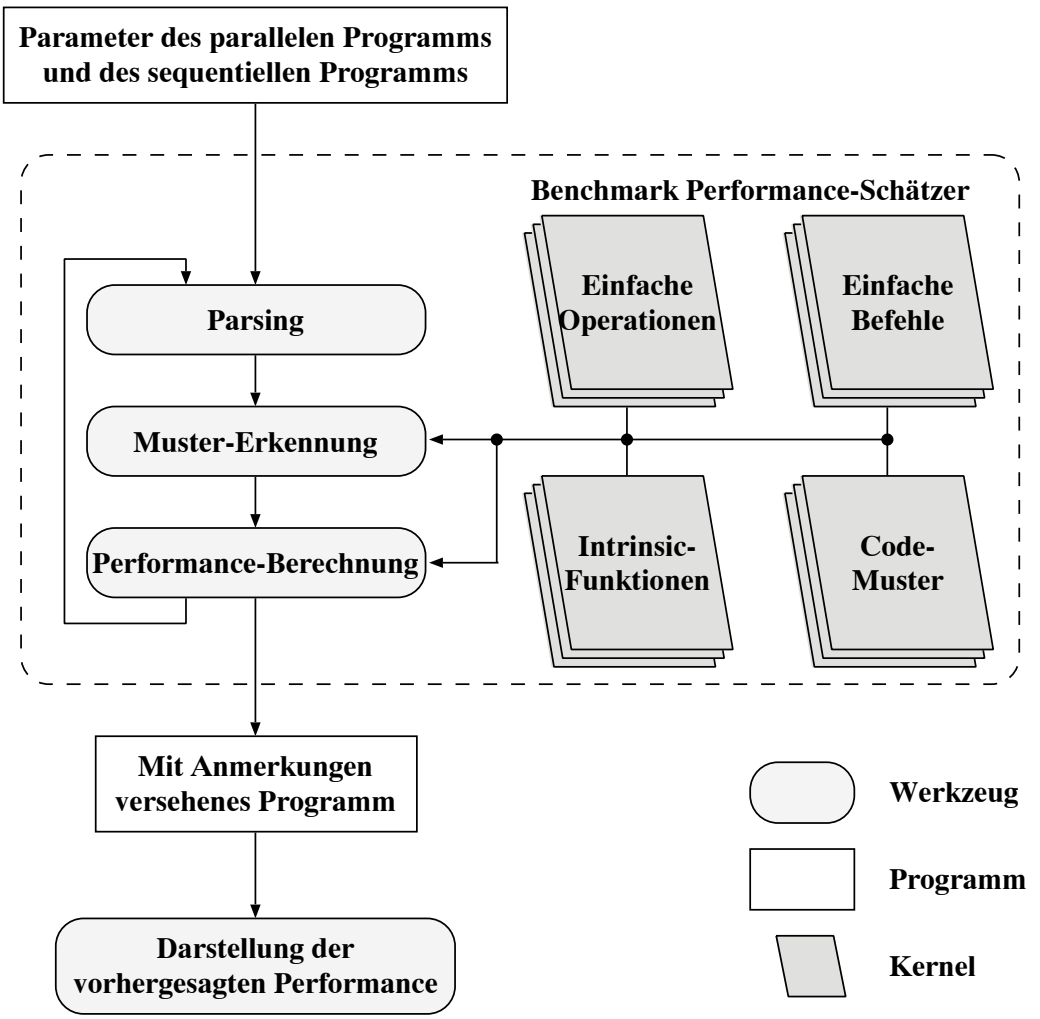


Abbildung 2.1: Performance-Vorhersage mit Benchmark-Kernel-Library

3. Intrinsic-Funktionen wie `SIN`, `COS`, `MOD`, `LOG` usw. Neben globalen Kommunikationsanweisungen (z.B. `broadcast`) sind in dieser Klasse sowohl implizite (`MIN`, `MAX`, `INDEX` usw.) als auch maschinenspezifische Reduktionsfunktionen zu finden.

4. Code-Muster: Hier werden standardisierte Code-Muster abgespeichert, die leicht innerhalb eines Programms wiedererkannt werden können. Dazu zählen grundlegende Operationen der linearen Algebra (Matrixmultiplikation, Matrixinversion, Determinantenberechnung usw.) und oft genutzte Operationen auf Matrizen (z.B. Jacobi-Relaxation, LU-Zerlegung, Gauß-Jordan-Verfahren).

Darüber hinaus ist jede Klasse in maschinenabhängige und -unabhängige Einheiten unterteilt [Fah94].

Zur Bewertung der Leistungsfähigkeit des Benchmark-Kernel-Library Ansatzes zieht Fahringer folgende Kriterien heran:

- Zeitaufwand zur Erstellung und Pflege eines Werkzeuges zur Performance-Vorhersage mit der beschriebenen Library als Basis.
- Die Übertragbarkeit des Ansatzes auf andere Architekturen.
- Der mit den Messungen verbundene Aufwand.
- Der lokale und globale Einfluß der Architektur und des Compilers des Zielsystems auf die Performance.
- Das Erkennen der im Programm vorhandenen Code-Muster.
- Die erreichbare Genauigkeit bei der Performance-Vorhersage.

In [Fah94] sind die Ergebnisse der Performance-Vorhersage mit Hilfe der oben beschriebenen Methode der Benchmark-Kernel-Library dargestellt. Es wurde jeweils für eine SIMD-Architektur *(MasPar MP-1)* und eine MIMD-Architektur *(Intel iPSC/860)* eine Kernel-Library erstellt. Die Untersuchungen wurden an der *Intel iPSC/860* mit einem Prototyp eines Werkzeuges zur Performance-Vorhersage (Auswertung dynamischer und statischer Programmparameter) durchgeführt.

Fahringer widerspricht in seiner Arbeit deutlich der in [BFKK91] gemachten Aussage, daß das Erzeugen und die Pflege einer Kernel-Library ohne großen Zeitaufwand durchgeführt werden könne. Er vergleicht den Aufwand für die Erstellung der vorgestellten Kernel-Library (2 Mann-Jahre) mit dem für sein Werkzeug zur Performance-Vorhersage P^3T, siehe Abschnitt 2.1.3, (1,5 Mann-Jahre). Neben dem höheren zeitlichen Aufwand bei der Erstellung traten bei der Kernel-Library noch weitere Probleme auf:

- Alle Eigenheiten der zugrundeliegenden Architektur und des Compilers müssen vom Programmierer einer Kernel-Library durch viele Variationen einzelner Programm-Kernels widergespiegelt werden.
- Ein großer Teil der verwendeten Einheiten ist nicht auf andere Architekturen übertragbar, zum Teil nicht einmal zwischen verschiedenen Compiler-Versionen auf einer Architektur.
- Die Ausführungszeiten für die Einheiten werden nicht innerhalb eines Programmkontextes gemessen. Daher werden Veränderungen durch gegenseitige Beeinflussung innerhalb des parallelen Programms nicht richtig gewichtet.

Der Prototyp des Werkzeuges zur Performance-Vorhersage auf der *Intel iPSC/860* wurde mit dem „Red-Black Checkerboard"-Algorithmus (punktweises Relaxationsverfahren) [PFTV88] getestet. Das Werkzeug liefert wegen seines Prototyp-Charakters nur Ergebnisse für ein sequentielles Programm auf einem einzigen Prozessor. Für eine Problemgröße von N = 1024 ergab sich eine Abweichung des Schätzwertes für die sequentielle Ausführungszeit vom tatsächlich gemessenen Wert von 27 %. Fahringer erklärt diese deutliche Abweichung mit der ungenauen Modellierung des prozessorinternen Caches und den Auswirkungen des CPU-Pipelinings.

2.1.2 Die Benchmark-Kernel-Library für Fortran D

In *Fortran D* gibt der Programmierer durch Direktiven die Verteilung der Daten vor. Der Grad der Parallelisierung und damit auch die erzielbare Performance ist so direkt von dem Wissen und der Erfahrung des Programmierers abhängig. Um von diesem „Können" unabhängiger zu werden, soll er bei der Wahl einer geeigneten Datenverteilung Unterstützung durch ein Werkzeug erhalten. Dieses Werkzeug untersucht verschiedene vom Programmierer vorgegebene Datenverteilungen in bezug auf ihre Auswirkungen auf die Performance. Als Ergebnis erhält der Programmierer eine Schätzung, welche Datenverteilung unter den gegebenen Randbedingungen (Zahl der zur Verfügung stehenden Prozessoren, Größe der Eingabedatenfelder) die wahrscheinlich beste Performance erzielen wird. Für diese Auswahl reicht eine Schätzung aus, die eine relativ „beste" Datenverteilung findet. Wichtig ist, daß der „Crossover-Punkt", bei dem eine Datenverteilung besser als eine andere wird, möglichst genau erkennbar ist. In Zukunft soll dieses Werkzeug selbständig eine optimale Datenverteilung finden, denn erst dann ist eine Unabhängigkeit von dem „Können" des Programmierers erreicht. Die Informationen über diese Datenverteilung werden dann an den Compiler weitergegeben [HKKK91]. In Abbildung 2.2 ist die Einbettung des Werkzeuges zur Performance-Schätzung in die Programmierumgebung des *Fortran D*-Systems dargestellt.

Zur Performance-Vorhersage in *Fortran D* wurde die Methode der Benchmark-Kernel-Library ausgewählt, da sich mit ihr nach [BFKK91] genauer als mit einem theoretischen Modell bestimmen läßt, wann eine Datenverteilung hinsichtlich der Performance des Programms besser als eine andere wird. Außerdem gestaltete es sich als schwierig, mit einem theoretischen Modell das experimentell beobachtete Verhalten eines Programms für verschiedene Datenverteilungen vorherzusagen.

Die einzelnen Programm-Kerne zur Performance-Vorhersage werden in einer Trainings-Menge zusammengefaßt. Die Gewinnung der zur Schätzung notwendigen Daten erfolgt in zwei Initialisierungsschritten, die auf jeder Zielarchitektur einmal auszuführen sind (siehe Abbildung 2.3).

Zunächst wird die Datei `raw.data` erzeugt, in der mehrere Ausführungszeiten von arithmetischen und Kontrollfluß-Operationen gespeichert werden. Durchschnittliche Zeiten für verschiedene Kommunikationsmuster bei einer sich verändernden Zahl der verwendeten Prozessoren oder der Größe der Eingabedatenfelder werden ebenfalls

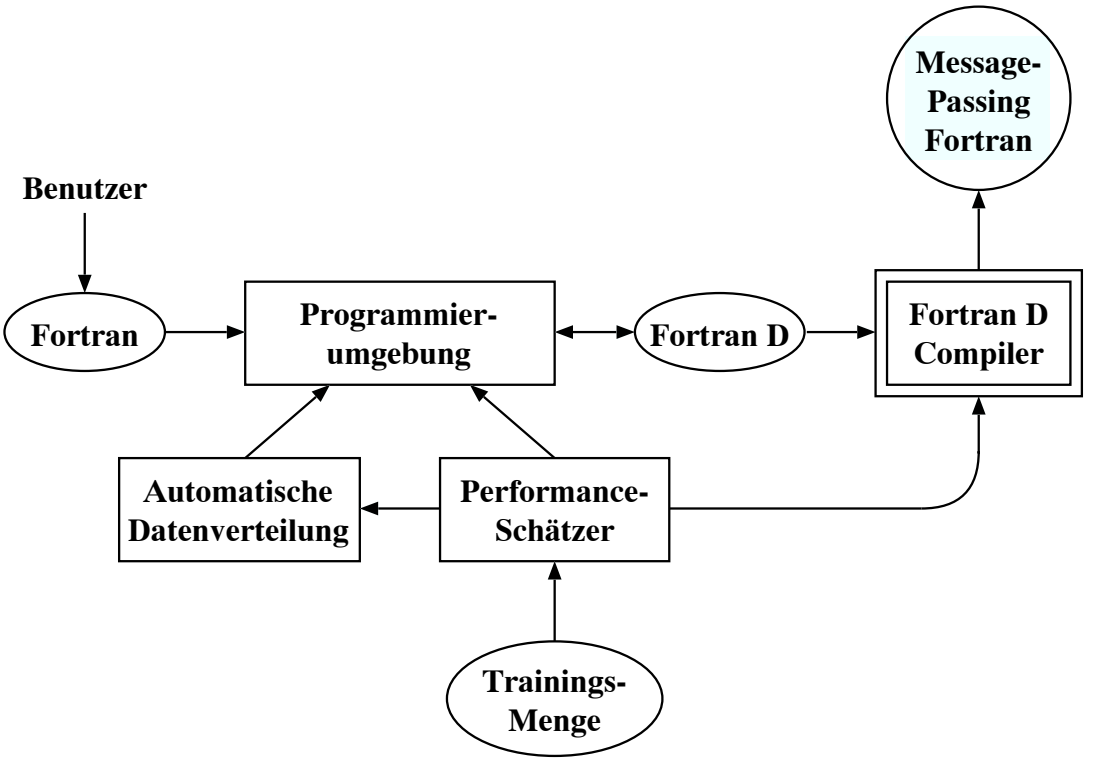


Abbildung 2.2: *Fortran D*-Programmierumgebung

in dieser Datei gespeichert. Durch Komprimierung der Daten entsteht die Datei `perf.data`. Die Komprimierung wird erreicht, indem die einzelnen Werte für die gespeicherten Kommunikationszeiten durch eine Funktion angenähert werden. Es werden also nicht mehr alle Werte gespeichert, sondern nur noch die sie beschreibende Funktion. Von den Ausführungszeiten der arithmetischen und Kontrollfluß-Operationen wird für jede Operation nur noch ein Mittelwert gespeichert. Die Schätzung der Kommunikations- und Ausführungszeiten geschieht durch einen Algorithmus, der sich auf die in der Datei `perf.data` gespeicherten Funktionen und Mittelwerte abstützt. Dieser Algorithmus eignet sich am besten zum Vergleich zweier gegebener Datenverteilungen, er liefert also eine relative Performance-Schätzung [BFKK91].

Eine Prototyp-Implementierung des Werkzeuges zur Performance-Schätzung in *Fortran D* zeigte, daß die Schätzung bei irregulären Problemen[1] nicht einwandfrei funktionierte. Deswegen erfolgte eine Beschränkung auf mehrdimensionale reguläre Strukturen. In diesen Fällen bewegte sich der Schätzfehler zwischen 5 % und

[1] Irreguläre Probleme bzw. Strukturen finden sich bei der Methode der Finiten Elemente. Die dort auftretenden unregelmäßigen Gitterstrukturen werden auf eindimensionale Felder abgebildet, indem die beliebig durchnumerierten Gitterpunkte als Feldindizes dienen. In einer Liste werden die Nummern/Feldindizes der nächsten Nachbarn eines Gitterpunktes gespeichert. Der Zugriff auf diese Nachbarn (Stencil-Operation) erfolgt über die Liste, da sich die Feldindizes auf Grund der unregelmäßigen Gitterstruktur nicht mehr in der Form `i+1`, ... berechnen lassen.

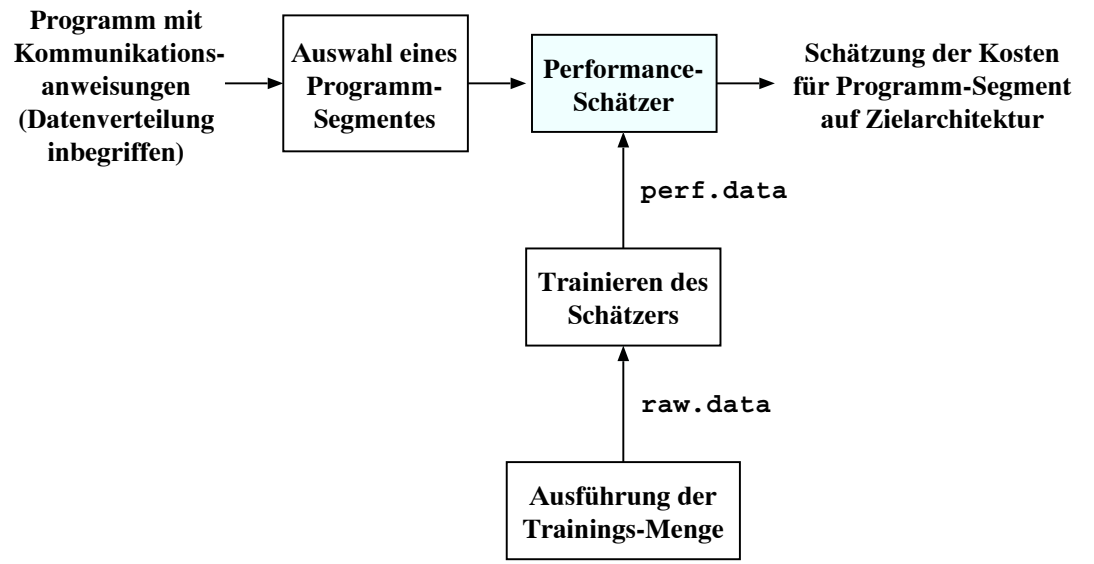


Abbildung 2.3: Performance-Schätzung in *Fortran D*

10 %. Für die Entscheidung zwischen einer Block- und einer Spaltenverteilung der Daten wurde der Crossover-Punkt hinreichend genau erkannt [BFKK91].

Mit Hilfe der Programm-Kerne wird der Performance-Schätzer trainiert. Bei Änderungen der Hardware oder der Software (z.B. des Compilers) muß er neu trainiert werden. In Zukunft soll das Wissen, welches in der Trainings-Menge gespeichert ist, durch ein neuronales Netz dargestellt werden [BFKK91].

2.1.3 Das Parameter-basierte Modell für Vienna Fortran

Im Rahmen des *Vienna Fortran Compiling Systems (VFCS)* wurde das *Parameter based Performance Prediction Tool (P^3T)* entwickelt [Fah93]. Die Performance-Vorhersage in diesem System erfolgt an Hand von Parametern, die mit Hilfe des *Weight Finders* (siehe Abschnitt 2.1.1) während eines sequentiellen Programmlaufes gewonnen werden. Basierend auf diesen Parametern und dem parallelen Programm erzeugt P^3T während der Übersetzung einen neuen Satz von Parametern für das parallele Programm, der

- die Arbeitsverteilung des parallelen Programms,
- den Kommunikations-Overhead und
- die Lokalität der Daten

beschreibt. Diese Parameter können für bestimmte einzelne Anweisungen, `DO`-Schleifen und Prozeduren oder das gesamte Programm berechnet werden. Bei Programmen mit irregulären Kontrollstrukturen oder indirekten Feldzugriffen ist es

nicht möglich, absolute Werte bei der Performance-Vorhersage anzugeben. Es ist jedoch sinnvoll, einen Vergleich zwischen verschiedenen Programmversionen anzustellen, um so zu einer relativen Performance-Vorhersage zu gelangen. Auf diese Weise kann eine „beste“ Programmversion gefunden werden. Das Prinzip der Performance-Vorhersage mit P^3T verdeutlicht Abbildung 2.4.

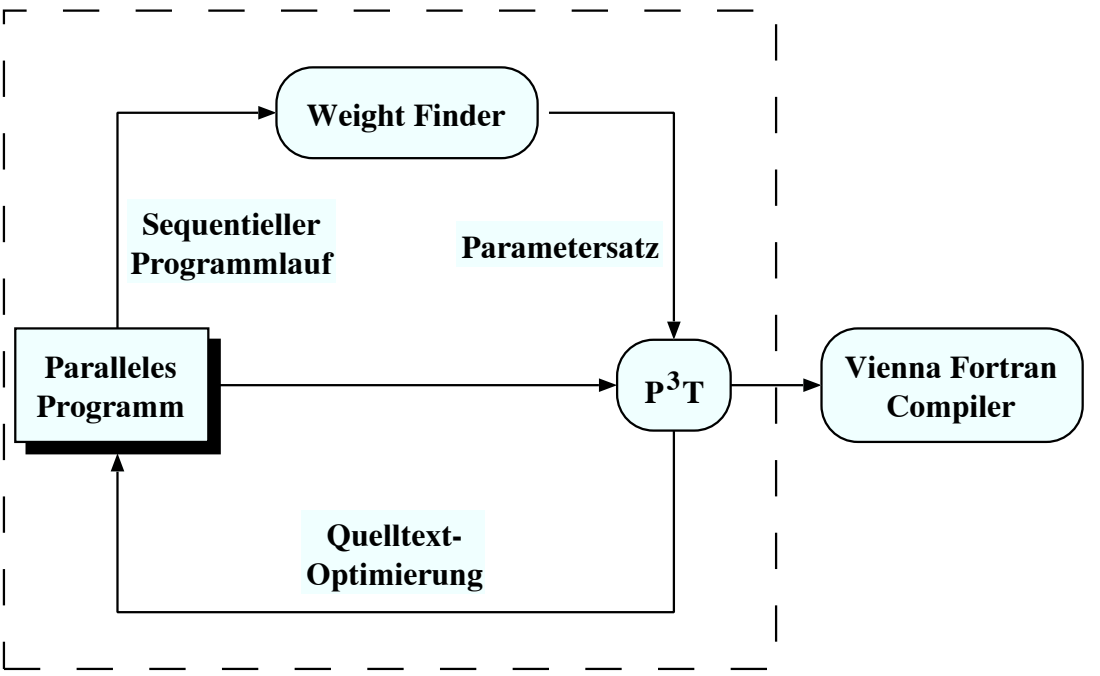


Abbildung 2.4: *Parameter based Performance Prediction Tool P^3T*

Während einer Trainings-Phase wird die Wichtigkeit der einzelnen Modellparameter in P^3T für die jeweilige Zielarchitektur bestimmt. Kriterium für die Wichtigkeit eines Modellparameters ist, wie sehr er die Performance des parallelen Programms auf der Zielarchitektur beeinflußt. Die Bestimmung der Modellparameter erfolgt durch Programm-Kerne, die empirisch ausgewertet werden. Je mehr verschiedene Datenverteilungen und Programmsegmente ausgewertet sind, desto besser ist das Werkzeug trainiert, aber desto größer ist auch die Zahl der auszuwertenden Programm-Kerne.

Durch die berechneten Parameter für das parallele Programm können die Teile des Programms ermittelt werden, welche für eine Performance-Steigerung zu verbessern sind. Sie liefern darüber hinaus Hinweise darauf wie der entsprechende Programmteil zu verändern ist. Die Berechnung der Parameter erfolgt durch eine Reihe von neuen Algorithmen. Iterationsräume von Schleifen lassen sich in der Ebene durch konvexe Polyeder darstellen. Einer dieser neuen Algorithmen bestimmt die Schnittmenge zwischen beliebigen n-dimensionalen konvexen Polyedern und Unterräumen. Für die Bestimmung der Lokalität der Daten wird auf ein Hybrid-Modell aus bekannten Ansätzen und in [Fah93] entwickelten Verbesserungen zurückgegriffen. Viele

der in diesem Werkzeug angewandten Techniken lassen sich auch zur Performance-Vorhersage bei anderen Programmiersprachen wie zum Beispiel C verwenden.

Getestet wurde P^3T mit vier unterschiedlich umfangreichen realen Programmen, darunter ein Programm zur Wettervorhersage. Diese Programme zeichneten sich durch eine sehr hohe Datenparallelität aus. Durch die Wahl einer passenden Datenverteilung sollte die den Programmen inhärente Datenverteilung möglichst gut ausgenutzt werden. P^3T gelang es, eine in diesem Sinne optimale Datenverteilung zu finden, indem Transformationen innerhalb des Programm-Codes durchgeführt wurden. Bei allen getesteten Programmen lagen die ermittelten Modellparameter sehr nah an den tatsächlich gemessenen Parametern. Die Bestimmung des Crossover-Punktes (siehe Abschnitt 2.1.2) gelang bei einem Teil der untersuchten Programme mit hinreichender Genauigkeit.

Leider waren die in [Fah93] und in [Fah94] untersuchten Programme nicht identisch, so daß eine vergleichende Bewertung des Werkzeuges P^3T und der Benchmark-Kernel-Library nicht möglich ist.

2.1.4 Das Hybrid-Modell

In [Adv93] wird zunächst der Einfluß nicht-deterministischer Größen auf die Ausführungszeit eines Programms untersucht. Ein analytisches Modell des Programmverhaltens, kombiniert mit Messungen an Programmen, zeigte, daß die Ausführungszeiten von Prozessen zwischen Synchronisationspunkten nur in sehr geringem Maße von nicht-deterministischen Faktoren (z.B. Verteilung von Betriebsmitteln oder Kommunikation auf dem Verbindungsnetzwerk) abhängen. Es zeigte sich, daß sich die Abhängigkeit der Ausführungszeiten von diesen Faktoren auch bei einer sehr hohen Systemauslastung nicht vergrößerte. Nach [Adv93] sollen diese Ergebnisse bei Programmen ähnlicher Granularität auch bei neuen Parallelrechnern mit verteiltem Speicher Bestand haben.

Aus diesen Ergebnissen leitet Adve sein deterministisches Modell zur Performance-Vorhersage ab. In die deterministische Repräsentation sind nicht nur die mittleren Ausführungszeiten von Tasks eingeschlossen, sondern auch die Kommunikation. Untersuchungen an Programmen für gemeinsamen Speicher hatten ergeben, daß der Einfluß von Verzögerungen durch die Kommunikation auf die Ausführungsdauer des Programms zwischen Synchronisationspunkten eine vernachlässigbare Varianz aufwies. Da dieses Ergebnis unabhängig von der Auslastung des Verbindungsnetzwerkes war, konnte die Kommunikation in dem Hybrid-Modell deterministisch dargestellt werden. Die Vergabe von gemeinsam genutzten Betriebsmitteln wird durch ein statistisches Modell dargestellt, welches den deterministischen Ansatz ergänzt.

Zur Überprüfung seines Modells verwendete Adve mehrere reale Programme mit typischen Eingabedatensätzen. Die Abweichungen zwischen tatsächlichen und vorhergesagten Ausführungszeiten waren in allen Fällen kleiner als 10 %, meistens lagen die Abweichungen bei ca. 2 % und 3 %. Die Messungen wurden auf Parallelrechnern

mit gemeinsamem Speicher und mit verteiltem Speicher durchgeführt. Bei dem System mit verteiltem Speicher wurde der gemeinsame Speicher durch eine Software emuliert. Als reale Programme verwendete Adve vier Programme aus der *Splash Suite* [SWG92] und zwei weitere parallele Programme, die als Produktions-Codes eingesetzt werden.

2.1.5 Der Communication-to-Computation-Verhältnis-Ansatz

Eine Methode zur Performance-Vorhersage, die auf der Auswertung des Communication-to-Computation-Verhältnisses beruht, ist in [CBLM92] beschrieben. Als Ergebnis erhält man eine Vorhersage über den Speedup des Programms bei Erhöhung der Zahl verwendeter Prozessoren. Ein wichtiges Ziel bei der Entwicklung dieser Methode war es, Daten aus der Performance-Vorhersage so früh wie möglich für die Programmentwicklung zu erhalten. Aus diesem Grund kann das Verfahren bereits Ergebnisse für unvollständige Programme ermitteln. Diese Methode ist sowohl für Architekturen mit verteiltem Speicher als auch für solche mit gemeinsamem Speicher geeignet.

Die Kosten für „Communication“ (C_1) sind in [CBLM92] definiert als Zeit, die ein Prozessor inaktiv oder mit busy waiting infolge von Datenbewegungen an anderen Stellen des Systems verbringt. Kosten für „Computation“ (C_2) werden als Zeit definiert, die sich ein Prozessor im aktiven Status befindet. Dabei kann er entweder Berechnungen durchführen oder aktiv auf den Zugriff auf eine gemeinsame Variable warten. Das C_1/C_2-Verhältnis für die Ausführung eines bestimmten Programmteiles auf einer bestimmten Maschine ist also durch die Kommunikations-Kosten („Communication“) dividiert durch die Kosten für die Berechnung („Computation“) bestimmt.

Zur Festlegung des Kommunikationsverhaltens des Parallelrechners wird zunächst ein hypothetisches Modell seiner Kommunikationsstruktur erstellt. Dieses beruht auf Annahmen zur Konkurrenzsituation um bestimmte Kommunikationsbetriebsmittel. Das Modell wird dann an Hand von synthetischen Programmen, deren Kommunikationsverhalten im System einfach analysiert werden kann, verifiziert. Ist die Genauigkeit der vorhergesagten Werte nicht ausreichend, wird das Modell durch Annahme zusätzlicher Konkurrenzsituationen um Kommunikationsbetriebsmittel verfeinert. So kann das Kommunikationsverhalten des Systems ohne ein a priori Wissen über seine Kommunikationsstruktur beschrieben werden.

Die aus den Hardware-Benchmarks gewonnenen Modelle werden dann mit Messungen an Anwendungen kombiniert, um so zu einer C_1/C_2-basierten Performance-Vorhersage zu gelangen. Die Überprüfung des theoretischen Ansatzes erfolgte an zwei Vertretern der beiden erwähnten Speicher-Architekturen. Dazu wurde neben einem synthetischen Programm ein Programmstück, welches eine Gaußelimination

durchführt, verwendet. Bei der Gaußelimination ist eine gute Übereinstimmung zwischen vorhergesagtem und erzieltem Speedup erreicht worden.

2.1.6 Der Simulationsansatz

Ein zu den bisher vorgestellten gänzlich anderer Ansatz zur Performance-Vorhersage ist in [Gem93] beschrieben. Mit Hilfe der Simulationssprache *Pamela* (PerformAnce ModEling LAnguage) wird das parallele Programm und die Architektur des Zielrechners simuliert. Zu der Simulationssprache gehört ein Performance-Kalkül, welches als „serialization analysis" bezeichnet wird. Das Kalkül basiert auf der statischen Programmanalyse und schließt zusätzlich eine Näherungsanalyse für den konkurrierenden Zugriff auf gemeinsame Ressourcen ein.

Pamela ist eine C-artige Programmiersprache. Sie enthält Ergänzungen zur Behandlung von konkurrierenden Prozessen und der (virtuellen) Zeit. Die Einführung einer Simulationszeit ermöglicht die Modellierung von Ausführungszeiten in Situationen, in denen Prozesse eine bestimmte Zahl von (virtuellen) Zeiteinheiten warten müssen. In *Pamela* wird nicht zwischen Architekturen mit gemeinsamem oder verteiltem Speicher unterschieden. Ein System wird in dieser Simulationssprache immer mit gemeinsamem Speicher dargestellt. Die Abbildung von Systemen mit verteiltem Speicher auf solche mit gemeinsamem Speicher wird durch ein Message-Passing-Interface auf der Ebene des gemeinsamen Speichers geleistet. Durch den Charakter einer imperativen Programmiersprache[2] können in *Pamela* formulierte Modelle direkt in einem Simulator ausgeführt und überprüft werden.

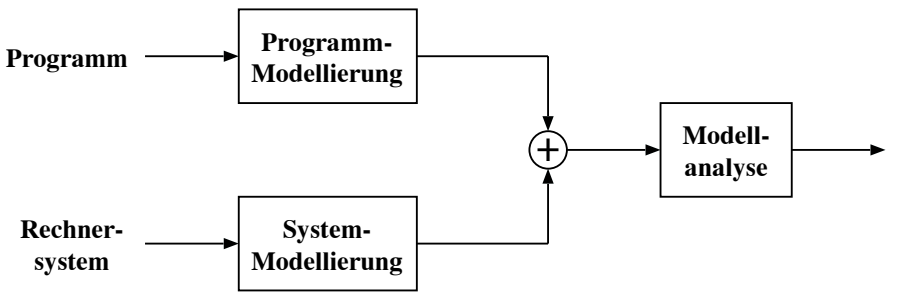


Abbildung 2.5: Modellbildung mit *Pamela*

Die Modellbildung mit *Pamela* erfolgt in zwei Schritten (siehe Abbildung 2.5). Zunächst wird ein Modell des Programms erstellt. Bei *Pamela* ist die Modellierung eines Programms aus einer imperativen Programmiersprache einfach, da beide Sprachen auf demselben Programmierparadigma beruhen. Auch *Pamela* kann Parameter eines Programms, die zur Compile-Zeit nicht festliegen (Sprungwahrscheinlichkeiten, Grenzen für Schleifendurchläufe usw.), nicht ermitteln. Zur Lösung dieses Problems werden drei Möglichkeiten vorgesehen, wie diese Parameter dem Modell zur Verfügung gestellt werden können:

[2]Bei diesen Sprachen besteht ein Programm aus einer Folge von Befehlen an den Computer. Wesentlich ist das Variablenkonzept: Eingabewerte werden in Variablen gespeichert und weiterverarbeitet [Dud88].

- Einsetzen von Standard-Werten
- Erfragen der Parameter zur Compile-Zeit von dem Anwender
- Direktiven innerhalb des Modellierungs-Codes

Im zweiten Schritt wird die Systemarchitektur modelliert. Dies kann mit beliebiger Genauigkeit geschehen, im Extremfall bis hinunter auf die Schaltkreis-Ebene. Ein solcher Feinheitsgrad bei der Modellierung führt aber nicht zwangsläufig zu genaueren Ergebnissen bei der Performance-Vorhersage. So erhält man durch eine schaltkreisgetreue Modellierung eines Prozessorcaches keine genaueren Ergebnisse als durch eine Simulation seiner Caching-Strategie. Letzteres zu simulieren ist sicherlich bedeutend weniger aufwendig. Wie eine Komponente des Systems modelliert wird, hängt weiterhin davon ab, wie sie die auf ihr ablaufenden Prozesse beeinflußt. Zentrale Komponente des Systemmodells in *Pamela* ist der logisch gemeinsame Speicher. Diese Ressource wird global über ein Verbindungsnetzwerk (mit oder ohne Cache-Speicher) verfügbar gemacht.

2.2 Bewertung der vorgestellten Methoden

Am weitesten fortgeschritten ist sicherlich das hier vorgestellte Hybrid-Modell, die Benchmark-Kernel-Libraries und das Parameter-basierte Modell, da sie bereits das Stadium von Prototypen-Implementierungen erreicht haben. Allerdings können bis jetzt, trotz der Beschränkung auf Fortran als Programmiersprache, nur einfache Kontrollstrukturen und reguläre Datenverteilungen modelliert werden. Durch die erzielte Vorhersagegenauigkeit bei realen Programmen ist der Einsatz mit den erwähnten Einschränkungen als Werkzeug zur Performance-Vorhersage möglich. Neben dem eigentlichen Zweck der Modelle, der relativen bzw. absoluten Performance-Vorhersage, sind bei der Entwicklung und Verifikation wichtige Erkenntnisse über das Verhalten von parallelen Programmen auf verschiedenen Architekturen gewonnen worden.

Das Hybrid-Modell erlaubt die Performance-Vorhersage für parallele Programme mit beliebigen Task-Graphen und statischen, halb-statischen und dynamischen Scheduling-Strategien.

Das Werkzeug zur Performance-Vorhersage P^3T und der Performance-Schätzer für *Fortran D* sollen wichtige Informationen für den Compiler zur Übersetzung des parallelen Programms liefern. Theoretische Überlegungen über gute oder schlechte Datenverteilungen können allerdings durch die Umsetzung des Compilers widerlegt werden [HKKK91]. Noch nicht gelöst ist die Vorhersage von Ausführungszeiten für irreguläre Programmstrukturen. Dazu gehören auch `FORALL`-Schleifen (u.a. in *Vienna Fortran*), bei denen die Arbeitsverteilung zur Laufzeit festgelegt wird.

Die größte Schwierigkeit bei der Benchmark-Methode ist die Auswahl repräsentativer Programm-Kerne und deren Wiedererkennung in Programmen [HKKK91]. Die

Benchmark-Methode ist trotz der in Abschnitt 2.1.1 beschriebenen Schwierigkeiten als ein zweckmäßiges Verfahren zur Performance-Vorhersage zu betrachten.

Der Communication-to-Computation-Verhältnis-Ansatz liefert Ergebnisse zur Vorhersage des Speedup's beim Übergang von einem auf mehrere Prozessoren. Da die gemachten Untersuchungen sich auf synthetische Programme und Programmteile beschränken, ist es fraglich, welche Ergebnisse bei der Anwendung auf reale Programme erzielbar sind.

Der Simulationsansatz mit *Pamela* ist sehr vielseitig. So ist *Pamela* als Programmiersprache, als Beschreibungssprache für eine Architektur und als Kalkül zur Performance-Vorhersage verwendbar. Ob sich diese Vielseitigkeit als Vorteil erweist, ist fraglich, zumal in [Gem93] keine Ergebnisse für die Performance-Vorhersage angegeben sind. Nachteilig ist auf jeden Fall, wie bei allen Simulationsverfahren, der Rechenzeit- und Speicheraufwand.

2.3 Weitere Methoden

In diesem Abschnitt werden weitere Methoden kurz vorgestellt. Zunächst zu den klassischen *statistischen Modellen.* Warteschlangentheorie, Verteilungsfunktionen, Markoff-Ketten und -Prozesse sind die mathematischen Hilfsmittel, mit denen die parallele Rechnerarchitektur und das Programm modelliert wird. Durch die Gedächtnislosigkeit der verwendeten Verteilungsfunktionen und Markoff-Ketten ist der aktuelle Zustand innerhalb des Modells unabhängig von seiner Vorgeschichte. Dies kann sich je nach Situation als Vorteil oder Nachteil auswirken. Rein statistische Modelle zur Performance-Vorhersage liefern bei realen Programmen meist nur sehr ungenaue Werte, da viele vereinfachende Annahmen gemacht werden müssen [Fah93]. Statistische Modelle sind in [Söt89, YaVi91, HaMe92] beschrieben.

Simulationsverfahren können mit Erfolg zur Analyse komplexer Systeme eingesetzt werden, bei denen Messungen schwierig sind. Zu diesen Verfahren gehören unter anderem Petri-Netze und Monte Carlo-Methoden. Durch Simulationen können wichtige Parameter bei der Entwicklung neuer Systeme getestet werden. Leider sind Simulationsverfahren nicht ohne Nachteile [Fah93]. Für eine hinreichend genaue Simulation ist eine große Zahl von Parametern notwendig. Diese beschreiben nicht nur die Architektur des Zielrechners (z.B. das Verhalten des Cache-Speichers und von CPU-Pipelines), sondern auch Eigenschaften des Programms (z.B. Ausführungszeiten von Anweisungen). Ein Teil der benötigten Parameter ist nicht immer verfügbar oder läßt sich nicht genau genug bestimmen. Für große Systeme sind vereinfachende Annahmen zu treffen, wodurch die Genauigkeit des verwendeten Simulationsmodells abnimmt. Selbst mit Vereinfachungen sind Simulationen oftmals noch sehr rechenzeit- und speicherintensiv. Wie sich Vereinfachungen auf das Modell auswirken, ist nicht immer genau bestimmbar. Die Ergebnisse der Simulation müssen mit realen Werten verglichen werden, um eine Aussage über die Genauigkeit der Simulation zu erhalten. Arbeiten zu Simulationsverfahren finden sich in [PrKi92, BCVY91].

Bei *analytischen Modellen* erfolgt die Modellrepräsentation durch einen Satz von mathematischen Formeln und Gleichungen. Wie bei den Simulationsverfahren müssen auch bei analytischen Modellen sinnvolle Vereinfachungen vorgenommen werden, damit die auf diesen Verfahren basierende Performance-Vorhersage überhaupt möglich ist. Die Differenzierung zwischen „wichtigen“ und „unwichtigen“ Modellparametern ist der entscheidende Teil bei der Modellbildung. Zu den Vorteilen der analytischen Modelle zählt unter anderem, daß die Entwicklung im Vergleich zu den anderen Methoden schnell und relativ einfach durchgeführt werden kann. Als Nachteile müssen die Schwierigkeiten, die bei der Beschreibung von dynamischen Vorgängen entstehen, sowie die Vereinfachungen, die für große Systeme gemacht werden müssen, genannt werden. Zu den dynamischen Vorgängen gehören zum Beispiel Routing-Mechanismen, adaptives Load-Balancing und Arbeitsverteilungs-Strategien [Fah93]. Ausführlichere Beschreibungen von analytischen Modellen sind in [Wan90, MiSc90, SVS88, VSSG84] zu finden.

Die Funktionsweise von *Benchmark-Modellen* wurde bereits in den Abschnitten 2.1.1 und 2.1.2 beschrieben. In [Sar89a, Sar89b] wird ein im Rahmen des PTRAN-Projektes entwickeltes Verfahren zur Bestimmung durchschnittlicher Programmausführungszeiten erläutert.

Werkzeuge zur Performance-Vorhersage sind wichtige Hilfsmittel für Programmierer und Compiler. Soll bei dem Programmiermodell *Compiler-Bridged Shared Memory* automatisch eine optimale Datenverteilung ausgewählt werden, so sind Werkzeuge mit einer genauen Performance-Vorhersage zwingend notwendig.

Die in diesem Kapitel vorgestellten Methoden und Werkzeuge zeigen allerdings, daß es noch viele Probleme bei der Performance-Vorhersage zu lösen gibt. Die geschilderten Beschränkungen auf reguläre Strukturen und einfache Kontrollflüsse in parallelen Programmen lassen die erfolgreiche Anwendung der Werkzeuge auf Produktions-Codes derzeit fraglich erscheinen.

Eine vollständige und genaue Performance-Analyse des Ablaufes eines parallelen Programms erhält man durch Laufzeit-Informationen (Trace-Daten). Die Erzeugung und Auswertung von Trace-Daten mit entsprechenden Werkzeugen wird in den nächsten Kapiteln dieser Arbeit beschrieben.

Kapitel 3

Verwendung von Trace-Daten zur Performance-Analyse

Die Performance-Analyse von parallelen Programmen auf massiv parallelen Rechnerarchitekturen geschieht heute in vielen Fällen auf der Basis von Trace-Daten, die während der Ausführung einer instrumentierten Programmversion erzeugt werden. Neben der Auswertung von Trace-Daten kann die Performance-Analyse auch auf der Basis eines Ausführungsprofils (Execution Profiling) oder der Probennahme (Sampling) von System-Zuständen erfolgen. Durch Ausführungsprofile ist das Auffinden von Programmteilen, die Performance-Engpässe bilden, meist recht schnell möglich. Da aber die Ausführungszeiten der einzelnen Funktionen über die gesamte Laufzeit aufsummiert sind, spiegeln diese Daten Veränderungen der Performance über der Zeit (z.B. durch zeitabhängigen Parallelismus) nicht wider.

Die Gründe für eine schlechte Performance werden bei Betrachtung der gemessenen Ausführungszeiten und der gezählten Funktionsaufrufe ebenfalls nicht deutlich. Durch das Sampling von System-Zuständen läßt sich das Verhalten des Systems über der Zeit darstellen, ein Rückschluß auf die Stellen im Programm-Code, die die mangelhafte Performance verursachen, ist jedoch nicht möglich.

Ereignis-Tracing ermöglicht die Darstellung der Performance-Veränderungen während der Programmausführung über der Zeit, die Auswertung von System-Zuständen (z.B. Message-Aufkommen zu einem bestimmten Zeitpunkt) und den Rückschluß auf für die Performance kritische Stellen im Programm-Code. Die mit dem Ereignis-Tracing erzielbaren Ergebnisse hängen dabei unmittelbar von der Instrumentierung des Programms ab [RAMN92].

3.1 Die Instrumentierung von Programmen

Unter der Instrumentierung eines Programms versteht man das Einfügen spezieller Funktionsaufrufe an bestimmten Stellen des Programm-Codes. Das Programm wird

so auf das Performance-Monitoring vorbereitet. Typische Stellen für die Instrumentierung sind:

- Anfang und Ende von parallelen Schleifen sowie
- globale Reduktions- und Synchronisationsbefehle.

Bei der Instrumentierung sind zwei Ebenen unterscheidbar:

- Quelltext-Instrumentierung
- Instrumentierung von ausführbaren Programmen

Zunächst zur Quelltext-Instrumentierung. Das Einfügen der Funktionsaufrufe für die Instrumentierung kann entweder per Hand oder durch ein Software-Werkzeug erfolgen. Ein solches Werkzeug ist zum Beispiel *iPablo* [Shi94] aus dem *Pablo Performance Analysis Environment* [RAMN92]. Abbildung 3.1 gibt einen Überblick über die Quelltext-Instrumentierung.

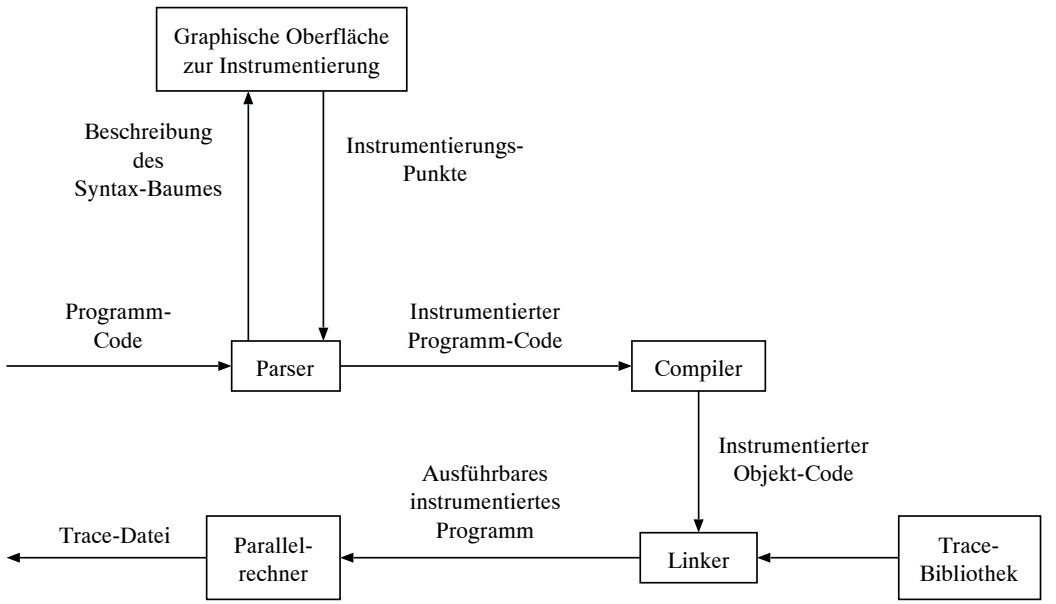


Abbildung 3.1: Instrumentierung auf Quelltext-Ebene

Wird die Instrumentierung manuell durchgeführt, entfällt die graphische Oberfläche zur Instrumentierung. Von der Mächtigkeit der in der Trace-Bibliothek vorhandenen Funktionen hängt unmittelbar der Informationsgehalt der Trace-Daten ab. Dieser wird weiterhin bestimmt durch das Instrumentierungswerkzeug. Kann es nicht alle Möglichkeiten der Trace-Bibliothek umsetzen, gehen dem Benutzer unter Umständen wichtige Informationen verloren, denn durch die Instrumentierung wird vor der Übersetzung festgelegt, was beobachtet werden soll. Abbildung 3.2 zeigt *iPablo* während einer Instrumentierungs-Session. Voraussetzung für die Instrumentierung eines

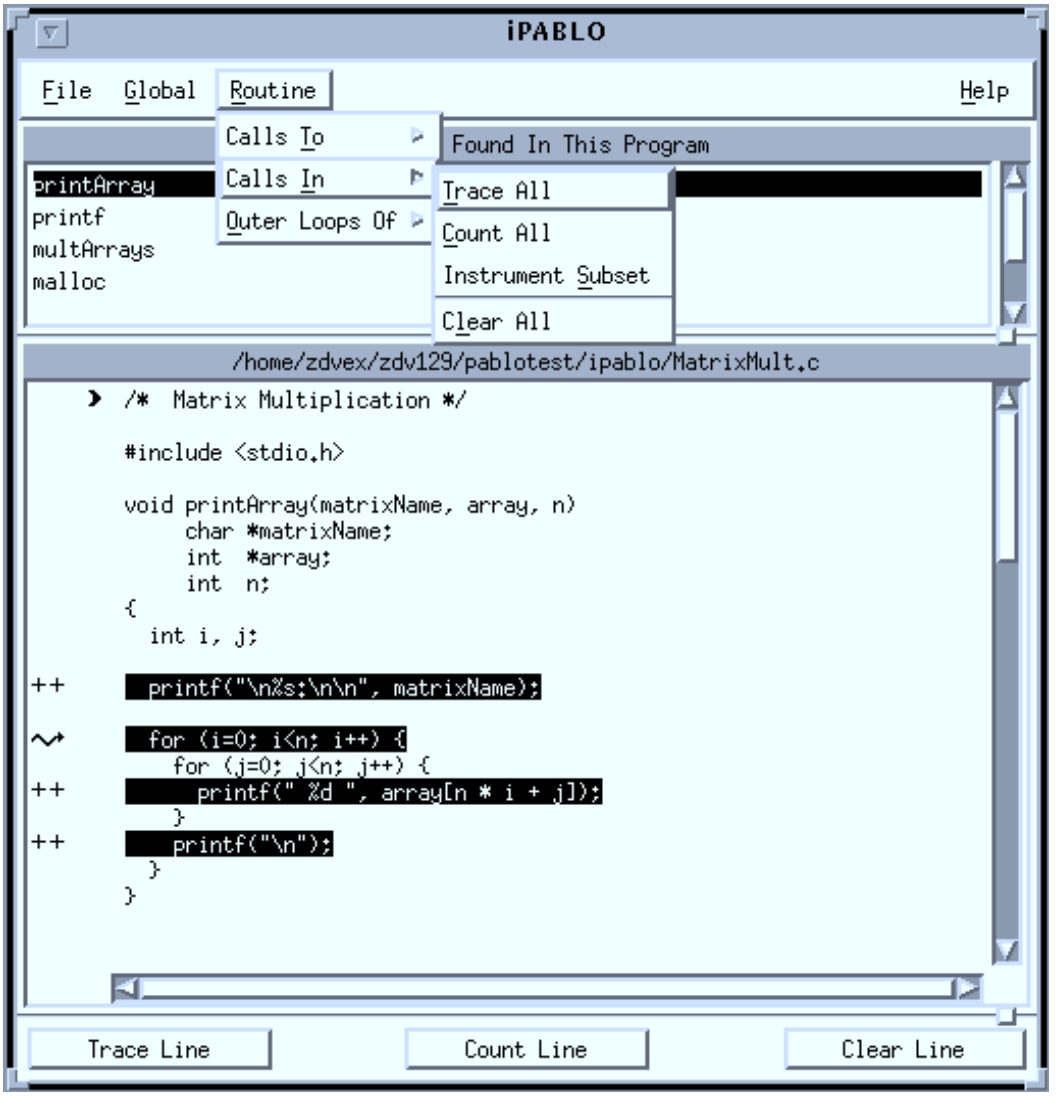


Abbildung 3.2: *iPablo* Instrumentierungs-Oberfläche

Programms durch *iPablo* ist, daß dieses in der Programmiersprache C geschrieben ist.

In den Abbildungen 3.3 und 3.4 sind ein Quelltext und seine instrumentierte Version gegenübergestellt. Die eingefügten Funktionsaufrufe aus der Instrumentierungsbibliothek sind entsprechend gekennzeichnet. Die Instrumentierung wurde mit *iPablo* durchgeführt (es handelt sich um das in der Abbildung 3.2 gezeigte Programm).

Um eine Instrumentierung von ausführbaren Programmen vornehmen zu können, muß der Binär-Code unmittelbar modifiziert werden. Dazu muß das Format des vorliegenden Binär-Codes bekannt sein. Durch die übrigen in dem Binär-Code vorhandenen Informationen (z.B. Symboltabelle) werden die Stellen ermittelt, an denen Sprungbefehle in die Trace-Routinen eingefügt werden. Es wird also tatsäch-

```
/* Matrix Multiplication */
...
void printArray(matrixName, array, n)
     char *matrixName;
     int  *array;
     int   n;
{
  int i, j;

  printf("\n%s:\n\n", matrixName);

  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      printf(" %d ", array[n * i + j]);
    }
    printf("\n");
  }
}
...
```

Abbildung 3.3: Original C-Quelltext (Ausschnitt)

lich ein fertig übersetztes und gebundenes Programm modifiziert. Die Generierung von Trace-Daten auf der Intel Paragon wird auf diese Weise durch den *ipd* (*Intel Parallel Debugger* [Int94]) durchgeführt. Man bestimmt mit Hilfe eines Parameters (`instrument -./parvis` oder `instrument -paragraph`) die Ziel-Umgebung der Performance-Analyse, für die die Trace-Daten generiert werden. Der Parameter des `instrument`-Befehls wählt die Steuerungsdatei für die Instrumentierung aus.

```
#line 1 "/home/pablo/ipablo/MatrixMult.c"
/* Matrix Multiplication */
...
#line 4 "/home/pablo/ipablo/MatrixMult.c" 2

void printArray(matrixName, array, n)
     char *matrixName;
     int  *array;
     int   n;
{
...
#line 14 "PabloParseCode.c"
{ if (PabloTraceFlag) PabloTraceLoop(52, 948, 14);
#line 14 "/home/pablo/ipablo/MatrixMult.c"
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {

#line 16 "PabloParseCode.c"
(PabloTraceFlag && PabloCountProc(0, 1002, 16), _printf_ReturnValue_ =
```

Abbildung 3.4: Instrumentierter C-Quelltext (Ausschnitt)

Da der *ipd* ausführbare Programme im *COFF*-Format (*Common Object File Format* [Gir88]) erwartet, war die Erstellung einer entsprechenden Steuerungsdatei zur Erzeugung von Trace-Daten für *PARvis* problemlos möglich [Arn93]. Abbildung 3.5 gibt einen Überblick über die Instrumentierung von Binär-Code.

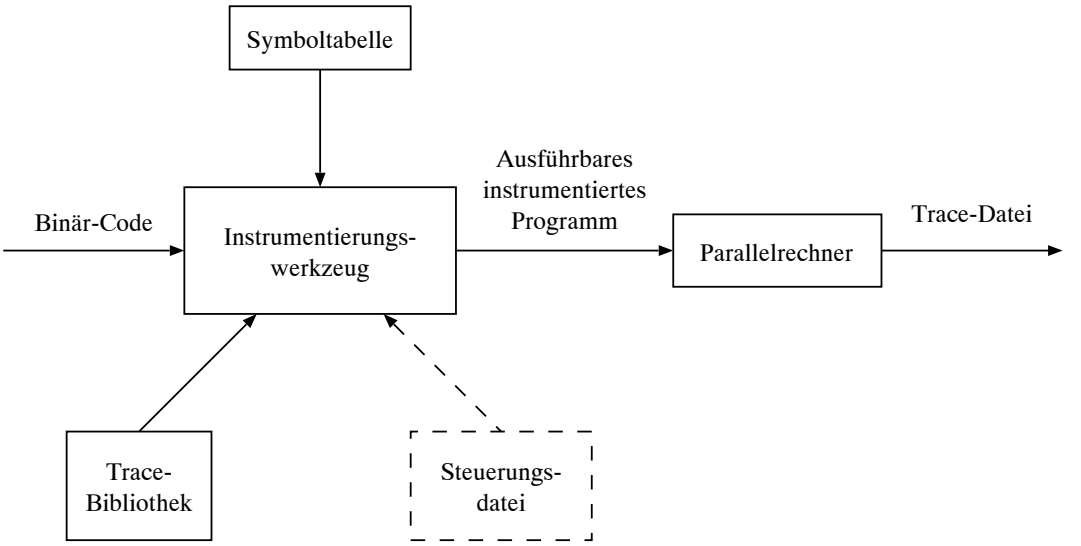


Abbildung 3.5: Instrumentierung auf Binär-Code Ebene

Durch die Instrumentierung wird ein zusätzlicher Overhead bei der Programmausführung erzeugt, denn die Ausführung der Instrumentierungsroutinen kostet Zeit. Um das eigentliche Programm so wenig wie möglich zu beeinflussen, müssen die Instrumentierungsroutinen in ihrer Ausführung sehr effizient sein. Durch Hardware-Monitore kann die Protokollierung der Programmaktionen (z.B. das Versenden von Nachrichten) unterstützt werden. Auch die Verwaltung und Speicherung der erzeugten Trace-Daten darf das I/O-Verhalten des Programms nicht stören. Läuft das Programm auf p Prozessoren parallel ab, so werden im allgemeinen auch p einzelne Dateien mit Trace-Daten erzeugt. Unter dem Betriebssystem `UNIX` kann jeder Prozessor einen eigenen Prozeß starten, der für das Schreiben der Trace-Daten auf das Speichermedium zuständig ist. Diese Prozesse laufen quasi-parallel ab (entsprechend der Scheduling-Strategie des Betriebssystems). Da die Schreiboperationen nicht direkt auf der Festplatte durchgeführt werden, sondern auf einem schnellen Pufferspeicher, entsteht beim gemeinsamen Zugriff der Prozesse auf die Festplatte kein Engpaß. Steht für die Speicherung dieser Daten ein paralleles Dateisystem zur Verfügung, beispielsweise das *PFS* der Intel Paragon, ist beim Herausschreiben der Trace-Daten noch eine Beschleunigung möglich. Dies wird durch ein Umgehen der `UNIX`-Puffer bei Schreiboperationen erreicht. Die Voraussetzung für eine Beschleunigung ist, daß die zu schreibende Datenmenge eine bestimmte Mindestgröße (64 KByte) aufweist. Über eine asynchron ablaufende Kommunikationsoperation sendet der Prozessor seine Daten an den I/O-Knoten, wodurch eine sehr hohe Übertragungsbandbreite erreichbar ist. Der Programmierer muß darauf achten, daß die Schreiboperationen

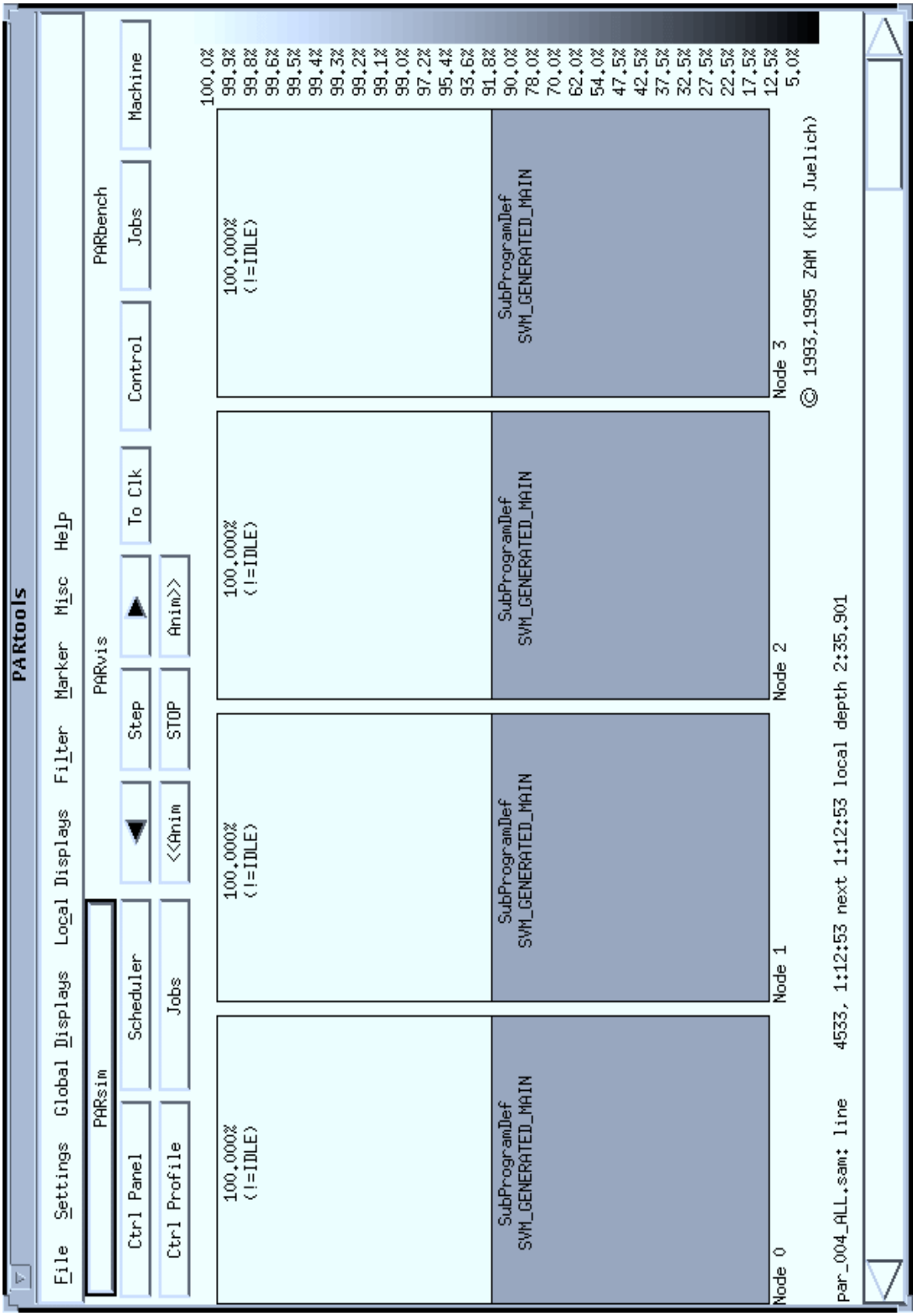


Abbildung 3.6: Startbildschirm von *PARvis*

der verschiedenen Prozessoren nicht vom gleichen I/O-Knoten durchgeführt werden [DFK94].

Trace-Daten sollten immer binär gespeichert werden, da selbst dann die einzelnen Dateien typischerweise Größen im MByte-Bereich aufweisen. Daten in diesem Umfang sind nur noch durch geeignete Visualisierungswerkzeuge auswertbar. Zu diesen Werkzeugen gehören unter anderem *PARvis* [ADN95] (siehe Abbildung 3.6), *Pablo* [RAMN92] und *ParaGraph* [HeFi94]. Die Zielsetzungen dieser Visualisierungswerkzeuge unterscheiden sich zum Teil erheblich. Eine vergleichende Beschreibung verschiedener Visualisierungswerkzeuge und deren Darstellungskonzepte findet sich in [GaHu94]. Ein Bericht über die Performance-Analyse mit den Werkzeugen des *Pablo Performance Analysis Environment* ist in [Vos95] verfaßt worden.

3.2 Die Instrumentierung für SVM-Fortran

Die Programmiersprache *SVM-Fortran* [BeGe95] ist eine Erweiterung von Fortran 77, die am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich entwickelt wurde. Sie unterstützt die parallele Programmierung auf Systemen mit gemeinsamem Speicher und stellt spezielle Sprachmittel zur Optimierung von Datenlokalität und Lastbalancierung zur Verfügung. Dabei spielt es keine Rolle, ob der gemeinsame Speicher virtuell (*Shared Virtual Memory – SVM*, z.B. ASVM-Kernel für Intel Paragon) oder durch einen Hardware-basierten gemeinsamen Adreßraum (z.B. Cray T3D) bereitgestellt wird [GeBe95].

Die Performance-Analyse von *SVM-Fortran*-Programmen wird durch das Werkzeug *OPAL (Optimizer and Locality Analyzer)* zur Instrumentierung unterstützt [GKÖ95]. Ähnlich dem in Abschnitt 3.1 erwähnten Werkzeug *iPablo* unterstützt *OPAL* die Quelltext-basierte Instrumentierung. In Abbildung 3.7 sind einige besondere Möglichkeiten von *OPAL* zu erkennen:

- Die Instrumentierung sowohl von *SVM-Fortran*-Direktiven als auch von Variablen.
- Die Auswertung bestimmter Ereignisse aus der Trace-Datei.
- Die Präsentation der Trace-Informationen im Quelltext.

Eine ausführlichere Beschreibung dieser Möglichkeiten erfolgt in Abschnitt 5.1.

Durch Auswertung der Trace-Informationen des ASVM-Kernels (Advanced Shared Virtual Memory, [MaZe94]) lassen sich Statistiken beispielsweise über die aufgetretenen Seitenfehler (*Read Page Faults* und/oder *Write Page Faults*) gewinnen. Durch den *SVM-Fortran Application Monitor* (*SAM* [Özm95]) werden die verschiedenen Ereignisse (*SVM-Fortran*-Sprachereignisse und ASVM Kernel-Ereignisse) erfaßt und im SDDF-Datenformat gespeichert (siehe Abbildung 3.8).

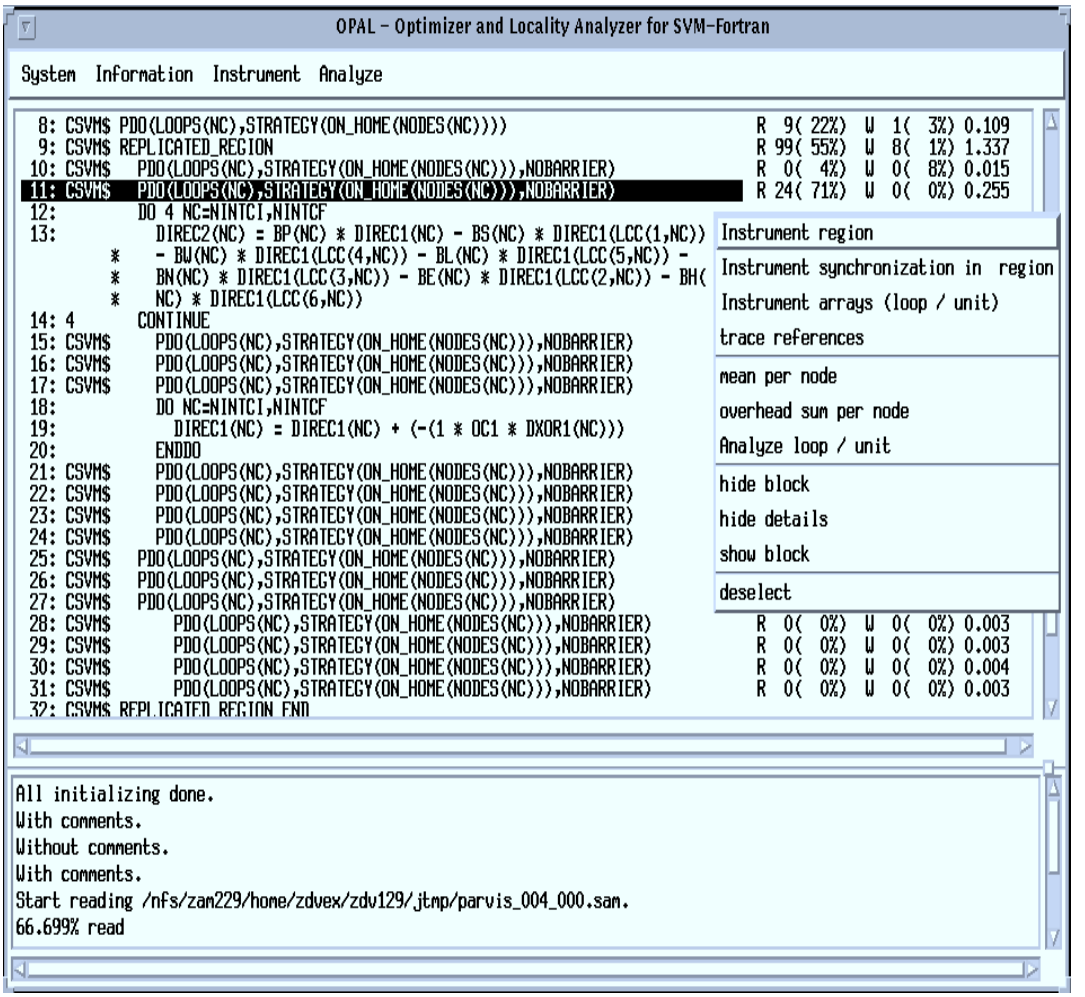


Abbildung 3.7: *OPAL* – Optimizer and Locality Analyzer für *SVM-Fortran*

Das SDDF-Datenformat *(Self-Defining Data Format)* ist ein selbstbeschreibendes Datenformat für Trace-Daten, das an der University of Illinois für das *Pablo Performance Analyses Environment* entwickelt wurde [Ayd94]. Es gibt sowohl eine binäre Repräsentation innerhalb des SDDF-Datenformates als auch eine ASCII-Repräsentation. Für das binäre Format spricht die komprimierte Speicherung

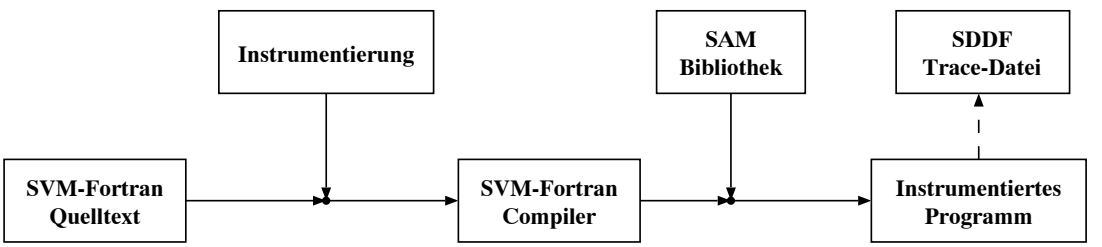


Abbildung 3.8: Erzeugung von Ereignissen mit dem *SAM*

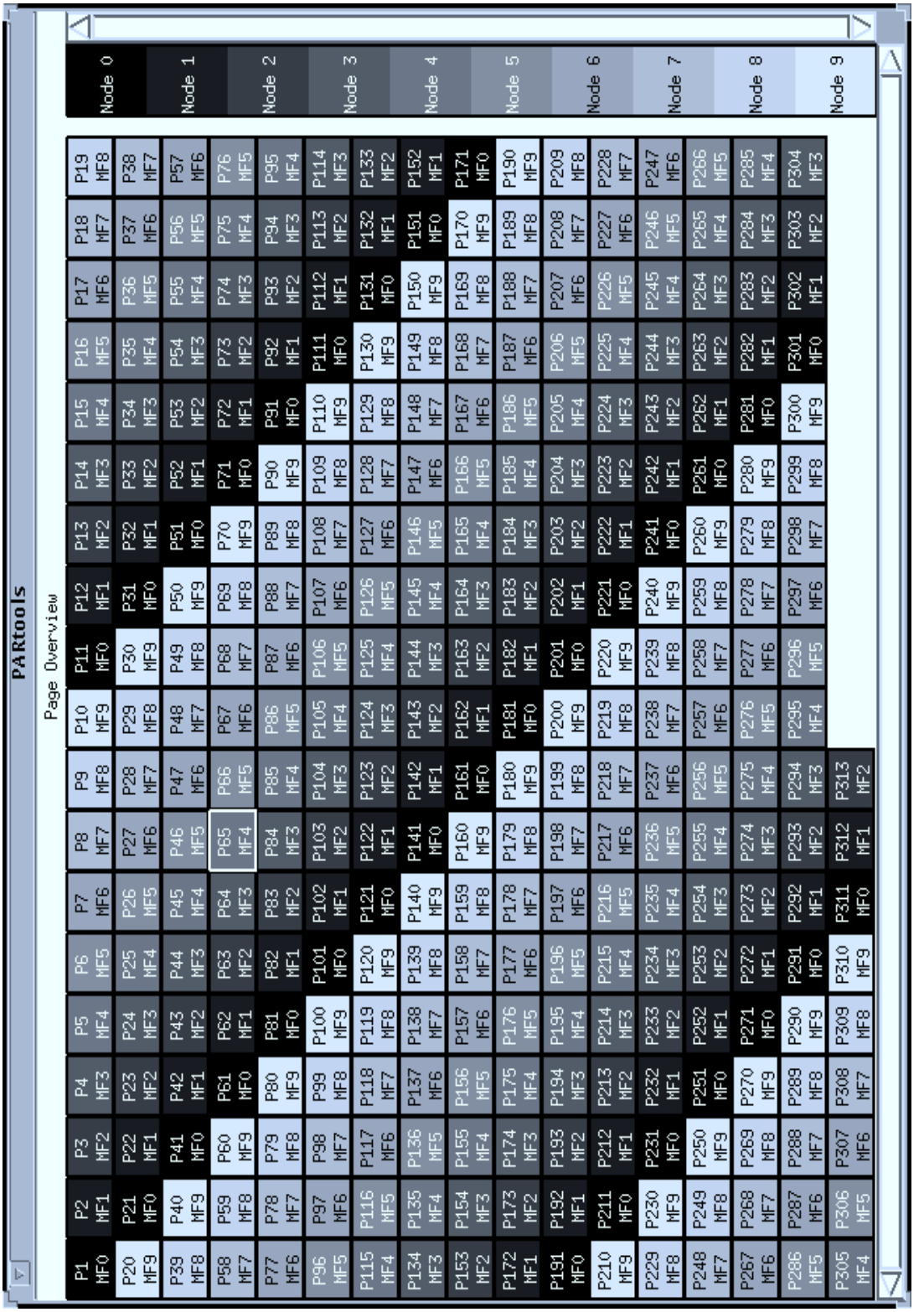


Abbildung 3.9: Seitenübersicht in *PARvis*

der Trace-Daten. Probleme bereitet jedoch die unterschiedliche Zahlendarstellung („Little Endian“ und „Big Endian“) auf verschiedenen Systemen. Bei einer ASCII-Darstellung wird dieses Problem vermieden. Weiterhin hat diese Darstellung den Vorteil, daß die Trace-Daten im Klartext lesbar sind. Nachteilig ist, daß die ohnehin großen Trace-Dateien durch die ASCII-Darstellung weiter vergrößert werden. Eine genaue Beschreibung des SDDF-Datenformates erfolgt in Abschnitt 3.4.

Zur Visualisierung der Trace-Daten von *SVM-Fortran*-Programmen soll das ebenfalls am ZAM entwickelte Werkzeug *PARvis* Verwendung finden. In [Mül94] ist die Erweiterung von *PARvis* zur Darstellung von Informationen über Speicherseiten im SVM-Kontext beschrieben. Eine dieser Darstellungsmöglichkeiten ist beispielsweise eine sogenannte *Page Overview* (Seitenübersicht), wie sie in Abbildung 3.9 gezeigt wird. Jedes Rechteck symbolisiert dabei eine Speicherseite. Die Identifikation einer Speicherseite erfolgt über ihre Nummer (z.B. P65 = Page 65). Als zusätzliche Information ist hier noch der für die jeweilige Speicherseite zuständige Manager-Knoten eingetragen. So ist der Knoten 4 (MF4 = Manager Field 4) der Manager der Speicherseite 65. Eine globale Übersicht über die Verteilung der einzelnen Speicherseiten auf die Manager-Knoten erhält man über die Farbzuordnungen rechts in der Seitenübersicht (in der Abbildung dargestellt durch Graustufen). Da *PARvis* eine Komponente der PARtools-Umgebung ist, liegt es auf der Hand, daß *PARvis* auch das PARtools-eigene Format für Trace-Daten verwendet [Arn93]. Durch die Abspeicherung der Trace-Daten als ASCII-Text ist eine plattformunabhängige Verarbeitung der Daten möglich. Eine genaue Beschreibung des von *PARvis* verwendeten Datenformates erfolgt in Abschnitt 3.3.

Für die Konvertierung von mit dem *ipd* erzeugten Trace-Daten im SDDF-Datenformat in das *PARvis*-Datenformat existiert bereits das Werkzeug *trf2pv*. Die Konvertierung von *SVM-Fortran* Trace-Daten im SDDF-Datenformat wird in Abschnitt 3.7 beschrieben. Hierfür ist ein neuer Konverter notwendig, da zwar die physikalische Definition des SDDF-Datenformates gleich bleibt, die logische Definition (d.h. die Definition der Ereignisse) sich jedoch ändert.

3.3 Das PARvis-Datenformat für Trace-Daten

Bei der Entwicklung des *PARvis*-Datenformates für Trace-Daten stand die Portabilität im Vordergrund. Aus diesem Grund wurde eine ASCII-Repräsentation für die Trace-Daten gewählt. Einfluß auf die Grammatik dieses Datenformates hatte auch der Parallelrechner-Simulator der PARtools-Umgebung *PARsim*. In diesem Simulator werden Programme abgearbeitet, die durch ein anwendungsnahes Lastprofil beschrieben sind. Für eine realitätsnahe Gestaltung der Lastprofile kann man auf die mit dem dritten Werkzeug der PARtools-Umgebung *(PARbench)* gemachten Erfahrungen zurückgreifen. *PARbench* erzeugt synthetische Programmkerne aus verschiedenen Parametern, die ein bestimmtes Lastprofil charakterisieren. Die in [NaLi93] veröffentlichten Ergebnisse zeigen, daß die durch die synthetischen Programmkerne

erzeugte Last auf dem untersuchten Rechner sehr gut den durch die Parameter gemachten Vorgaben entsprach.

Bei einer Simulation in *PARsim* treten in der Hauptsache Zustandswechsel der einzelnen Prozessoren als Ereignisse auf. Diese können durch Zuordnung eines bestimmten Ereignistyps und einer Zeitinformation für den Zeitpunkt des Auftretens eindeutig beschrieben werden. Die Kennzeichnung dieses Zeitpunktes kann beispielsweise durch die Angabe der Zahl der Maschinenzyklen, die seit Simulationsbeginn verstrichen sind, erfolgen. Da die Trace-Daten als ASCII-Text abgespeichert sind, werden die einzelnen Ereignisse durch Zeilenvorschübe voneinander getrennt. So enthält jede Zeile der Trace-Datei jeweils genau einen Eintrag (und damit auch genau ein Ereignis) der Form:

```
[Zeitinformation] ⟨Ereignistyp⟩ [Parameter [Parameter...]]
```

Fehlt die Angabe der Zeitinformation, wird die zuletzt angegebene Zeitinformation verwendet. Die Folge der Zeitinformationen muß nur monoton steigend sein, es können also mehrere direkt aufeinanderfolgende Zeilen mit derselben Zeitinformation existieren.

Neben den in der Simulation aufgetretenen Ereignistypen gibt es spezielle Ereignistypen, die reservierte Namen haben. Mit ihnen werden *PARvis* für die Visualisierung wichtige Simulationsparameter übermittelt. Dazu gehört beispielsweise die Zahl der von dem Programm verwendeten Prozessoren `NCPUS` oder die Dauer eines Maschinenzyklus in Sekunden `CLKPERIOD`. Für eine weitergehende Beschreibung des *PARvis* Trace-Datenformates siehe [Arn95].

Mit der Erweiterung der *PARvis*-Funktionalität zur Visualisierung von Speicheraktivitäten von SVM-Systemen wurde ein neuer Ereignistyp eingeführt. Er dient der Kennzeichnung der Zustandsänderungen einer einzelnen Speicherseite und hat die folgende Syntax:

```
[Zeitinformation] PAGE#    [Parameter [Parameter...]]
```

Mit Hilfe eines angenommenen *Write Page Faults* soll die *PARvis*-Notation eines solchen Ereignisses veranschaulicht werden (für eine vollständige Beschreibung sämtlicher Parameter dieses Ereignistyps siehe [Mül94]). Zum Zeitpunkt t_1 will der Prozessor p auf die Speicherseite n schreiben, stellt aber fest, daß diese Speicherseite in seinem lokalen Speicher nicht vorhanden ist, bzw. er kein Schreibrecht für diese Speicherseite hat. Mit dem Versenden der angeforderten Speicherseite wird zum Zeitpunkt t_2 begonnen. Dieser Vorgang ist nach $t_3 - t_2$ Zeiteinheiten abgeschlossen, die angeforderte Speicherseite ist von Prozessor q vollständig bei ihm eingetroffen. Er erhält die alleinige Schreibberechtigung für diese Speicherseite und wird der neue Besitzer dieser Speicherseite. In der Trace-Datei im *PARvis*-Format befinden sich die folgenden Einträge:

```
t1   PAGE# n   WW p
⋮
t2   PAGE# n   OF p   WF p   CT p   RC 1   RT
⋮
t3   PAGE# n   RB q   CT
```

In der Tabelle `WW` (Waiting Table Write) stehen die Prozessoren, die auf das Schreibrecht für die Speicherseite warten. Das Besitzerfeld `OF` (Owner Field) kennzeichnet durch die nachfolgende Prozessornummer den aktuellen Besitzer der Speicherseite. In dem Schreiberfeld `WF` (Writer Field) steht die Nummer des Prozessors, der die Schreibberechtigung für diese Speicherseite hat. Die Tabelle `CT` (Changing Table) enthält Prozessoren, die zwar schon die Rechte für eine Speicherseite besitzen, mit ihr aber noch nicht arbeiten können, da das Versenden noch nicht abgeschlossen ist. Da mit der Schreibberechtigung auch eine Leseberechtigung einhergeht, steht in dem Feld `RC` (Read Count) eine 1. Erhöht sich die Anzahl der Prozessoren mit einer Leseberechtigung für diese Speicherseite, wird die Zahl im Feld `RC` entsprechend inkrementiert (und im umgekehrten Fall dekrementiert). Durch die Angabe von `RT` (Read Table) ohne Argument ist sichergestellt, daß die Tabelle mit den Nummern der Prozessoren, die ein Leserecht für diese Speicherseite haben, nach dem Besitzerwechsel neu initialisiert wird. Das Feld `RB` (Received By) enthält die Prozessornummer des „Absenders“, also von wem die Speicherseite gekommen ist.

Zum Zeitpunkt t_2 entfällt der Eintrag in der „Wartetabelle“, da mit dem Versenden der Speicherseite begonnen wird. Der Prozessor p ist dann der Besitzer der Speicherseite und hat das alleinige Schreibrecht. Nach Abschluß des Zusendevorganges (t_3) wird der Prozessor p aus der Tabelle `CT` ausgetragen.

Aus dem vorangegangenen Beispiel wird noch einmal deutlich, daß durch diese Art der Notation nur eine Zustandsänderung pro Zeitpunkt dargestellt werden kann, was ein grundlegender Unterschied zum SDDF-Format ist, welches im folgenden Abschnitt vorgestellt wird.

3.4 Das SDDF-Datenformat für Trace-Daten

Wie bereits in Abschnitt 3.2 erwähnt, wurde das SDDF-Datenformat im Rahmen des *Pablo Performance Analysis Environment* entwickelt. Ziel bei der Entwicklung von *Pablo* war die Bereitstellung einer Umgebung zur Performance-Analyse für SIMD- (Single Instruction Multiple Data) wie auch für MIMD- (Multiple Instruction Multiple Data) Architekturen mit verteiltem Speicher. Das SDDF-Datenformat findet Anwendung nicht nur innerhalb des *Pablo Performance Analysis Environment*, sondern unter anderem auch in der *ParAide*-Umgebung der Intel Paragon.

Folgende Kriterien bestimmten die Entwicklung des SDDF-Datenformates [Ayd94]:

- Trace-Dateien können sehr groß werden (typischerweise im MByte-Bereich). Die Kompaktheit der Darstellungsform ist also sehr wichtig. Eine mögliche Komprimierung sollte nicht ausgeschlossen werden.
- Die Auswertung der Trace-Daten wird meist auf einer anderen Rechnerarchitektur als dem Parallelrechner erfolgen. Das Format muß also in der Lage sein, mögliche Unterschiede in der Reihenfolge der Bytes, der Wortlänge und der Fließkomma-Darstellung aufzuzeigen und zu handhaben.
- Je nach untersuchtem Programm (Anwendungs- oder Systemprogramm) sind jeweils bestimmte Ereignisse von ganz besonderem Interesse. Dieses gilt entsprechend für die zugrunde liegende Systemarchitektur (gemeinsamer oder verteilter Speicher). Daher müssen sich sehr verschiedene Ereignisse darstellen lassen.
- Wenn sich während der Performance-Analyse herausstellt, daß zur genaueren Untersuchung weitere Ereignistypen benötigt werden, müssen diese zu den vorhandenen hinzugefügt werden können.

Aus diesen Kriterien ergibt sich, daß ein zu definierendes Format für Trace-Daten kompakt, portierbar, umfassend und erweiterbar sein muß. Diesen Forderungen trägt das SDDF-Format Rechnung. Es ist genau genommen eine Beschreibungssprache für Trace-Daten, welche die Struktur von Datensätzen und Instanzen dieser Datensätze spezifiziert. In einem Header wird zunächst festgelegt, ob die nachfolgenden Trace-Daten in ASCII oder binär vorliegen. Falls die Trace-Daten in Binärdarstellung vorliegen, folgen Angaben über die maschinenspezifische Darstellung von Daten (z.B. Fließkomma-Darstellung). So ist sichergestellt, daß die Trace-Daten unabhängig von der Art der Darstellung auf dem „Auswertungsrechner“ richtig interpretiert werden. Das Kriterium der Portierbarkeit wird so erfüllt. Dies ist auch ein wichtiger Gesichtspunkt für das in der Zukunft sicherlich immer wichtiger werdende Metacomputing. Trace-Dateien im SDDF-Format, die auf verschiedenen im Rahmen eines Metacomputing-Verbundes zusammengeschalteten Rechnern erzeugt wurden, können gleichzeitig durch ein Werkzeug auf einem Rechner (typischerweise einer Workstation) ausgewertet werden.

Der vollständige Header für die Binärdarstellung ist folgendermaßen definiert:

⟨sddf binary indicator⟩ ⟶ SDDFB
⟨bytes in header⟩ ⟶ ⟨numeric-valued byte⟩
⟨byte order⟩ ⟶ `BIG_ENDIAN` | `LITTLE_ENDIAN`
⟨integer format⟩ ⟶ `TWOS_COMPLEMENT` | `ONES_COMPLEMENT` | `SIGN_MAGNITUDE`
⟨float format⟩ ⟶ `IEEE` | `VAX` | `IBM` | `CRAY`
⟨double format⟩ ⟶ `IEEE` | `VAX` | `IBM` | `CRAY`
⟨long double format⟩ ⟶ `IEEE` | `VAX` | `IBM` | `CRAY`
⟨character format⟩ ⟶ `ASCII` | `EBCDIC`
⟨sizeof char⟩ ⟶ ⟨numeric-valued byte⟩
⟨sizeof short⟩ ⟶ ⟨numeric-valued byte⟩

⟨sizeof int⟩ ⟶ ⟨numeric-valued byte⟩
⟨sizeof long⟩ ⟶ ⟨numeric-valued byte⟩
⟨sizeof float⟩ ⟶ ⟨numeric-valued byte⟩
⟨sizeof double⟩ ⟶ ⟨numeric-valued byte⟩
⟨sizeof long double⟩ ⟶ ⟨numeric-valued byte⟩

Der Header für die ASCII-Darstellung besteht nur aus einem Eintrag:

⟨ascii header⟩ ⟶ SDDFA

Danach folgen bereits die Trace-Daten in ASCII-Darstellung. Dem Vorteil der Lesbarkeit der Trace-Daten steht der Nachteil der ineffizienten Speicherung der Trace-Daten gegenüber. Für eine kompakte Speicherung sorgt die Binär-Darstellung. Nach Angaben in [Ayd94] sind binäre Trace-Dateien im SDDF-Format zwischen 42 % und 75 % kleiner als ihre ASCII-Pendants. Wenn nicht auf eine direkte Lesbarkeit der Trace-Daten Wert gelegt wird, sollte für die Speicherung das Binär-Format gewählt werden. Zumal das bisherige Argument für ein ASCII-Format, die Portierbarkeit der Daten, hier auch für das Binär-Format gilt.

Der nachfolgende eigentliche Datenbereich für Trace-Informationen ist in Paketform strukturiert. Jedes Paket gliedert sich wiederum

- in die Angabe der Paketlänge in Bytes,
- in den *Packet Type*,
- das *Packet Tag*,
- und den eigentlichen Datenbereich.

Als Pakettypen werden unterschieden

- *Stream Attribute*-Pakete,
- *Record Descriptor*-Pakete,
- *Record Data*-Pakete
- und *Command*-Pakete.

Die Anzahl der Pakete in einer Trace-Datei ist beliebig. Eine Übersicht über die Grammatik des SDDF-Formates gibt Abbildung 3.10.

Wie aus der Abbildung hervorgeht, ist die Reihenfolge der einzelnen Pakettypen keinem festen Schema unterworfen. Um jedoch die spätere Auswertung der Trace-Dateien zu erleichtern, sollten die *Record Descriptor*-Pakete in einem Block vor den *Record Data*-Paketen in der Trace-Datei liegen. In den *Record Descriptor*-Paketen ist

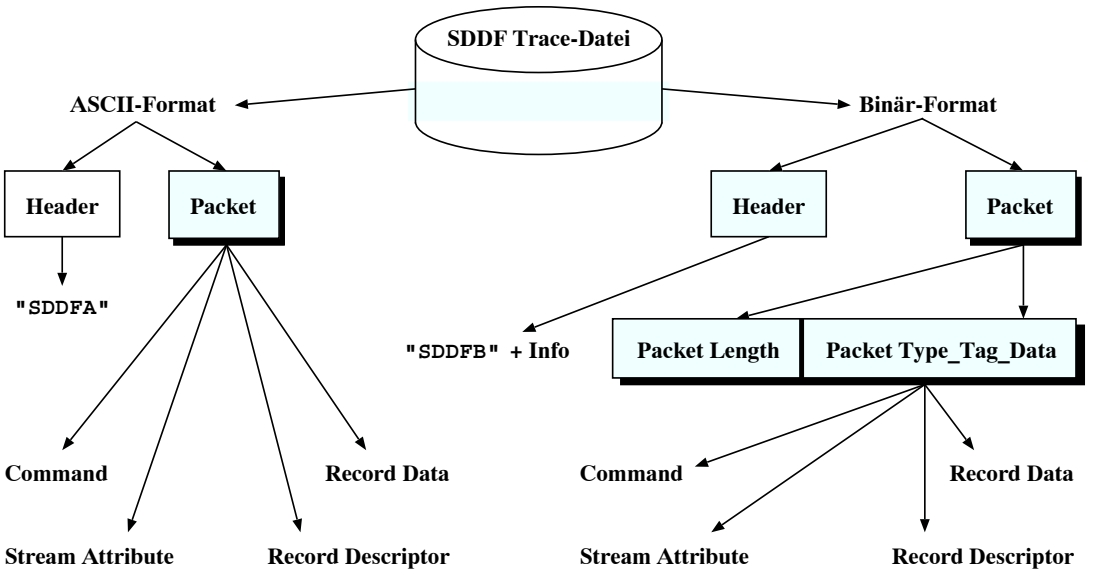


Abbildung 3.10: Übersicht über die SDDF-Grammatik

die Struktur eines Ereignisses definiert. Diese Definition ist dem `struct`-Konstrukt der Programmiersprache C sehr ähnlich. Zur Verdeutlichung sei ein *Read Page Fault*-Ereignis aus der Definition der Trace-Daten von *SVM-Fortran* angegeben (zur besseren Übersicht in der ASCII-Darstellung):

```
#1:
// "event" "read page fault has occurred and is serviced"
"read-page-fault-serviced" {
        int     "node id";
        double  "time";
        double  "time occurred";
        int     "data address";
        int     "write permission";
        int     "service node id";
};;
```

Gemäß dieser Definition werden die Dateien in einem *Record Data*-Paket abgelegt. Die Zuordnung zwischen Definition und Daten erfolgt über das *Packet Tag*, welches eindeutig vergeben werden muß. Sollte beim Auftreten eines *Record Data*-Paketes der entsprechende *Record Descriptor* noch nicht bekannt sein, können die Daten nicht interpretiert werden. Bei der Gestaltung einer Definition ist man völlig frei. So wird sichergestellt, daß alle während der Programmausführung verfügbaren Informationen in *Record Data*-Paketen festgehalten werden können.

Stream Attribute-Pakete dienen dazu, Kommentare in den Daten-„Strom“ von Paketen einzustreuen. Intel verwendet diesen Pakettyp bei der Generierung von Trace-

Dateien durch den *ipd* beispielsweise dazu, Angaben über die Erstellung in der Datei abzuspeichern:

```
/*
 * "Paragon Trace Format" "Version 1.0"
 * "Run date" "12/07/94"
 * "System" "zam158"
 * "Partition" ".compute"
*/ ;;
```

Im Gegensatz zu den bisher vorgestellten Pakettypen besteht das *Command*-Paket nur aus dem *Packet Type* und dem *Packet Tag*; der Datenbereich ist leer. Bei der Auswertung von Trace-Daten kann es notwendig sein, zu bestimmten Zeitpunkten bestimmte Aktionen auszulösen. Eine Zuordnung dieser Aktionen zu einem *Command*-Paket erfolgt über das *Packet Tag*. Die Bedeutung dieses Tags wird, im Gegensatz zu einem Ereignis, nicht in der Trace-Datei festgelegt. Diese Information muß extern übermittelt werden. *Command*-Pakete können bei der Auswertung von Trace-Daten eingesetzt werden, um beispielsweise in einen anderen Auswerte-Modus zu wechseln. Eine andere Anwendungsmöglichkeit besteht in der Steuerung der Trace-Daten Erzeugung selbst. Bei einem auftretenden Prozeßwechsel des instrumentierten Programms kann so zum Beispiel das Leeren von Puffern für Trace-Daten veranlaßt werden.

Für den Zugriff auf Trace-Daten im SDDF-Format existiert eine sehr umfangreiche C++-Klassenbibliothek, die ebenfalls im Rahmen des *Pablo*-Projektes entwickelt wurde [Pab95].

3.5 Die Pablo C++-Klassenbibliothek für das SDDF-Datenformat

Der Umgang mit SDDF-Trace-Daten wird durch die in der C++-Klassenbibliothek implementierten Methoden wesentlich erleichtert. Die Grammatik des SDDF-Formates wurde so in Objekte abgebildet, daß es für den Programmierer transparent ist, ob er Trace-Daten im ASCII-Format oder in binärer Darstellung bearbeitet. Eine bei binären Daten gegebenenfalls notwendige Umsetzung in die Byte-Reihenfolge („Little Endian“ nach „Big Endian“ oder umgekehrt) wird automatisch durchgeführt. Die Klassenbibliothek ist symmetrisch in bezug auf die Richtung der zu verarbeitenden Datenströme und sie unterstützt gleichermaßen die Erzeugung wie die Bearbeitung von Trace-Dateien im SDDF-Format.

Die in den *Record Descriptor*-Paketen transportierten Definitionen für die einzelnen Ereignisse können entweder separat in Objekten der Klasse *Record Dossier* oder zusammen in einem Objekt der Klasse *Record Dictionary* abgespeichert werden. Aus

einem gefüllten *Record Dictionary* kann später eine einzelne Definition wieder in ein Objekt der Klasse *Record Dossier* extrahiert werden. Die Unterscheidung der einzelnen Definitionen voneinander erfolgt wieder über das *Packet Tag*. Der Zugriff auf einzelne *Record Data*-Pakete kann über das *Record Dictionary* oder das *Record Dossier* erfolgen. Das *Record Dictionary* hat den Vorteil, daß sämtliche Definitionen stets vorrätig gehalten werden. Ist ein *Record Data*-Paket als solches über den *Packet Type* identifiziert worden, wird über das *Packet Tag* die „passende" Ereignis-Definition im *Record Dictionary* lokalisiert. Dann werden die Daten ihren korrespondierenden Feldern zugeordnet. Der Zugriff auf ein einzelnes Datum erfolgt dann beispielsweise über die Angabe des betreffenden Komponenten-Namens. In Abbildung 3.11 ist dieser Vorgang und der Zusammenhang zwischen dem *Record Dictionary* und dem *Record Dossier* noch einmal dargestellt. Ähnlich wie diese beiden beschriebenen Klassen gibt es weitere, die einfachen Umgang auch mit komplexeren Ereignis-Definitionen ermöglichen.

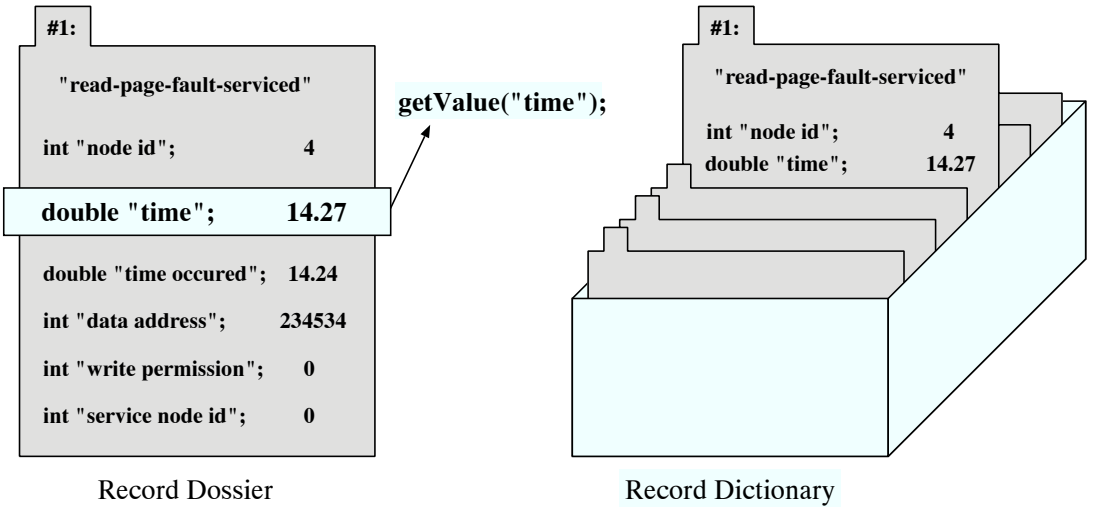


Abbildung 3.11: Die Verwendung von *Record Dossier* und *Record Dictionary*

Neben den Klassen, die die SDDF-Grammatik widerspiegeln, gibt es noch weitere Klassen, die bei der Vorverarbeitung von Trace-Dateien nützlich sind. So gibt es zum Beispiel die Klasse *SDDFmerge*, die die Ereignisse aus einzelnen Trace-Dateien zu einer einzigen Folge von chronologisch sortierten Ereignissen zusammenmischt. Diese Folge ist dann in einer Trace-Datei gespeichert. Auf einer *Sun Sparc Station 20* konnten für das Mischen von 8 Trace-Dateien folgende Zeiten gemessen werden (die Größe der durch das Mischen erzeugten Datei betrug 12 MByte):

```
SDDFmerge:
real 6:21.8     (Gesamte Ausführungszeit des Programms)
user 5:50.2     (Ausführungszeit der programmeigenen Prozeduren)
sys      2.7    (Ausführungszeit für Systemaufrufe des Programms)
```

Der zeitliche Aufwand für das Mischen läßt sich verringern (wie in Abschnitt 3.7.1 gezeigt wird), wenn das über den Aufbau einer speziellen SDDF-Trace-Datei (z.B. bei

SVM-Fortran) vorhandene Wissen direkt in die Arbeitsweise des Mischprogramms einfließen kann. Die universelle Anwendbarkeit auf alle Arten von Trace-Dateien im SDDF-Format geht dabei natürlich verloren. Interessant bei den obigen mit dem `Unix`-Kommando `time` durchgeführten Zeitmessungen war das generell gute I/O-Verhalten der entsprechenden Methoden aus der Klassenbibliothek. Dies bestätigte sich auch bei Messungen an anderen Anwendungen mit der SDDF-Klassenbibliothek.

Zusätzlich zu den hier beschriebenen Klassen in der C++-Klassenbibliothek gibt es darauf aufbauende Hilfsprogramme. Sie erlauben beispielsweise die Konvertierung von der ASCII-Darstellung in die binäre Darstellung usw. Zur Arbeit mit SDDF-Trace-Dateien existiert also ein umfangreicher „Werkzeugkasten“, mit dessen Hilfe die Realisierung eines kompakten Treibers für diese Trace-Dateien in *PARvis* möglich wurde. Im folgenden Abschnitt wird zunächst auf die Besonderheiten der Definitionen von *SVM-Fortran*-Ereignissen eingegangen, um dann die Realisierung des Treibers in *PARvis* (Abschnitt 3.7) zu beschreiben.

3.6 Die Darstellung von SVM-Fortran-Ereignissen im SDDF-Datenformat

SVM-Fortran-Ereignisse lassen sich in drei Gruppen aufteilen:

- *Kernel Event*-Ereignisse,
- *Kernel Information*-Ereignisse und
- *Language Information*-Ereignisse.

Die Gruppe der *Kernel Event*-Ereignisse unterscheidet sich von den beiden anderen Gruppen, da hier direkt Ereignisse des ASVM-Kernels der Intel Paragon protokolliert werden. Zu diesen Ereignissen gehören unter anderem sogenannte *Read Page Faults* und *Write Page Faults*. Zur Realisierung des Konzeptes eines gemeinsamen virtuellen Speichers wird der lokale Speicher der einzelnen Prozessoren in Speicherseiten aufgeteilt. Möchte ein Prozessor Daten aus dem gemeinsamen virtuellen Speicher lesen oder in ihn hineinschreiben, wird zunächst die Adresse der Daten im Speicher auf einer Speicherseite lokalisiert. Fehlt diese Speicherseite im lokalen Speicher des Prozessors, so wird sie angefordert und, je nach Vorgang, ein *Read Page Fault* oder *Write Page Fault* generiert. Die bei einem Read Page Fault protokollierten Informationen wurden bereits auf Seite 37 gezeigt.

In der Gruppe der *Kernel Information*-Ereignisse sind statistische Auswertungen von *Kernel Event*-Ereignissen zusammengefaßt. Während der Programmausführung zählt der *SAM* die verschiedenen *Kernel Event*-Ereignisse und bildet so zum Beispiel ein *Read Page Fault Sum*-Ereignis.

Die letzte Gruppe der Ereignisse wird von den *Language Information*-Ereignissen gebildet. Diese Ereignisse dienen dazu, den Ablauf eines Programms und der dabei entstehenden Datenverteilung später rekonstruieren zu können. Daher wird die Erzeugung dieser Ereignisse vom Kontrollfluß des Programms gesteuert, indem beispielsweise zu Beginn und am Ende einer `PDO`-Schleife ein Ereignis erzeugt wird.

Eine genaue Beschreibung dieses Vorganges findet sich in [Özm95]. Abbildung 3.12 gibt noch einmal eine Übersicht über die unterschiedlichen Gruppen von Ereignissen in *SVM-Fortran*-Programmen.

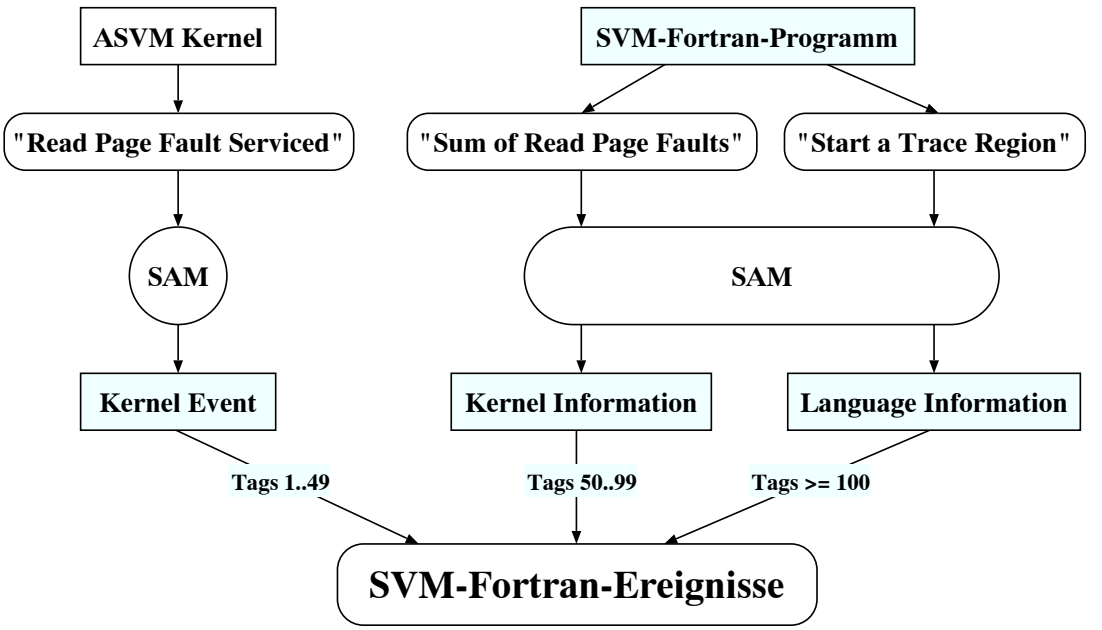


Abbildung 3.12: Übersicht über die Ereignis-Gruppen für *SVM-Fortran*

3.7 Die Konvertierung der Trace-Daten von SVM-Fortran-Programmen für PARvis

Um mit *PARvis* Trace-Daten in einem anderen Format verarbeiten zu können, gibt es prinzipiell zwei Möglichkeiten:

- externer Konverter
- interner Treiber

Ein externer Konverter kann eine tatsächliche Formatumsetzung auf Dateiebene vornehmen. Außer einer genauen Kenntnis des Aufbaues der beiden Dateiformate ist kein weiteres Wissen zur Erstellung eines Konverters notwendig. Unterscheiden sich

die beiden Dateiformate stark voneinander (z.B. *PARvis*: repräsentiert Zustandsänderungen, SDDF: Darstellung von verschiedensten Ereignissen), kann die Erstellung eines Konverters sehr aufwendig werden. Außerdem kann die Konvertierung mit einem Informationsverlust einhergehen, da unter Umständen nicht alle Informationen des Ausgangsformates im *PARvis*-Format darstellbar sind. Für den Anwender von *PARvis* ergibt sich der Nachteil, daß er vor der Auswertung der Trace-Datei diese erst konvertieren muß. Bei der physikalischen Größe von Trace-Dateien kann dies eine nicht nur als „lang“ empfundene Zeitspanne in Anspruch nehmen. Außerdem muß er den richtigen Konverter auswählen, so daß das verwendete Trace-Format für ihn nicht mehr transparent ist.

Mit Hilfe des *ipd* lassen sich Trace-Daten von Message-Passing-Programmen gewinnen. Die Speicherung der Daten erfolgt dabei im SDDF-Format. Zur Umsetzung in das *PARvis*-Format stand bereits ein in der Programmiersprache C implementierter externer Konverter zur Verfügung. Die Konvertierung vom SDDF- in das *PARvis*-Format war mit vertretbarem Aufwand zu realisieren, da in den *Record Descriptors* jeweils genau ein Ereignis definiert wird. Die Abbildung auf die Zustandsänderungen des *PARvis*-Formates gelang durch die Einführung eines `stacks`, der die Auflösung von Unterprogramm-Rücksprüngen ermöglicht [Arn95]. Nachdem die *Pablo* C++-Klassenbibliothek zur Verfügung stand, wurde im Rahmen dieser Arbeit eine Neuimplementierung des Konverters in C++ unter Verwendung von Methoden der *Pablo* C++-Klassenbibliothek durchgeführt. Der Quelltext des Konverters konnte in der C++-Version sehr viel kompakter ausfallen, da durch die Methoden der Klassenbibliothek ein direkter Zugriff auf die benötigten Daten möglich wurde. Weiterhin spielt es jetzt auch keine Rolle mehr, in welchem der möglichen SDDF-Formate (ASCII, binär oder converted – „verkehrte“ Byte-Reihenfolge) die Trace-Datei vorliegt.

Interessant sind die ermittelten Zeiten für die Konvertierung. Durch einen um den Faktor 10 geringeren Zeitbedarf für I/O gegenüber der Implementierung in C konnte eine etwas längere Rechenzeit mehr als ausgeglichen werden. Der Vorteil des übersichtlicheren Zugriffes auf die einzelnen Elemente der Trace-Datei mußte also nicht mit dem Nachteil eines höheren Zeitbedarfs für die Konvertierung erkauft werden.

Als zweite Möglichkeit ist noch das Konzept eines internen Treibers für das „neue“ Format in *PARvis* zu nennen. Ein interner Treiber wurde im Rahmen dieser Arbeit entwickelt. Ein Vorteil des Konzeptes ist ohne Zweifel die völlige Transparenz für den Benutzer von *PARvis*. Er wählt lediglich die zu analysierende Trace-Datei aus; die Erkennung des Formates und die Auswahl des richtigen Treibers erfolgt durch *PARvis*. Ein weiterer Vorteil ist, daß alle Informationen, die in der speziellen Trace-Datei vorhanden sind, auch ausgewertet werden können. *PARvis* kann um weitere Funktionen ergänzt werden, die vielleicht nur für spezielle Formate zur Verfügung stehen, hier aber eine vollständige Auswertung aller Informationen erlauben (siehe auch Abschnitt 4.1.5). Weiterhin entfällt der zusätzliche Zeitaufwand für die Konvertierung in das *PARvis*-Format, da die gelesenen Trace-Daten direkt in die internen Datenstrukturen von *PARvis* eingefügt werden können. Zu diesem Zweck erhielt *PARvis* für diesen Teilbereich eine neue interne Struktur, die jetzt eine definierte

Treiber-Schnittstelle für ein spezielles Trace-Datenformat vorsieht. Diese Schnittstelle wird in Abschnitt 3.7.2 genauer beschrieben. Zunächst soll jedoch auf die speziellen Gegebenheiten bei der Umstrukturierung von *SVM-Fortran*-Ereignissen für *PARvis* eingegangen werden.

3.7.1 Die Umwandlung von SVM-Fortran-Ereignissen für PARvis

Zur Darstellung der Problematik bei der Abbildung von *SVM-Fortran*-Ereignissen auf *PARvis*-Ereignisse soll noch einmal kurz auf die Generierung der *SVM-Fortran*-Ereignisse durch den *SAM* eingegangen werden. Wie in Abbildung 3.13 gezeigt, verwaltet der *SAM* zwei Puffer.

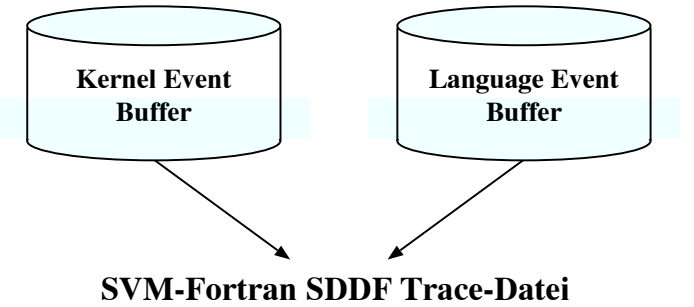


Abbildung 3.13: Interne Pufferstruktur des *SAM*

In einem Puffer werden die *Kernel Event*-Ereignisse zwischengespeichert, in dem anderen die *Kernel Information*- und die *Language Information*-Ereignisse. Die beiden Puffer werden asynchron ausgelesen; dabei wird ihr Inhalt konsekutiv in die Trace-Datei geschrieben. Da jedem Knoten jeweils ein Satz dieser Puffer zugeordnet ist, entstehen bei Ablauf des Programms auf n Knoten n Trace-Dateien. Jede Trace-Datei enthält zwei gemischte, in sich zeitlich sortierte Folgen von Ereignissen. Damit alle Ereignisse in einer Trace-Datei chronologisch sortiert werden können, enthalten alle Ereignisse einen Zeitstempel, realisiert durch ein Feld `"time"` in jeder *Record Descriptor*-Definition. Bei *Read Page Fault*- und *Write Page Fault*-Ereignissen kommt ein weiterer Zeitstempel hinzu: `"time occurred"`. Er gibt an, wann der Seitenfehler aufgetreten ist, während der Eintrag `"time"` den Zeitpunkt angibt, wann die Anforderung der Speicherseite befriedigt wurde. So sind eigentlich in dieser Ereignis-Definition zwei Ereignisse zusammengefaßt: das Auftreten eines Seitenfehlers und die Befriedigung der Seitenanforderung.

Als Sortierkriterium wird der Eintrag in dem Feld `"time"` jedes Ereignisses verwendet. Die im Rahmen dieser Arbeit in der Programmiersprache C++ erstellte Klasse *SortSVMFTrace* führt das Sortieren der Folgen von Ereignissen in jeweils einer Trace-Datei durch. Die damit erzielte Performance war gut. Zunächst wurde zum

Mischen der nun sortierten Trace-Dateien die Klasse *SDDFmerge* der *Pablo* C++-Klassenbibliothek verwendet. Davon wurde jedoch wieder Abstand genommen, da die erzielbare Performance für das Mischen der Trace-Dateien die gesetzten Vorgaben nicht erfüllte. Verzichtet man auf den Vorteil der universellen Anwendbarkeit dieser Klasse auf alle Trace-Dateien im SDDF-Format, sind deutlich bessere Ergebnisse erzielbar. Möglich wird dies, indem Informationen über den Aufbau der zu mischenden Ereignisse im Quelltext der Misch-Funktion „hart codiert“ werden. Da bei *SVM-Fortran*-Ereignissen das Feld `"time"` stets dieselbe Position innerhalb eines Ereignisses hat, kann direkt auf dieses Feld zugegriffen werden. Da die Länge eines Ereignisses in Bytes ebenfalls bekannt ist, kann durch einfaches „umkopieren“ des gesamten Ereignisses dieses an der richtigen Position in der Trace-Datei plaziert werden. Eine semantische Interpretation jedes Ereignisses, wie sie in der Klasse *SDDFmerge* durchgeführt wird, entfällt. Die gesetzten zeitlichen Vorgaben für das Mischen von Trace-Dateien wurden so erfüllt.

Nach der Sortierung der beiden Folgen von Ereignissen und dem Zusammenführen der einzelnen Trace-Dateien in eine Trace-Datei sind die *Read Page Faults*- und die *Write Page Faults*-Ereignisse „aufzulösen“, denn *PARvis* erwartet alle einzelnen Ereignisse in einer chronologischen Reihenfolge. Der Zeitpunkt, an dem ein Seitenfehler auftritt, muß also in die vorhandene Folge von Ereignissen einsortiert werden. Da der zeitliche Abstand zwischen dem Auftreten eines Seitenfehlers und der Befriedigung der Seitenanforderung nicht festgelegt ist, kann ein Einsortieren nur bei im Speicher gehaltenen Ereignissen vorgenommen werden. Als Datenstruktur zur Speicherung der Ereignisse wurde eine doppelt verkettete Liste ausgewählt. Für die minimale Zahl der in dieser Liste zu haltenden Ereignisse ist eine Abschätzung möglich. Dazu werden nicht alle in der Trace-Datei vorhandenen Ereignisse in die Liste aufgenommen, sondern nur die tatsächlich von *PARvis* benötigten. Dies stellt eine Filterung dar. Für die Abschätzung gilt dann:

- Tritt ein Seitenfehler auf, wird zunächst vom ASVM-Kernel mit Hilfe des *Dynamic Forwarding* versucht, die Speicherseite auf einem Knoten zu lokalisieren. Die dafür benötigte Zeit beträgt $Knotenzahl \cdot 0{,}37\ ms$.

- Das Umschalten auf das *Static Forwarding* ist mit $2\ ms$ anzusetzen.

- Die dann ausgeführte *Global Forwarding* Strategie kann ebenfalls mit $Knotenzahl \cdot 0{,}37\ ms$ abgeschätzt werden.

- Am längsten dauert das *Page In* einer auf Platte ausgelagerten Speicherseite, dies dauert $25\ ms$.

- Pro Millisekunde können maximal *3 Page Faults* erzeugt werden [Göb95].

Als Abschätzung ergibt sich also:

$$2 \cdot Knotenzahl \cdot 0{,}37\ ms + 25\ ms + 2\ ms$$

Während der Tests des Treibers für SDDF-Trace-Daten von *SVM-Fortran*-Programmen traten im Rahmen dieser Arbeit mit dieser Abschätzung keine Probleme auf.

Von der Realisierung eines externen Konverters für Trace-Dateien von *SVM-Fortran*-Programmen wurde Abstand genommen, da für jede Speicherseite im Konverter eine Zustandstabelle verwaltet werden müßte. Diese Zustandstabellen sind notwendig, da nur durch sie die für das *PARvis* Trace-Datenformat charakteristischen Zustandswechsel einzelner Speicherseiten aus den Ereignissen der *SVM-Fortran*-Trace-Daten gewonnen werden können. Eine effiziente Konvertierung ist so nicht möglich. Durch die wohldefinierte Schnittstelle für interne Treiber, die *PARvis* inzwischen zur Verfügung stellt, kann solch ein Treiber ohne die aufwendigen Zustandstabellen implementiert werden. Diese Schnittstelle wird im folgenden Abschnitt beschrieben.

3.7.2 Verwendung der Treiber-Schnittstelle in PARvis für SDDF-Trace-Daten

In der Treiber-Schnittstelle ist ein Satz von Funktionen definiert, mit deren Hilfe eine abstrakte Trace-Datei mit ihren Ereignissen verwaltet wird. In *PARvis* sind lediglich die Funktionsprototypen definiert. Für jeden Treiber wird ein Satz von Funktionszeigern erzeugt, die in einem `struct`-Datentyp der Programmiersprache C zusammengefaßt sind. So ist eine eindeutige Zuordnung zwischen Funktionsname und Treiber gewährleistet. Zu diesen Funktionen gehören `Open()`, `Close()`, `ReadNextRecord()` usw. Die Funktionen, die ein Ereignis einlesen, geben einen Zeiger auf das jetzt im Speicher stehende Ereignis zurück. Soll das Ereignis inhaltlich ausgewertet werden, ruft *PARvis* die Funktion `ExecuteRecord()` auf. Sie hat als Parameter den Zeiger auf das im Speicher stehende Ereignis. Innerhalb dieser Funktion werden die internen Datenstrukturen von *PARvis* gefüllt. Dazu stellt *PARvis* spezielle Funktionen bereit, die entweder auf die zur Visualisierung der Trace-Daten von Message-Passing-Programmen notwendigen Datenstrukturen oder auf die entsprechenden Datenstrukturen zur Visualisierung von Speicheraktivitäten zugreifen. Hier werden nicht nur die Funktionsprototypen, sondern auch die dazugehörige Funktionalität zur Verfügung gestellt. Im Gegensatz dazu muß der Treiber die Funktionalität der in *PARvis* definierten Funktionsprototypen der Schnittstelle bereitstellen. Die im vorherigen Abschnitt 3.7.1 erwähnte Datenstruktur einer doppelt verketteten Liste zur zeitlichen Sortierung der einzelnen Ereignisse wird ebenfalls vom Treiber verwaltet. So ist gewährleistet, daß die Eigenheiten des spezifischen Trace-Datenformates für *PARvis* transparent sind. Eine Übersicht über dieses Treiberkonzept gibt Abbildung 3.14.

Für die Realisierung eines SDDF-Treibers für Trace-Daten von *SVM-Fortran*-Programmen bietet dieses Treiberkonzept weiterhin den Vorteil, daß für den Zugriff auf die Trace-Datei im SDDF-Format die *Pablo* C++-Klassenbibliothek genutzt

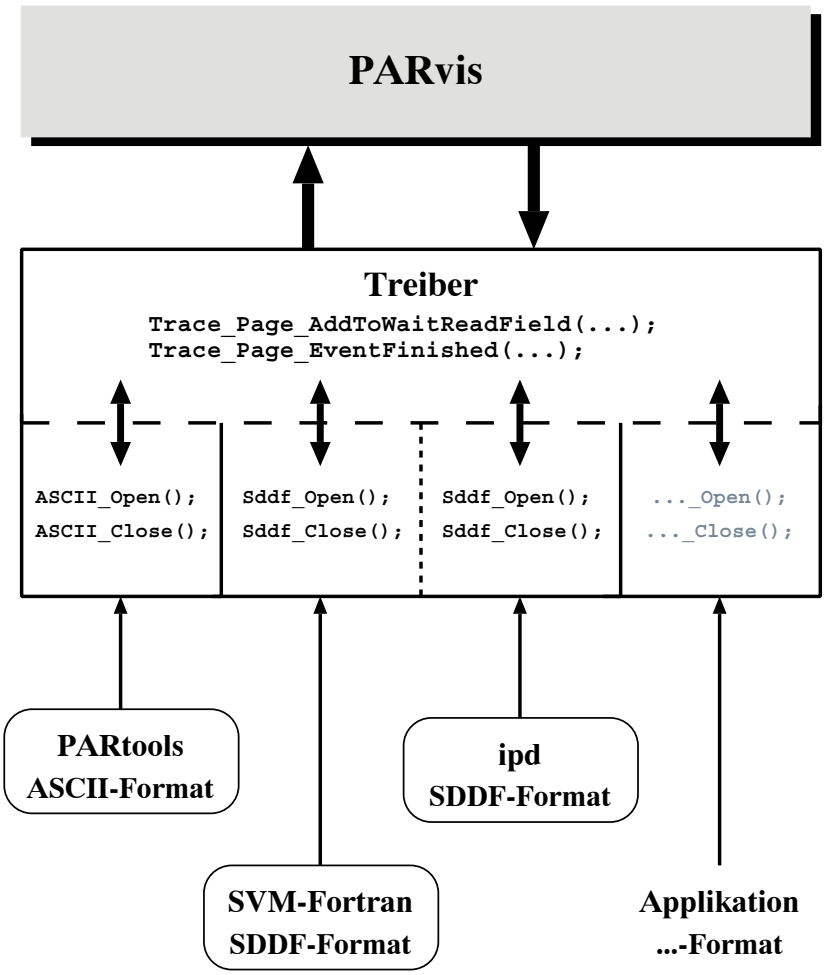


Abbildung 3.14: Treiberkonzept in *PARvis*

werden kann. Die Funktionen sind in C++ programmiert, lediglich der Funktionsprototyp ist als `extern "C"` deklariert. So wird der C++-Compiler veranlaßt für diesen Funktionsnamen keine erweiterten Symbolnamen zu erzeugen (sogenanntes „name mangling"). Die Integration von C++-Quelltext in den in C verfaßten Quelltext von *PARvis* funktioniert damit problemlos. Um beim Linken der einzelnen Objektmodule die erweiterten Symbolnamen der verwendeten Methoden aus der C++-Klassenbibliothek zuzuordnen, muß statt der bisher verwendeten C-Linkers ein C++-Linker zum Einsatz kommen.

Neben dem oben beschriebenen Treiber wird als Fortführung dieser Arbeit ein zusätzlicher Treiber für Trace-Daten von Message-Passing-Programmen implementiert werden. Die Erzeugung dieser Trace-Daten erfolgt mit Hilfe des *ipd* ebenfalls im SDDF-Format. Geplant ist weiterhin die Realisierung eines dritten Treibers für

Trace-Daten im SDDF-Format, die mit der *Pablo* Trace-Capture-Bibliothek generiert wurden.

Nachdem in diesem Kapitel der „physikalische“ Zugriff auf die Trace-Daten mit Hilfe eines speziellen Treibers beschrieben wurde, geht es im nächsten Kapitel um die Visualisierung der in den Trace-Daten enthaltenen Informationen. Es wird dargestellt, inwieweit die bereits in *PARvis* vorhandenen Visualisierungskomponenten für die Darstellung der Trace-Informationen von *SVM-Fortran*-Programmen genutzt werden können. Erweiterungen der Visualisierungskomponenten zur Anzeige der in speziellen Ereignissen enthaltenen Informationen sind ebenfalls in dem nächsten Kapitel (Abschnitt 4.1.5) beschrieben.

Kapitel 4

Komponenten zur Darstellung von SVM-Informationen

Die bisher in *PARvis* vorhandenen Komponenten zur Darstellung von SVM-Informationen leiten sich aus den Erfordernissen der in der PARtools-Umgebung vorhandenen SVM-Komponente des Parallelrechnersimulators *PARsim* ab. Ausgehend von einer Übersicht, in der eine beliebige Zahl von Speicherseiten ausgewählt werden kann, können weitere Darstellungen aktiviert werden. Diese gliedern sich in zwei Gruppen:

Zu den *Zeitpunktdarstellungen* gehören

- die CPU-orientierten Seitenaktivitäten,
- die CPU-/Speicherseiten-Tabellen und
- die Kommunikation im Verbindungsnetzwerk.

Bei den *Zeitachsendarstellungen* hat man die Möglichkeit, zwischen

- Kommunikationszeitlinien und
- Aktivitätszeitlinien

zu wählen. Hervorzuheben ist, daß sich sämtliche Darstellungsmöglichkeiten in die *PARvis*-Philosophie integrieren. Das schließt nicht nur ein einheitliches Erscheinungsbild und eine einheitliche Bedienungsoberfläche ein, sondern auch die Bereitstellung der *PARvis* auszeichnenden Funktionen (wie z.B. Zooming). Für eine genaue Beschreibung der oben angeführten Darstellungen wird auf [Mül94] und [Arn95] verwiesen.

Trotz der geänderten Struktur der Trace-Daten von *SVM-Fortran*-Programmen gegenüber den bisher von *PARvis* auswertbaren Trace-Daten lassen sich die oben angegebenen Darstellungen auch bei der Auswertung dieser Daten verwenden. Weiterhin

bot sich die Möglichkeit, die Nutzung der in *PARvis* integrierten Statistikanzeigen für die Trace-Daten von *SVM-Fortran*-Programmen zu erweitern. Die dazu notwendigen Trace-Informationen werden in entsprechenden Ereignissen durch den *SAM* protokolliert [Özm95]. Diese und andere Erweiterungen von *PARvis* zur Darstellung der Trace-Informationen werden im folgenden Kapitel vorgestellt.

4.1 Angleichung der SVM-Funktionalität an den PARvis-Standard

Bisher fehlen den SVM-Komponenten von *PARvis* einige Darstellungsmöglichkeiten, die beispielsweise bei der Analyse von Trace-Daten von Message-Passing-Programmen zur Verfügung standen. Bei der Auswertung der Trace-Daten von *SVM-Fortran*-Programmen lassen sich, bedingt durch die andere Struktur dieser Daten, einige der vorhandenen Komponenten nicht mehr in der bisherigen Form nutzen. Bei der Anpassung oder Erweiterung der vorhandenen Visualisierungskomponenten von *PARvis* wurde im Rahmen dieser Arbeit darauf Wert gelegt, daß sie sich in ihrem „Look and Feel“ in die *PARvis*-Umgebung integrieren.

4.1.1 Nutzung der Kommunikationszeitlinien-Darstellung

Entsprechend dem in der PARtools-Umgebung integrierten SVM-Simulator kennt *PARvis* bestimmte Zustände einer Speicherseite, unterteilt in

- Zustandsinformationen und
- Zustandsänderungen [Mül94].

Nach dem in dem SVM-Simulator implementierten SVM-Modell gibt es einen *Manager*, der für die Administration der Speicherseiten zuständig ist. Ein Prozessor wird als Besitzer *(Owner)* einer Speicherseite geführt. Je nach verwendeter Kohärenzstrategie gibt es Prozessoren, die das alleinige Schreibrecht für eine Speicherseite haben (starke Kohärenz), oder aber mehrere Prozessoren können gleichzeitig schreibend auf eine Kopie der Speicherseite zugreifen (schwache Kohärenz) [Ott95]. Der *Manager* besitzt auch die Informationen über die Verteilung von Speicherseitenkopien, auf die von den Prozessoren nur lesend zugegriffen wird. Tritt während der Programmabarbeitung bei einem Prozessor ein Seitenfehler auf, fordert er die entsprechende Speicherseite beim *Manager* an. Der anfordernde Prozessor geht in den Zustand „Warten-auf-Seite“ über. Dieser Zustand wird wieder verlassen, sobald das Versenden der Speicherseite beginnt. Solange die Speicherseite noch nicht vollständig eingetroffen ist, befindet sich der Prozessor in einem „Wechsel“-Zustand. Ist die Speicherseite eingetroffen, erreicht der Prozessor den Zustand, der den Zugriffsrechten der angeforderten Speicherseite entspricht. Die Protokollierung dieser Ereignisse

in einer Trace-Datei im *PARvis*-Format ist in Abschnitt 3.3 beschrieben. In Abbildung 4.1 sind diese Zustandsübergänge dargestellt.

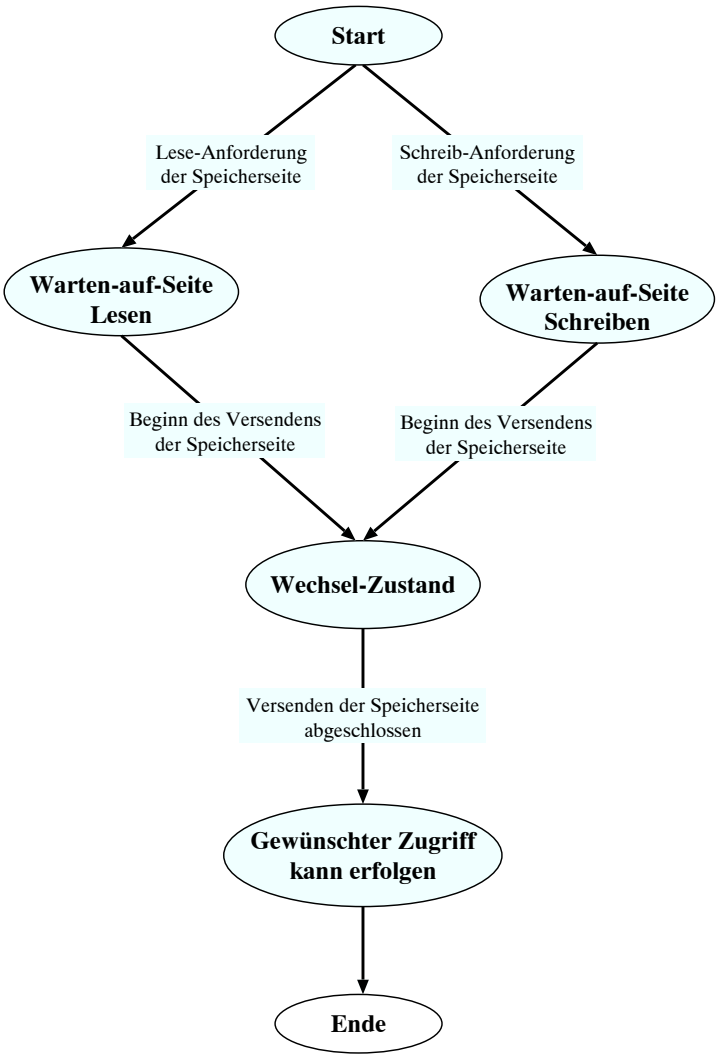


Abbildung 4.1: Zustandsübergänge beim Versenden einer Speicherseite

Gemäß der Definition bei Trace-Daten von *SVM-Fortran*-Programmen wird das Auftreten eines Seitenfehlers und die Befriedigung der Seitenanforderung in einem Ereignis protokolliert. Nach den oben beschriebenen Zustandsübergängen müßte der Prozessor in den Zustand „Warten-auf-Seite“ übergehen, wenn der Seitenfehler aufgetreten ist. Da der Start des Versendens nicht erfaßt wird, überspringt der Prozessor sozusagen den „Wechsel“-Zustand. Er geht nach dem Eintreffen der Seiten direkt in den Lese- oder Schreibzustand über. Die Visualisierung des Anfangs- und Endpunktes des Versendens einer Speicherseite ist in der Kommunikationszeitlinien-Darstellung dadurch nicht möglich. Um diese Visualisierung dennoch nutzen zu können, wird bei dem in Rahmen dieser Arbeit implementierten SDDF-Treiber für *PARvis* der Zeitpunkt des Auftretens eines Seitenfehlers

als Versendebeginn der Speicherseite interpretiert. Der anfordernde Prozessor geht also direkt in den „Wechsel"-Zustand über. Abbildung 4.2 zeigt einen Ausschnitt aus der Kommunikationszeitlinien-Darstellung eines *SVM-Fortran*-Programms.

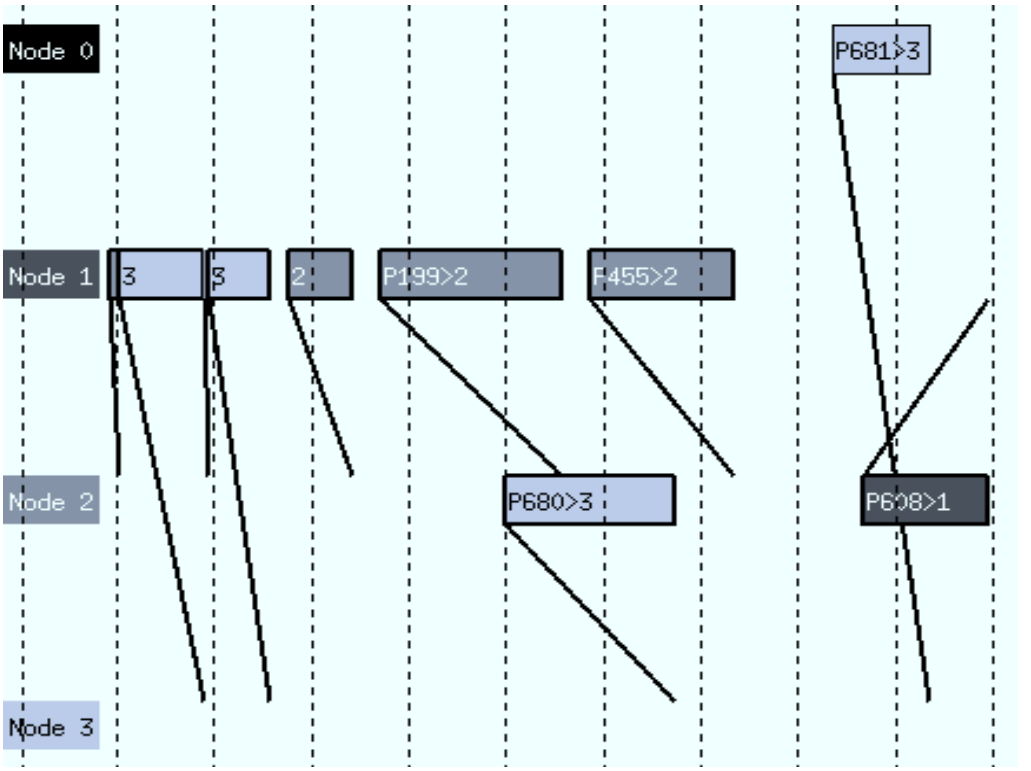


Abbildung 4.2: Kommunikationszeitlinien-Darstellung am Beispiel eines *SVM-Fortran*-Programms

Links in der Abbildung sind die an der Programmausführung beteiligten Prozessoren zu erkennen. Jedem von ihnen ist eine Farbe zur Identifikation zugeordnet (in der Abbildung durch Graustufen dargestellt). Die gestrichelten vertikalen Linien beschreiben ein Zeitraster, die horizontalen Balken kennzeichnen den Versendevorgang von Speicherseiten. Dabei wird die Länge eines Balkens aus der Dauer des Versendevorganges bestimmt. Ist der Balken lang genug, ist in ihm die Nummer der Speicherseite (mit einem vorangestellten `P`) und der Zielprozessor angegeben. Der Eintrag `P680>3` bedeutet also, daß die Speicherseite 680 an Prozessor 3 verschickt wird. Ist der Balken zu kurz, wird nur die Nummer der Speicherseite in ihm vermerkt. Die Identifikation des empfangenden Prozessors geschieht dann über die Farbe des Balkens. Als Hilfe für den Betrachter dienen die schwarzen Verbindungslinien. Sie markieren Ausgangs- und Zielknoten bei der Versendung der Speicherseite. Da dieser Vorgang über der Zeit aufgetragen wird, gibt die Steigung der Verbindungslinie einen Anhaltspunkt für die Geschwindigkeit, mit der die Übertragung der Speicherseite vonstatten ging.

In der Aktivitätszeitlinien-Darstellung und der CPU-/Speicherseiten-Tabelle in *PARvis* ist dagegen der „Wechsel"-Zustand *(Changing)* durch eine Schraffur der

entsprechenden Bereiche kenntlich gemacht. In [Mül94] findet sich eine genaue Beschreibung der hier angesprochenen Darstellungsmöglichkeiten.

4.1.2 Anzeige von Ereignissen in ASCII-Form

Da die bisher von *PARvis* ausgewerteten Trace-Dateien im ASCII-Format vorlagen, ließ sich die Anzeige eines einzelnen Ereignisses sehr einfach realisieren. Dazu war es ausreichend, die aktuelle Zeile der Trace-Datei in einem Editor anzuzeigen. Eine Bearbeitung der Trace-Datei in dem Editor war nicht vorgesehen.

Durch die gänzlich andere Struktur der Trace-Dateien von *SVM-Fortran*-Programmen scheidet die direkte Anzeige der Trace-Informationen in *PARvis* durch einen Editor aus. Der Anwender müßte die Definition des zu bearbeitenden Ereignisses genau kennen, um die einzelnen Datenfelder eines Ereignisses richtig zu interpretieren. Zur Verdeutlichung hier ein *Write Page Fault*-Ereignis, wie es in einer Trace-Datei protokolliert ist:

```
"write-page-fault-serviced" { 0, 4366.379938, 4366.377927
                              1074169472, 3 };;
```

Ohne die genaue Kenntnis der Definition des *Write Page Fault*-Ereignisses sind diese Angaben nicht zu verstehen. Daher wird innerhalb des SDDF-Treibers von *PARvis* eine Aufbereitung der in einem Ereignis enthaltenen Informationen vorgenommen. Durch die *Pablo* C++-Klassenbibliothek wird gegebenenfalls eine Umsetzung der binären Trace-Daten in eine ASCII-Form durchgeführt. Danach erfolgt im Treiber die Kombination der Ereignisdefinition mit den Daten des ausgewählten Ereignisses. Diese formatierte Zeichenkette wird vom Treiber an *PARvis* übergeben und dort in einem Fenster angezeigt.

Die große Zahl von annehmbaren Fehlerquellen hat es auch hier nicht sinnvoll erscheinen lassen, dem Benutzer eine Bearbeitung einzelner Ereignisse zu ermöglichen.

In Abbildung 4.3 ist die Anzeige von Ereignissen in ASCII-Form durch *PARvis* dargestellt. In dieser Form sind die Angaben für den Anwender wesentlich leichter zu interpretieren als in der oben angegeben Darstellungsweise.

4.1.3 Symbol- und Regionennamen

Neben den bisher bereits von *PARvis* ausgewerteten Symbolnamen bietet sich bei den Trace-Daten von *SVM-Fortran*-Programmen die Möglichkeit, den Quelltextbezug durch die Auswertung weiterer Ereignisse zu verbessern. Die Interpretation der Trace-Information wird wesentlich erleichtert, wenn feststellbar ist, an welcher Stelle im Quelltext ein Ereignis protokolliert wurde. Der hier beschriebene Zugriff auf Symbol- und Regionennamen ist allerdings nur ein erster Schritt in die Richtung

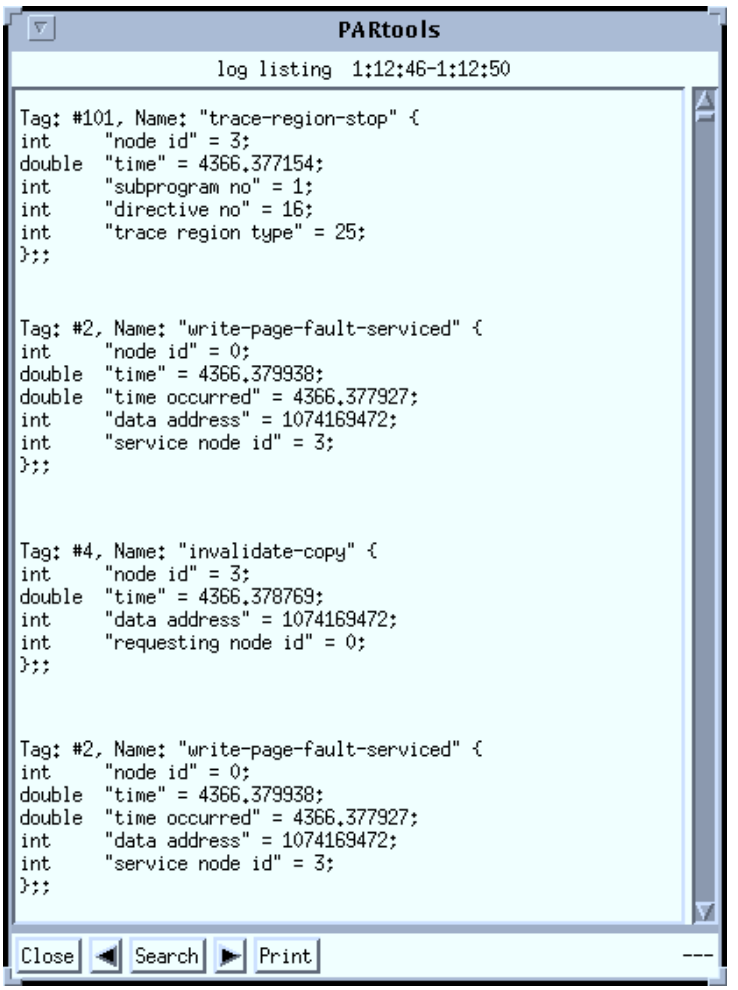


Abbildung 4.3: Anzeige von Ereignissen in ASCII-Form

auf einen vollständigen Quelltextbezug. In Kapitel 5 werden neue Komponenten von *PARvis* beschrieben, die Optionen eines erweiterten Quelltextbezuges aufzeigen.

Während bisher die Symbolnamen aus dem im *COFF*-Format vorliegenden Objektmodul gewonnen wurden, liegen diese Informationen nun in der Trace-Datei vor. Generiert werden die als *Compiler Information Data* bezeichneten Informationen während der Übersetzung eines Programms durch den *SVM-Fortran*-Compiler. Dieser schreibt Daten in die Datei `svmf_trace.dat`, die von dem *SAM* während der Ausführung des Programms interpretiert wird.

Der Bezug zwischen Quelltext und Trace-Information wird durch die Einteilung des Quelltextes in verschiedenen Ebenen hergestellt:

- Informationen über die Programmeinheiten (Unterprogramme bzw. Funktionen) des Programms,
- Informationen über die Variablen in jeder Programmeinheit,

- Zähler für das gesamte Programm, die die Zahl paralleler Regionen und *SVM-Fortran*-Direktiven innerhalb einer Programmeinheit festhalten,
- Informationen über die parallelen Regionen in jeder Programmeinheit und
- Zähler für Barrieren innerhalb des gesamten Programms.

Diese Bezugsebenen werden in einem geeigneten Schema durchnumeriert, welches in [Kru95] beschrieben ist.

Zur Anzeige der Symbol- und Regionennamen in *PARvis* werden die „Informationen über das Programm“ (`PROGID`) und die „Definitionen der parallelen Regionen in jeder Programmeinheit“ (`REGIONDEF`) verwendet. Die `PROGID` ordnet dem Namen einer Programmeinheit eindeutig eine Nummer zur Identifikation zu. In `REGIONDEF` ist für jede Programmeinheit der Regionentyp (beispielsweise eine `PDO`-Schleife), die Nummer der Region innerhalb der Programmeinheit und die Nummer der entsprechenden *SVM-Fortran*-Direktive festgehalten. Läuft das Programm während der Abarbeitung in eine parallele Region, wird ein entsprechendes Ereignis generiert. Beim Verlassen dieser Region wird erneut ein Ereignis erzeugt. Parallele Regionen können beliebig ineinander verschachtelt sein. Daher ist es bei der Verarbeitung der *SVM-Fortran*-Ereignisse durch den SDDF-Treiber notwendig, das beim Erreichen einer Region erzeugte Ereignis in einer `Stack`-Datenstruktur zwischenzuspeichern. Wird eine Region wieder verlassen, wird einfach das oberste Element vom `Stack` entfernt. So erhält man immer zusammengehörige Paare von Ereignissen, mit deren Hilfe *PARvis* die dazugehörenden Namen richtig anzeigen kann. Ein Vergleich der Daten des obersten Elementes des `Stacks` mit denen des `trace-region-stop`-Ereignisses erlaubt eine Konsistenzkontrolle. Da der SDDF-Treiber in der Programmiersprache C++ implementiert ist, wird der `Stack` durch ein entsprechendes `Stack`-Objekt realisiert. Für alle an der Programmausführung beteiligten Prozessoren wird ein Feld von `Stack`-Objekten bereitgestellt, so daß jeder Prozessor über ein eigenes `Stack`-Objekt verfügt.

Die in dem *Compiler Information File* vorhandenen Daten werden in *Stream Attribute*-Paketen (siehe Abschnitt 3.4) der Trace-Datei gespeichert. Ein *Stream Attribute* besteht aus einer *Attribute List*, die wiederum aus einer beliebigen Zahl von *Attribute Name*- und *Attribute Value*-Paaren besteht [Ayd94]. Ein Ausschnitt aus der Trace-Datei eines *SVM-Fortran*-Programms soll dies verdeutlichen (Darstellung im ASCII-Format):

```
/*
~* " compiler information" ""
~* " PROGIDCNT 10" ""
~* " PROGID SVM_GENERATED_MAIN 0" ""
~* " PROGID NEURAL 1" ""
~* " PROGID INPATS 2" ""
~* " PROGID HEBB 3" ""
```

```
~* " PROGID UPDATE 4" ""
~* " PROGID INIT_S 5" ""
~* " PROGID RECOMB 6" ""
~* " PROGID ENDE 7" ""
~* " PROGID RANMAR 8" ""
~* " PROGID RMARIN 9" ""
...
~* " REGIONDEF HEBB PDO 1 3" ""
~* " REGIONDEF HEBB PDO 2 4" ""
~* " REGIONDEF HEBB PDO 3 5" ""
...
~* " end_cif" ""
~*/ ;;
```

Auf das Schlüsselwort `PROGID` folgt ein Programmeinheitenname und die entsprechende Programmeinheitennummer. Weiterhin kann man erkennen, daß in der Programmeinheit `HEBB` eine parallele Region vom Typ `PDO` definiert ist. Die beiden Zahlen hinter dem Regionentyp geben die Nummer der Region und der entsprechenden *SVM-Fortran*-Direktive innerhalb der Programmeinheit an.

Wie aus dem obigen Ausschnitt zu erkennen ist, wird in der *Attribute List* nur der *Attribute Name* verwendet, der *Attribute Value* bleibt leer. Dies hat mit der Implementierung der *Pablo* C++-Klassenbibliothek zu tun. Es ist nicht möglich, einem *Attribute Value* mehrere verschiedene *Attribute Names* über Methoden aus der Klassenbibliothek zuzuordnen, obwohl dieses in der SDDF-Grammatik nicht explizit ausgeschlossen ist [Ayd94].

Das Werkzeug *OPAL* wertet ebenfalls die *Compiler Information Data* in Trace-Dateien von *SVM-Fortran*-Programmen aus. Dort werden, im Gegensatz zu dem hier beschriebenen SDDF-Treiber für *PARvis*, alle oben aufgeführten Informationen verwendet. In *OPAL* kann auf diese Weise ein weiterreichender Bezug zwischen instrumentiertem Quelltext und den dazugehörenden Ereignissen erfolgen. Durch eine gezielte Instrumentierung von bestimmten *SVM-Fortran*-Direktiven in Unterprogrammen oder Funktionen wird beispielsweise die Menge der erzeugten Trace-Daten beschränkt. Durch die Auswertung der entsprechenden Ereignisse in *OPAL* ist die quelltextbezogene Anzeige von Trace-Informationen, wie unter anderem die Summe von *Read Page Faults* in einer parallelen Region, möglich (siehe auch Abschnitt 5.1).

Die Auswertung der Symbol- und Regionennamen kann nur ein erster Schritt in Richtung einer vollständig quelltextbasierten Performance-Analyse sein. Der Anwender erhält so eine Orientierung über den Programmabschnitt, den er gerade mit *PARvis* analysiert (siehe Abbildung 4.4). Er kann jedoch noch nicht feststellen, welche Schleifeniteration gerade ausgeführt wurde oder welche Variablen sich auf einer bestimmten Speicherseite befinden. Wie bereits erwähnt, werden in Kapitel 5 hier erstellte Erweiterungen von *PARvis* vorgestellt, die dem Anwender diese Informationen liefern.

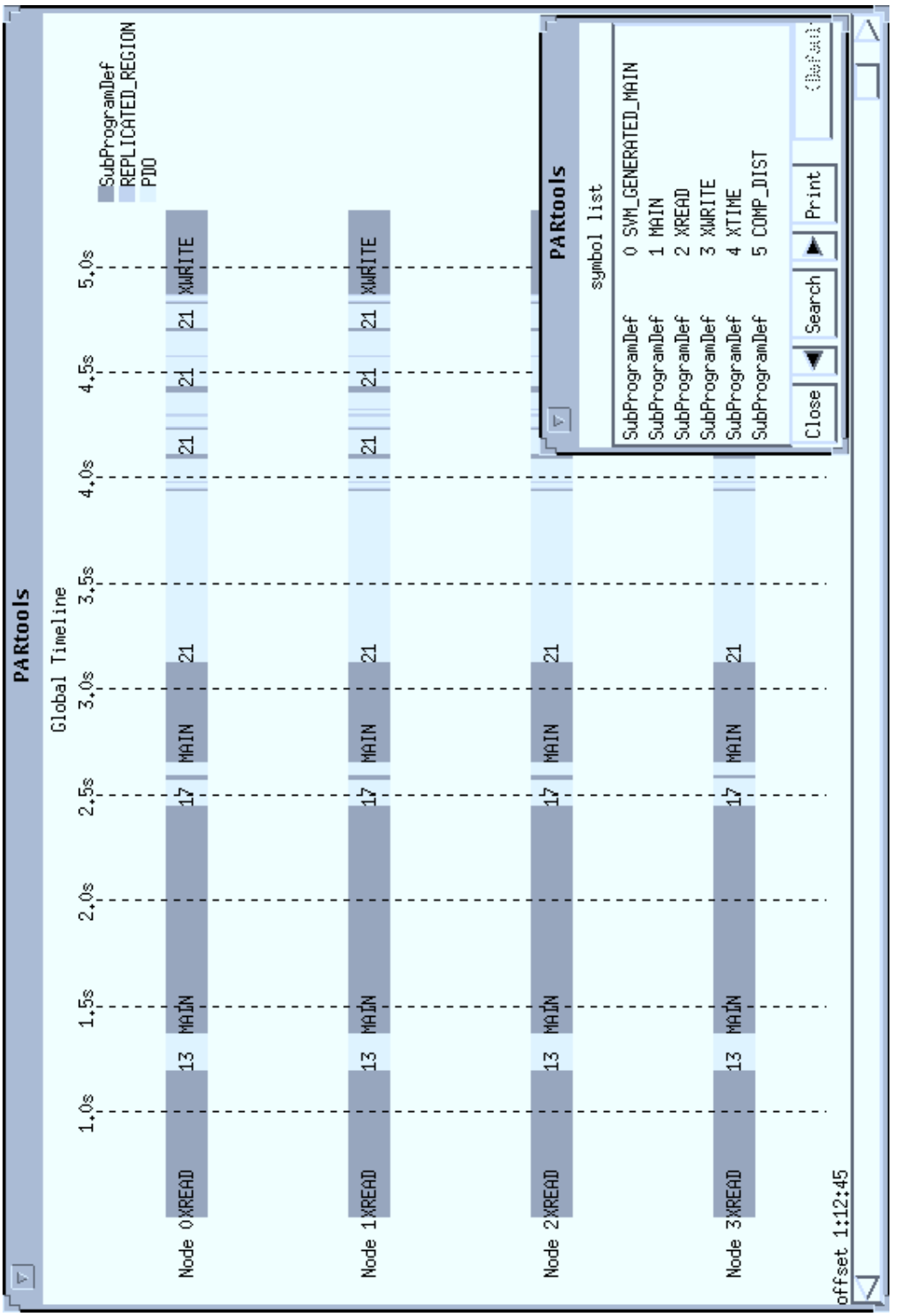


Abbildung 4.4: Darstellung von Symbol- und Regionennamen in der globalen Zeitlinie und der Symbolliste

4.1.4 Statistische Auswertungen

PARvis bietet dem Anwender bereits jetzt die Möglichkeit, auf verschiedene statistische Auswertungen zuzugreifen. Dazu gehören

- die Zahl der Wechsel in einen Zustand bei *Prozessoren*,
- die akkumulierten Zeiten für diese Zustandswechsel bei *Prozessoren*,
- die Zahl, durchschnittliche und gesamte Länge sowie die durchschnittliche Geschwindigkeit von *Nachrichten* und
- der Anteil der Kommunikationszeit einer *Speicherseite* an der gesamten Rechenzeit.

Dabei ist nur der letzte Punkt der obigen Aufzählung bei der Analyse von SVM-Trace-Daten verfügbar. Die bei den anderen Punkten angegebenen statistischen Auswertungen für Trace-Daten von Message-Passing-Programmen werden noch um die Anzeige der jeweiligen Minima und Maxima erweitert werden. Die für diese statistischen Angaben benötigten Daten werden intern von *PARvis* während der Auswertung der Trace-Daten erstellt. Die Trace-Daten von *SVM-Fortran*-Programmen bieten hier zusätzliche Möglichkeiten, denn während der Ausführung eines Programms wertet der *SAM* bereits Zähler des ASVM-Kernels der Intel Paragon aus [Özm95]. Daraus ergeben sich die folgenden Ereignisse [Kru95]:

```
#60 read-page-fault-sum
#61 write-page-fault-sum
#62 write-upgrade-sum
#63 page-in-pager-sum
#64 page-discard-sum
#65 page-out-node-sum
#66 page-out-pager-sum
#54 page-travel
```

Diese Zähler laufen nicht akkumulierend während der gesamten Programmausführung mit, sondern nur in den vorher instrumentierten Regionen. Wird eine dieser Regionen erneut durchlaufen, beginnt die Summierung der entsprechenden Ereignisse wieder bei Null, da alle Summenzähler beim Verlassen einer Region zurückgesetzt werden. Vor dem Zurücksetzen wird der Inhalt der Zähler in das dazugehörige Summen-Ereignis geschrieben. Innerhalb des SDDF-Treibers werden keine akkumulierenden Zähler verwaltet, da über die Instrumentierung des Programms nichts bekannt ist. Die in solchen Zählern ermittelten Summen wären wenig aussagekräftig, da nicht sichergestellt ist, daß tatsächlich alle Regionen instrumentiert sind.

Zur Anzeige dieser Summen wurde im Rahmen dieser Arbeit eine neue Statistik-Komponente in *PARvis* implementiert. Mit ihrer Hilfe ist ein rasches Auffinden von

Regionen mit überdurchschnittlich vielen Seitenfehlern möglich. In der Darstellungsform der Daten lehnt sie sich an die bereits vorhandenen an. So können die ermittelten Summen in Form von Säulendiagrammen und in Tabellenform[1] visualisiert werden. Zur Benutzung dieser Komponente ist in dem Fenster *Global Timeline* zunächst eine Auswahlliste zu aktivieren. Unter dem Menüpunkt **Options** ist der Punkt **Show Page Statistics** zu wählen. Danach kann innerhalb der Zeitlinie eines Prozessors die gewünschte Region markiert werden. Von *PARvis* wird ein neues Fenster geöffnet, welches die entsprechenden Daten zunächst für die ausgewählte Inkarnation[2] der Region darstellt. Vorgesehen ist die Erweiterung dieser Darstellung, in der dann automatisch alle Inkarnationen der ausgewählten Region angezeigt werden. Innerhalb des Statistik-Fensters kann der Benutzer dann zwischen den beiden oben angegebenen Darstellungsformen wählen (**Histogram** oder **Table**). Die Aktivierung dieses Menüs erfolgt, wie überall in *PARvis*, mit der rechten Maustaste. In Abbildung 4.5 ist die Auswertung der Summen-Ereignisse vom Typ `read-page-fault-sum` und `write-page-fault-sum` als Säulendiagramm dargestellt. Man erkennt den Namen der Programmeinheit bzw. die in der parallelen Region verwendete *SVM-Fortran*-Direktive. Da von jedem an der Programmausführung in der betreffenden Region beteiligtem Prozessor ein Ereignis dieses Typs generiert wird, kann über die Ausgewogenheit des Säulendiagramms eine Aussage über die Arbeitsverteilung in der Programmregion getroffen werden.

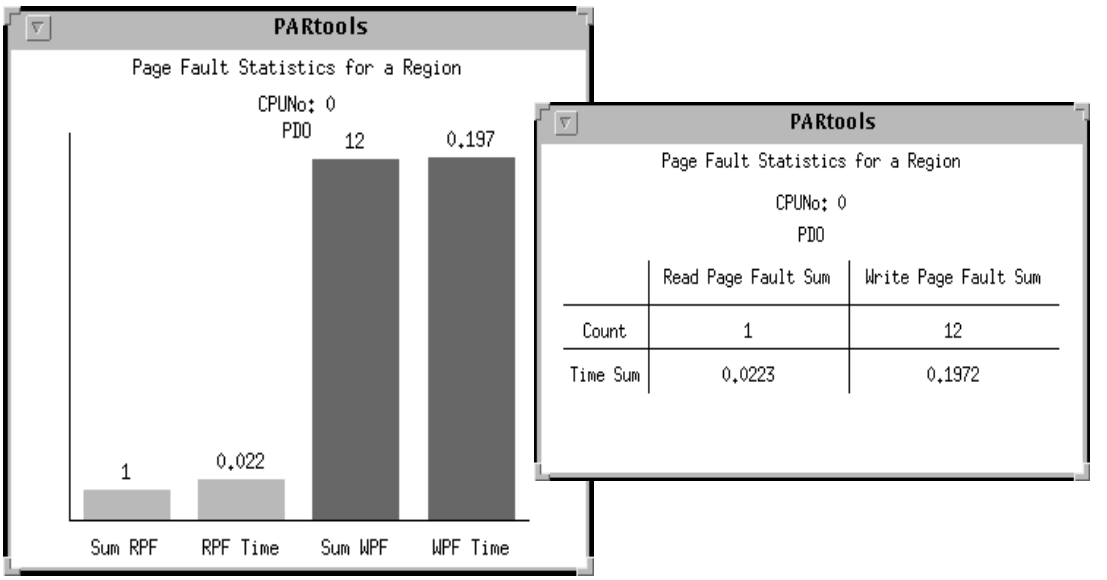


Abbildung 4.5: Darstellung von Summen-Ereignissen in *PARvis*

Zur Anzeige dieser Daten ist die Treiber-Schnittstelle von *PARvis* um die Funktion `Trace_Page_AddRegionStatistic()` erweitert worden. Diese wird von dem SDDF-Treiber immer dann aufgerufen, wenn ein neuer Datensatz innerhalb des Treibers

[1] Diese wird auch in *OPAL* verwendet (siehe Abschnitt 5.1).
[2] Unter einer *Inkarnation* versteht man eine einmal durchlaufene Region.

komplettiert werden konnte. Ein solcher Datensatz ist als `struct`-Datentyp in der Programmiersprache C realisiert und hat die folgenden Elemente:

```
ClockType Time;        /* Data from event #101 trace-region-stop */
CPU_Activity Act;      /* Data from event #101 trace-region-stop */
CPU_Act_Num ANum;      /* Data from event #101 trace-region-stop */
LongInt CountRPF;      /* Data from event #60 read-page-fault-sum */
Double SumTimeRPF;     /* Data from event #60 read-page-fault-sum */
LongInt CountWPF;      /* Data from event #61 write-page-fault-sum */
Double SumTimeWPF;     /* Data from event #61 write-page-fault-sum */
```

Die oben angeführten Ereignisse `trace-region-stop`, `write-page-fault-sum` und `read-page-fault-sum` liefern die Daten für diesen `struct`-Datentyp. Aus der in den Ereignissen vorhandenen `subprogram no` und der `directive no` werden die `CPU_Activity` und die `CPU_Act_Num` gebildet, mit deren Hilfe in *PARvis* die Unterprogramm-Namen und die *SVM-Fortran*-Direktive angezeigt werden können. Mit diesen Daten wird in *PARvis* eine Liste für jeden Prozessor aufgebaut, die chronologisch sortiert ist. Wird in der Trace-Datei rückwärts gesprungen, muß der Treiber-Zustand an der entsprechenden Stützstelle der Historienliste in *PARvis* restauriert werden [Arn93]. Die Liste mit den Summen-Ereignissen ist nicht in den Restaurationsvorgang eingebunden, da ein Element der Liste über den Suchschlüssel `time` jederzeit durch entsprechende Suchstrategien effizient lokalisiert werden kann.

Sobald solch ein Datensatz während des Vorauslesens von Ereignissen im SDDF-Treiber mit neuen Daten „gefüllt“ wurde, wird er von ihm direkt an *PARvis* weitergereicht. Ein Eintrag in die interne Ereignisliste des Treibers (siehe Abschnitt 3.7.1) findet nicht statt. Dies ist nicht notwendig, da diese Ereignisse im Gegensatz zu den Seitenfehler-Ereignissen nur einen Zeitstempel enthalten. Der SDDF-Treiber übernimmt lediglich die Aufgabe, die aus drei der oben angeführten zueinander passenden Ereignissen[3] gebildete Datenstruktur zu füllen. Um eine Zuordnung der von *PARvis* ermittelten zeitlichen Position innerhalb der Trace-Datei zu der von dem Benutzer angewählten Region zu erreichen, wird in das Feld `time` die Zeit aus dem Ereignis, das das Ende der Region markiert, eingetragen. Da diese Datensätze *PARvis* vor den anderen Trace-Informationen erreichen (sie stehen ja nicht in der internen Ereignisliste), kann damit gerechnet werden, daß die Aktivierung des Statistikfensters bereits möglich ist, auch wenn die entsprechende Region in der globalen Zeitlinie noch nicht abgeschlossen ist. Dies ist der Fall, wenn mit der Performance-Analyse bereits während des Einlesens der Trace-Daten begonnen wird.

Bei der Entwicklung der neuen statistischen Anzeigen von *PARvis* für die Trace-Daten von *SVM-Fortran*-Programmen wurde auf eine vollständige Integration Wert gelegt. Das bedeutet, daß sich die Anzeigen in Form und Funktionalität nicht von den bereits vorhandenen Komponenten unterscheiden.

[3] Das heißt daß sie in den Feldern `node id`, `subprogram no` und `directive no` identische Einträge aufweisen.

4.1.5 Verteilungen von Speicherseiten

Die Ereignisse vom Typ `page-travel` gehören eigentlich nicht in die Gruppe der „Statistik-Ereignisse“ (aus diesem Grund auch die nicht-konsekutive Numerierung dieses Ereignis-Tags). Dennoch liefert dieses Ereignis für eine statistische Auswertung interessante Informationen. Mit diesem Ereignis wird getrennt für jeden Prozessor festgehalten, wieviele Speicherseiten er innerhalb der instrumentierten Region an einen bestimmten Prozessor verschickt hat. Mit Hilfe dieses Ereignisses kann das sogenannte „Seitenflattern“ *(Page Thrashing)* entdeckt werden. Darunter ist der abwechselnde Schreib- und Lese-Zugriff verschiedener Prozessoren auf ein und dasselbe Element einer Speicherseite zu verstehen, was zur Folge hat, daß die Speicherseite ständig zwischen zwei Prozessoren hin und her geschickt wird. Dieser Effekt des Seitenflatterns ist unbedingt zu vermeiden, da aus ihm eine starke Verschlechterung der Performance resultiert. Ein erster Hinweis auf das Auftreten von Seitenflattern ist eine hohe Zahl von Seitenfehlern innerhalb einer Region. Dies kann durch Betrachtung der entsprechenden Summen-Ereignisse entdeckt werden. Durch Auswertung des `page-travel`-Ereignisses kann der Anwender seine Beobachtung verifizieren. Sind in den Datenfeldern des Ereignisses für eine Speicherseite sehr viele Seitenfehler für zwei Prozessoren verzeichnet, so ist dies ein sicheres Indiz für das Seitenflattern [GKÖ95].

Zur Visualisierung der `page-travel`-Ereignisse wurde das im vorigen Abschnitt beschriebene in *PARvis* eingeführte Statistik-Fenster erweitert. Dieses Fenster ist nach Implementierung der Ereignisse vom Typ `page-travel` im *SAM* nutzbar. Da aber die Ereignisdefinition bereits feststeht [Kru95], ist der in *PARvis* zur Visualisierung notwendige Programm-Code bereits implementiert worden. Dem Anwender stehen für die Darstellung der Daten des `page-travel`-Ereignisses ebenfalls zwei Möglichkeiten zur Verfügung. Es sind dies wieder ein Säulendiagramm und die Tabellenform. Erkennt der Anwender, daß sich in einer Region Seitenfehler einer Speicherseite für zwei Prozessoren häufen, kann er die gewählte Darstellungsform um die Anzeige des `page-travel`-Ereignisses ergänzen. Im nächsten Schritt kann er von diesem Fenster aus direkt die zu dieser Region gehörende Kommunikationszeitlinien-Darstellung *(CPU-Page-Communication)* aktivieren. Durch die graphische Visualisierung des Zugriffes einzelner Prozessoren auf eine Speicherseite ist dann eventuell aufgetretenes Seitenflattern unmittelbar zu erkennen.

Diese Vorgehensweise hat den Vorteil, daß sie in bezug auf die Granularität der präsentierten Informationen von einer groben zu einer feinen Darstellung übergeht. Dies entspricht einer *Top Down*-Vorgehensweise. Die grobe, aber übersichtliche Darstellung in Form eines Säulendiagramms ermöglicht die leichte Identifikation von Regionen, in denen das Seitenflattern auftritt. Außerdem geht man von den Regionen aus, also von dem Quelltext des Programms. Die reine Kommunikationszeitlinien-Darstellung visualisiert die Information über der Zeit. Interessiert man sich nur für das Verhalten einer bestimmten Region, muß der gesamte Zeitabschnitt in der Trace-Datei untersucht werden. Anders sieht man mit einem Blick, ob die ausgewählte Region einer genaueren Untersuchung bedarf.

Wie in diesem Abschnitt bereits erwähnt, lassen sich aus bestimmten statistischen Daten Rückschlüsse auf den Quelltext eines Programms ziehen. Eventuell vorhandene Fehler in der Programmkonzeption können so entdeckt werden. Bei der Auswertung von Trace-Daten können Statistik-Programme von Nutzen sein. Damit befaßt sich der nächste Abschnitt.

4.1.6 Anbindung von externen Statistik-Werkzeugen

Bisher liefert *PARvis* Unterstützung bei der Visualisierung von Trace-Daten. Bei der Interpretation dieser Daten ist der Anwender allein auf sein Wissen und seine Erfahrung angewiesen. Ein unerfahrener Anwender wird also nicht denselben Nutzen aus der Performance-Analyse ziehen wie ein erfahrener Anwender. Ein Ziel für die weitere Entwicklung der Performance-Analyse mit Werkzeugen wie *PARvis* oder *OPAL* sollte die Unterstützung des Anwenders bei der Interpretation von Trace-Daten sein. Werkzeuge zur statistischen Analyse von Daten wie beispielsweise das Paket *Statistical Analysis System* (SAS) [CoSm85] können einen Teil dieser Unterstützung liefern.

Ein erster Ansatz zur Anwendung statistischer Methoden auf Trace-Daten wäre ihr Einsatz als Filter, um die Menge der auszuwertenden Trace-Daten einzuschränken. Das Ziel bei dieser Anwendung ist eine Vorselektion bestimmter, für die Performance-Analyse besonders interessanter Trace-Daten. Ein weiterer wichtiger Aspekt wird sein, mit Hilfe statistischer Methoden verschiedene Zeitspannen für gleiche Vorgänge herauszufiltern. Diese entstehen unter anderem durch unterschiedliche Auslastung des Verbindungsnetzwerkes. Es muß also möglich sein, bei einer gewissen „Unschärfe“ der Ausgangsdaten trotzdem noch die relevanten Informationen zu erkennen. Bei Trace-Daten von Message-Passing-Programmen können dies charakteristische Kommunikationsmuster für bestimmte Algorithmen sein, bei Trace-Daten von *Shared Virtual Memory*-Systemen charakteristische Bewegungen von Speicherseiten durch den Adreßraum des Parallelrechners. Für bestimmte Fehler in der Programmkonzeption werden sich ebenfalls charakteristische „Muster“ ergeben, die durch statistische Methoden analysierbar sind. Lösungsvorschläge zur Behebung der Fehler können beispielsweise in einer Wissensbasis gesammelt werden und dem Anwender über eine gemeinsame Schnittstelle Performance-Analyse-Werkzeug/Wissensbasis zugänglich gemacht werden. Der Aufwand für die Kombination eines Performance-Analyse-Werkzeuges mit einer Wissensbasis ist wahrscheinlich nicht gering, sie erleichtert aber auch unerfahrenen Anwendern eine richtige Interpretation von Trace-Daten und die sich daran anschließende Optimierung ihrer Programme.

4.2 Zusätzliche Funktionalität durch besondere Ereignisse

Die bisher von *PARvis* ausgewerteten Trace-Daten hielten die Vorgänge im Parallelrechner während der Programmausführung fest. Diese Informationen protokolliert *SAM* auch (*Read Page Faults, Write Page Faults* usw.[4]). Je nach Instrumentierung geschieht dies für das gesamte Programm oder nur für bestimmte, ausgewählte Bereiche. Die Erzeugung von Ereignissen ist also nicht von Schwellwerten abhängig (wie zum Beispiel bei *Pablo* [RAMN92]), sondern wird über die Instrumentierung durch den Quelltext gesteuert. Dadurch lassen sich Informationen gewinnen, die bisher nicht verfügbar waren. Die Verteilung von Variablen auf Speicherseiten wird damit darstellbar (siehe Abschnitt 5.5). Aus den die Verteilung von Speicherseiten beschreibenden Ereignissen läßt sich ein „Snapshot"-Ereignis bilden, also ein Aufsetzpunkt in der Trace-Datei.

4.2.1 Das „Snapshot"-Ereignis

Durch das Stützstellen-Konzept von *PARvis* ist es möglich, eine Trace-Datei entlang der Zeitachse vorwärts und rückwärts „durchzublättern" [Arn93]. Um eine bestimmte Stelle in der Trace-Datei darzustellen, müssen alle Ereignisse ab der letzten Stützstelle eingelesen werden. Bei Trace-Daten von *SVM-Fortran*-Programmen kann dieser Vorgang verkürzt werden. Ermöglicht wird dies durch die Auswertung mehrerer Ereignisse, die zusammengefaßt ein „Snapshot"-Ereignis bilden. Die Bildung der Stützstellen ist von dem Auswertungswerkzeug in die Trace-Datei verlegt worden. Da sich die Snapshot-Funktion auf eine Region bezieht, braucht man für die globale Betrachtung einer Trace-Datei weiterhin die Stützstellen von *PARvis*. Abbildung 4.6 verdeutlicht das Prinzip dieses Makro-Ereignisses.

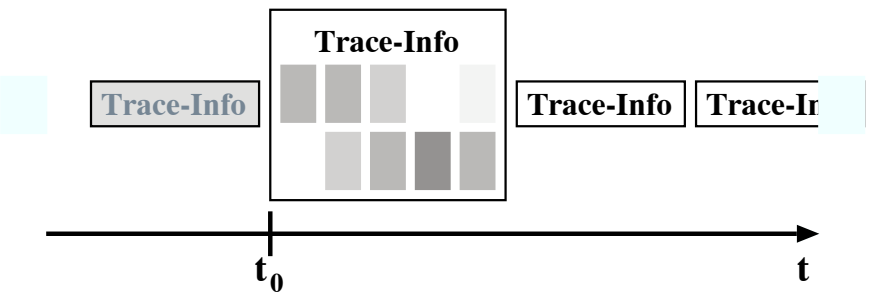


Abbildung 4.6: Darstellung des „Snapshot"-Ereignisses in *PARvis*

Zur korrekten Darstellung der Trace-Daten benötigt *PARvis* folgende Startinformationen:

[4] In [Kru95] sind sämtliche *SVM-Fortran*-Ereignisse verzeichnet.

- die Verteilung von Speicherseiten mit
  - Leseberechtigung,
  - Schreibberechtigung,
- die Regionen- und Symbolnamen und
- für den erweiterten Quelltextbezug (siehe Kapitel 5) die Verteilung von Variablen auf die Speicherseiten.

Diese Informationen werden durch die folgenden Ereignisse zur Verfügung gestellt:

```
#50~~read-page-dist-start
#52~~write-page-dist-start
#111 all-variable-mapping
#112 single-variable-mapping
```

Die Referenzierung der Regionen- und Symbolnamen erfolgt wieder durch Auswertung der *Compiler Information Data*. Wird das Ende der spezifizierten Region erreicht, wird auch die Darstellung der Trace-Daten durch *PARvis* beendet. Da die in der obigen Aufzählung angegebenen Ereignisse nur am Anfang einer instrumentierten Region generiert werden, ist die Betrachtung der Trace-Daten auch auf dieses „Fenster“ beschränkt. In dieser Region können die in Abschnitt 4.1.4 angeführten statistischen Auswertungen vom Anwender vorgenommen werden. Auf die in diesem Abschnitt beschriebenen globalen Zähler für verschiedene Ereignisse (beispielsweise *Read Page Faults*) muß bei Anwendung der Snapshot-Funktion in *PARvis* verzichtet werden.

4.2.2 Darstellung der Auslagerung von Speicherseiten

Wird der Speicherplatz während der Programmausführung auf einem Knoten knapp, kann er längere Zeit nicht mehr benötigte Speicherseiten auslagern. Je nach verwendetem Betriebssystem hat der implementierte Speichermanager verschiedene Möglichkeiten, auf diese Situation zu reagieren. Auf der Intel Paragon ist das Speicherkonzept des *Shared Virtual Memory* durch den aus Mach bekannten externen Memory Manager XMM realisiert, der in der ersten Version einer Neuimplementierung durch den ASVM-Kernel zur Verfügung gestellt wird. Die Verdrängungsstrategie dieses Speichermanagers sieht vor, daß eine auszulagernde Speicherseite nach Möglichkeit zuerst von einem Knoten aufgenommen wird, der noch über ausreichend freie Speicherkapazität verfügt. Dieses ist mit weitaus geringeren Kosten verbunden als die Nutzung des Auslagerungsbereiches des Dateisystems. Erst wenn eine Auslagerung zu einem anderen Knoten nicht mehr möglich ist, wird diese zweite Variante von dem Speichermanager ausgewählt.

Da der in *PARsim* integrierte SVM-Simulator davon ausgeht, daß ein Knoten stets über ausreichend Speicherplatz verfügt, ist die Darstellung der Verdrängung einer Speicherseite in *PARvis* bisher nicht notwendig gewesen. Zur Visualisierung der Paging-Informationen von *SVM-Fortran*-Programmen in *PARvis* ist im Rahmen dieser Arbeit folgende Darstellungsweise eingeführt worden:

- Wird eine Speicherseite zu einem anderen Knoten ausgelagert, wird für sie die diesem Knoten zugeordnete Farbe verwendet. Die Darstellung erfolgt jedoch schraffiert.
- Die Nutzung des Auslagerungsbereiches des Dateisystems wird durch eine Farbe gekennzeichnet, die für keinen Prozessor zur Kennzeichnung verwendet wird (beispielsweise weiß). Die Angabe einer Prozessornummer entfällt.

Auf diese Weise ist eine Darstellung „verdrängter“ Speicherseiten möglich. Speicherseiten verschwinden jetzt bei der Visualisierung der Trace-Daten von *SVM-Fortran*-Programmen nicht mehr, wenn sie ausgelagert werden.

Nachdem in diesem Kapitel die Visualisierung von Trace-Daten bereits um erste Elemente eines Quelltextbezuges ergänzt worden ist, wird im nächsten Kapitel dieser Bezug noch verstärkt. Durch die Entwicklung einer Schnittstelle von *PARvis* zu *OPAL* erhält der Benutzer Zugriff auf den Quelltext des analysierten Programms. Das Ziel der neuen Komponenten ist die Identifizierung von Daten auf Speicherseiten mit ihren Variablen.

Kapitel 5

Konzeption und Realisierung einer Schnittstelle zwischen PARvis und OPAL

In diesem Kapitel wird die Ergänzung von *PARvis* durch eine Schnittstelle zu dem Performance-Analyse-Werkzeug *OPAL* beschrieben. Dazu erfolgt einleitend eine kurze Erläuterung der Fähigkeiten von *OPAL*, bei der die Unterschiede in der Performance-Analyse zu *PARvis* herausgearbeitet werden. Die Möglichkeiten, die in einem Quelltextbezug bei der Analyse von Trace-Daten stecken, dienen als Motivation für diese Schnittstelle. Die hier beschriebenen Berührungspunkte in der Methodik der Performance-Analyse leiten dann über zu den Kriterien für eine Schnittstelle zwischen *PARvis* und *OPAL*. Den Abschluß dieses Kapitels bildet die Beschreibung der Realisierung dieser Schnittstelle, wobei jeweils die für die Zusammenarbeit der beiden Werkzeuge notwendigen Daten durch die Angabe der entsprechenden Ereignisse definiert werden.

5.1 OPAL – Optimierungs- und Lokalitätsanalyse-Werkzeug für SVM-Fortran

Das Performance-Analyse-Werkzeug *OPAL* bietet dem Benutzer die Möglichkeit der Untersuchung von *SVM-Fortran*-Programmen auf der Basis des Quelltextes [GKÖ95, BGM95]. Zu den Fähigkeiten von *OPAL* gehören unter anderem:

- Die Instrumentierung ausgewählter Bereiche des Programms.
- Die Interpretation des Programmtextes durch einen eingebauten Fortran-Parser.
- Die Analyse der generierten Trace-Daten.

Die Benutzeroberfläche wurde bereits in Abbildung 3.7 (Seite 30) gezeigt. In dem Hauptfenster wird der Quelltext angezeigt, das darunter liegende kleinere Fenster ist für Meldungen über die ausgeführten Aktionen von *OPAL* vorgesehen.

Während des Einlesens eines Quelltextes wird dieser von dem Fortran-Parser untersucht. Es erfolgt eine Unterteilung des Quelltextes in Module, sogenannte *Units*. Eine *Unit* ist ein Fortran-Unterprogramm (`SUBROUTINE`) oder eine Fortran-Funktion (`FUNCTION`). In dem Hauptfenster von *OPAL* wird die von dem Benutzer ausgewählte *Unit* angezeigt. Es gibt keine hierarchische Beziehung zwischen den einzelnen *Units*, sie sind alle gleichberechtigt. Durch die Interpretation des Programmtextes durch den Parser besitzt *OPAL* die Fähigkeit,

- die Symboltabelle und den Syntax-Baum des Programms aufzubauen und
- eine Konsistenzkontrolle der (Feld-)Variablen und `COMMON BLOCKS` hinsichtlich ihrer *SVM-Fortran*-Speicherklasse (`PRIVATE`, `SHARED` oder `PSHARED`) durchzuführen.

Wie bereits in Kapitel 3 angesprochen, kommt der Instrumentierung des Quelltextes entscheidende Bedeutung für die Ergebnisse der Performance-Analyse zu. Ist die Instrumentierung zu „grob“, werden zu viele Trace-Daten erzeugt, der Blick auf die wesentlichen Informationen ist verstellt. Aus diesem Grund bietet *OPAL* nicht nur die Möglichkeit der kompletten Instrumentierung einiger oder aller *Units* an, sondern auch die gezielte Auswahl bestimmter Bereiche des Programms. Zu diesen Bereichen gehören unter anderem die bereits in Abschnitt 4.1.3 erwähnten *SVM-Fortran*-Regionen (zum Beispiel parallele Schleifen (`PDO`) oder parallele Regionen (`REPLICATED_REGION`))[1]. Weiterhin sind auch Feldvariablen innerhalb einer parallelen Schleife oder einer *Unit* instrumentierbar. Eine genaue Steuerung der Erzeugung von Trace-Daten ist also möglich.

Bei der Instrumentierung für *SAM* durch *OPAL* wird der Quelltext nicht direkt modifiziert. Es werden lediglich sogenannte „Hook“-Funktionsaufrufe vor und nach jeder *SVM-Fortran*-Region in den Quelltext eingefügt. Diese Funktionsaufrufe stellen Einsprungpunkte in die *SAM* Trace-Bibliothek dar. Die in der *SAM*-Bibliothek schließlich auszuführenden Aktionen sind in der Steuerungsdatei (Trace Request-Datei) `SAM_TRF.rqf`, die von *SAM* während der Programmausführung ausgewertet wird, festgelegt. Durch die textuelle Beschreibung der instrumentierten Bereiche in der Steuerungsdatei ist dieses Monitoring-Verfahren sehr flexibel. So ist es möglich, ein einmal mit der *SAM*-Bibliothek gebundenes Programm durch Veränderung der Trace Request-Datei verschieden zu instrumentieren, ohne daß das gesamte Programm neu übersetzt werden muß [Özm95].

Die Trace Request-Datei wird von *OPAL* erstellt, sie kann aber auch von Hand erstellt oder modifiziert werden. Eine präzisere Instrumentierung als durch *OPAL*

[1] Eine Beschreibung der Programmiersprache *SVM-Fortran* findet sich in [BeGe95].

ist durch eine direkte Modifikation der Trace Request-Datei `SAM_TRF.rqf` möglich. Die Grammatik der „Instrumentierungsbefehle“ in dieser Steuerungsdatei ist in [Özm95] beschrieben. Für die Arbeit mit den Werkzeugen *PARvis* und *OPAL* ist eine Instrumentierung von Hand nicht notwendig, da sie sowohl für *OPAL* als auch für *PARvis* durch *OPAL* gesteuert wird. Die Erstellung dieser Steuerungsdatei für *PARvis* erfolgt durch Auswahl des Menüpunktes **Instrument**: *Instrument for PARvis* in *OPAL*.

Nachdem die erzeugten Trace-Daten von *OPAL* eingelesen worden sind, ist der Abruf der Trace-Informationen zu den instrumentierten Bereichen möglich. Dem Benutzer werden diese Informationen in Form von Statistiken präsentiert, eine Visualisierung findet in *OPAL* nicht statt. Wie in Abbildung 5.1 zu erkennen ist, zeigt *OPAL* dem Benutzer die Zahl der in der Region aufgetretenen *Read-* und *Write Page Faults* sowie ihre Ausführungszeit zusammen mit dem Quelltext.

```
12: CSVM$ REPLICATED_REGION                                    R 99( 55%)  W  8(  1%) 1.337
13: CSVM$   PDO(LOOPS(NC),STRATEGY(ON_HOME(NODES(NC))),NOBARRIER)   R  0(  4%)  W  0(  8%) 0.015
14: CSVM$   PDO(LOOPS(NC),STRATEGY(ON_HOME(NODES(NC))),NOBARRIER)   R 24( 71%)  W  0(  0%) 0.255
15:         DO 4 NC=NINTCI,NINTCF
16:           DIREC2(NC) = BP(NC) * DIREC1(NC) - BS(NC) * DIREC1(LCC(1,NC))
        *     - BW(NC) * DIREC1(LCC(4,NC)) - BL(NC) * DIREC1(LCC(5,NC)) -
        *     BN(NC) * DIREC1(LCC(3,NC)) - BE(NC) * DIREC1(LCC(2,NC)) - BH(
        *     NC) * DIREC1(LCC(6,NC))
17: 4       CONTINUE
```

Abbildung 5.1: Quelltext-Anzeige in *OPAL*

Über ein Untermenü (siehe Abbildung 3.7, Seite 30) sind weitere Funktionen abrufbar, die diese Informationen genauer spezifizieren. Abbildung 5.2 zeigt eine Auswahl aus diesen Funktionen. So läßt sich beispielsweise eine Übersicht mit den aus den Trace-Daten ermittelten Werten für die ausgewählte Region (Fenster **summary**) aktivieren.

Da sich die Präsentation der Trace-Daten auf den Quelltext bezieht, gibt es im Gegensatz zu *PARvis* keine Zeitlinie *(Timeline)*. Wird ein instrumentierter Bereich während der Programmausführung mehrfach durchlaufen, ist im Quelltext der Übersichtlichkeit halber nur die Anzeige eines Wertes für alle Inkarnationen dieses Bereiches vorgesehen. Für die Auswertung der Summe von Seitenfehlern bedeutet dies beispielsweise, daß sämtliche Summen-Ereignisse für den betroffenen Bereich aufaddiert werden. Anschließend wird der Mittelwert der einzelnen Summanden gebildet. Ist der Anwender an den Mittelwerten der bei den einzelnen Knoten aufgetretenen Seitenfehler interessiert, kann er dazu ein Fenster (**MAIN:loop**) öffnen, welches ihm die gewünschten Werte anzeigt.

Möchte der Benutzer wissen, welchen Wert nun die einzelnen Summanden tatsächlich hatten, so wird die Trace-Datei nach den „passenden“ Summen-Ereignissen durchsucht. Diese werden dann dem Benutzer in einem Textfenster in chronologischer Reihenfolge angezeigt. In Abbildung 5.2 ist dies das Fenster **SAM_READ_PAGE_FAULT_SUM**. Wie der Name des Fensters bereits andeutet, wurden hier Ereignisse vom Typ `read-page-fault-sum` ausgewertet.

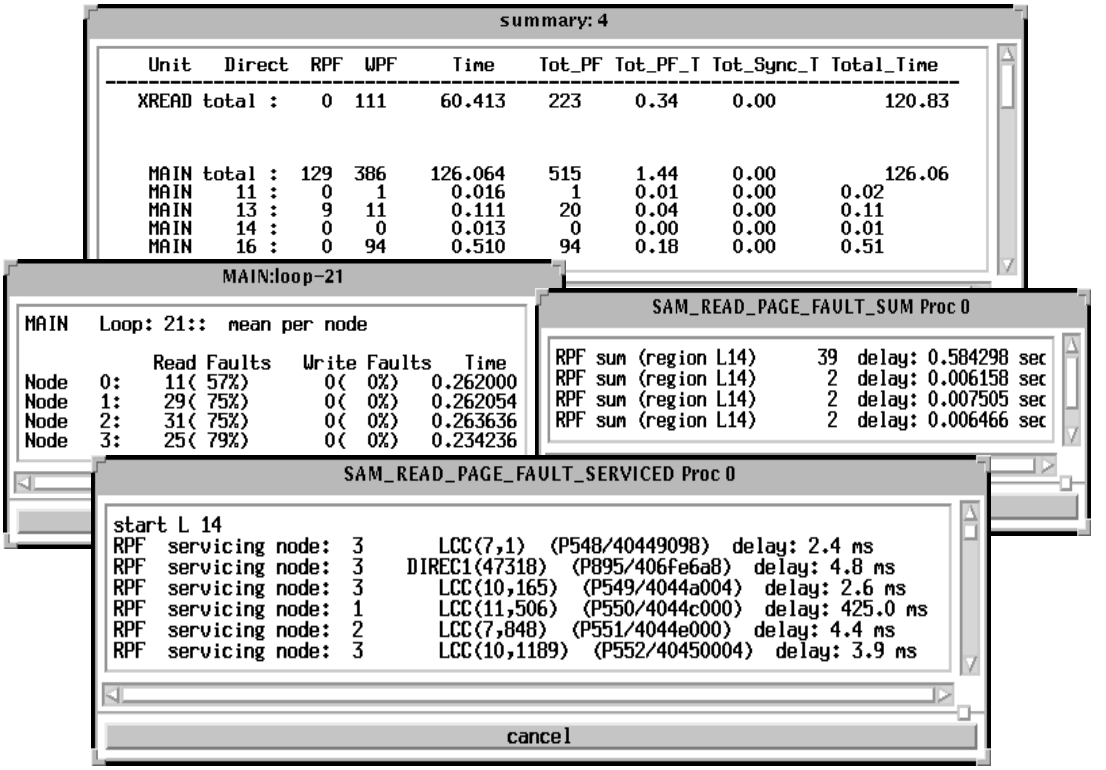


Abbildung 5.2: Quelltext-Anzeige in *OPAL*

Das letzte noch nicht beschriebene Fenster in Abbildung 5.2 (**SAM_READ_PAGE_FAULT_SERVICED**) ist ein aufbereiteter Ausschnitt aus der Trace-Datei. Als Vergleich dazu dient die entsprechende Darstellung in *PARvis*, Abbildung 4.3 auf Seite 54.

Die bereits weiter oben angesprochene Instrumentierung von Feldvariablen dient der Lokalisierung dieser Variablen auf einzelnen Speicherseiten. Während des Einlesens der Trace-Datei benutzt *OPAL* dazu das `all-variable-mapping`-Ereignis, um die globalen Variablen der Speicherklasse `SHARED` den einzelnen Speicherseiten zuzuordnen. Da auch hier keine Darstellung über die Zeit erfolgen kann, bekommt der Benutzer die Variablenverteilung angezeigt, wie sie zum Schluß der Programmausführung war. Man erwartet dabei, daß sich die Verteilung der einzelnen Variablen auf die Speicherseiten während der Programmausführung nicht mehr allzusehr verändert.

Neben den beschriebenen statistischen Angaben über Seitenfehler sind weitere Angaben über die Programmausführung abrufbar, wie zum Beispiel die Zeiten für den *Tracing Overhead* oder den *SVM-Fortran Overhead*.

Nachdem nun die Funktionsweise von *OPAL* bekannt ist, erfolgt im nächsten Abschnitt eine Gegenüberstellung der Fähigkeiten und Konzepte bei der Performance-Analyse von *PARvis* und *OPAL*. Anhand dieser Gegenüberstellung werden dann die Kriterien für eine gemeinsame Schnittstelle hergeleitet.

5.2 Gegenüberstellung der Fähigkeiten und Konzepte

Um die beiden Werkzeuge *PARvis* und *OPAL* zu vergleichen, kann man dazu drei Kategorien bilden, an denen die Unterschiede und Gemeinsamkeiten deutlich werden:

- Präsentation der Trace-Daten
- Quelltext-Bezug
- Auswahl der Trace-Informationen

Zunächst zu der *Präsentation* der Trace-Daten. In *PARvis* erfolgt sie symbolisch, das heißt, daß die Trace-Daten eine graphische Umsetzung zur Visualisierung erfahren. Die Zeit wird für die Darstellung nicht herausgefiltert, da die Trace-Daten chronologisch über die Zeit präsentiert werden. Um sich einen Eindruck von der Dynamik der Vorgänge während der Programmausführung zu verschaffen, kann der Benutzer eine Animation der Trace-Daten verfolgen. *OPAL* hingegegen bietet eine numerische Repräsentation der Trace-Daten, die sich am Quelltext ausrichtet. Die Zeit wird bei der Summation der Ergebnisse für gleiche Regionen herausgefiltert (siehe Abschnitt 5.1). Die Präsentation der Trace-Daten ist also, im Gegensatz zu *PARvis*, statisch. Eine graphische Umsetzung der gewonnenen Zahlen erfolgt nicht.

Der *Quelltext* ist der grundlegende Bezugspunkt für alle Aktionen in *OPAL*. An ihm richtet sich die Präsentation der Informationen zur Performance aus. Der Benutzer wählt die Regionen oder Variablen aus, deren zugehörige Trace-Informationen er angezeigt bekommen möchte. *OPAL* stellt diese Informationen in numerischer Form dar. Da der Quelltext nicht einfach nur als ASCII-Text in einem Fenster angezeigt, sondern durch den Parser syntaktisch interpretiert wird, können zusätzliche Informationen aus dem Quelltext gewonnen werden (siehe Abschnitt 5.1). In *PARvis* beschränkt sich der Quelltext-Bezug auf die Anzeige der Symbolnamen und im Fall von Trace-Daten von *SVM-Fortran*-Programmen auf die zusätzliche Anzeige der zu einer Region gehörenden Direktiven-Namen. Durch den Zeitbezug ist jedoch jede einzelne Inkarnation einer Region sichtbar. Bei beiden Werkzeugen kann der Benutzer nachträglich nur durch Auswertung der Trace Request-Datei Kenntnis darüber erlangen, wie die Instrumentierung erfolgte.

Durch die gezielte *Auswahl* der zu instrumentierenden Bereiche innerhalb des Quelltextes legt der Benutzer in *OPAL* fest, welche Trace-Informationen er erhalten möchte. Das realisierte Instrumentierungskonzept gestattet es ihm dabei, die Granularität der gesammelten Informationen in einem sehr weiten Bereich zu steuern. *PARvis* beschreitet hier einen anderen Weg. Der Benutzer kann durch Filter-Optionen bestimmte Ereignisse in der Trace-Datei ausblenden. Unwichtige Informationen können so verdeckt werden, die interessierenden treten in den Vordergrund. Während in *OPAL* die Gewinnung der Trace-Informationen gesteuert wird, ist in

PARvis die nachträgliche Beeinflussung der Auswertung der Trace-Informationen möglich.

Neben diesen drei Kategorien ist auch ein Vergleich von *PARvis* und *OPAL* auf der Ebene der Trace-Daten möglich. Als Unterscheidungsmerkmal bietet sich hier die jeweilige Verwendung der in Abschnitt 3.6 beschriebenen Ereignis-Gruppen an.

Die Gruppe der *Kernel Event*-Ereignisse stellt die grundlegenden Informationen über Veränderungen im Zustand von Speicherseiten zur Verfügung. Sie wird daher von beiden Werkzeugen gleichermaßen genutzt. Die Ereignisse, die in der *Kernel Information*-Gruppe zusammengefaßt sind, werten die *Kernel Event*-Ereignisse statistisch aus. Daher dienen die in diesen Ereignissen vorhandenen Daten auch bei beiden Werkzeugen als Ausgangsbasis für die statistischen Auswertungen der Trace-Daten. Bisher verwendet *PARvis* noch keine Ereignistypen aus der Gruppe der *Language Information*-Ereignisse. *OPAL* benutzt einige dieser Ereignisse, um so beispielsweise den Trace-Overhead oder die Zeiten für Synchronisationsvorgänge der Prozessoren untereinander auszuwerten. Wie der Name dieser Ereignisgruppe bereits andeutet, werden hier im Zusammenhang mit der Programmiersprache *SVM-Fortran* stehende Ereignisse protokolliert. Durch die Anzeige des Quelltextes in *OPAL* fällt es leichter, die in diesen Ereignissen enthaltenen Informationen dem Anwender zugänglich zu machen.

5.3 Kriterien für die Schnittstelle zwischen PARvis und OPAL

Eine Schnittstelle zwischen den beiden Werkzeugen *PARvis* und *OPAL* gibt dem Benutzer bei der Performance-Analyse neue, wichtige Möglichkeiten an die Hand. Bei der Auswertung der Trace-Daten verfolgen beide Werkzeuge unterschiedliche Konzepte, die sich jedoch gerade wegen ihrer Unterschiedlichkeit gewinnbringend für den Anwender kombinieren lassen. Dabei sollen beide Systeme weiterhin als unabhängige Programme verwendbar bleiben.

Die zu entwickelnde Schnittstelle muß für den Benutzer einfach zu bedienen sein. Er darf nicht durch eine komplizierte Handhabung von ihrer Anwendung abgeschreckt werden. Weiterhin muß sichergestellt sein, daß der Anwender die von ihm gewünschten Informationen mit möglichst geringem Aufwand erhält. Dazu muß die Schnittstelle effizient hinsichtlich

- der Bedienbarkeit und
- der Programmierung (geringe Performance-Einbußen bei der Werkzeugnutzung)

gestaltet sein. Im Endeffekt soll als Symbiose aus den beiden Werkzeugen eine integrierte Entwicklungsumgebung entstehen, in der von einer Komponente des einen

Werkzeuges eine Komponente des anderen Werkzeuges aufgerufen werden kann. Dabei wird *PARvis* zur Visualisierung der Trace-Daten eingesetzt, während *OPAL* den notwendigen Quelltextbezug liefert. So sind im einzelnen folgende Funktionen entwickelt worden:

- *Einschränkung der Visualisierung auf bestimmte Programmsymbole.*
  Diese Filterfunktion wird von *OPAL* gesteuert, indem der Anwender dort bestimmte Programmsymbole auswählt. Die mit diesen Symbolen im Zusammenhang stehenden Trace-Daten werden dann in *PARvis* visualisiert.

- *Anzeige, wo im Programm ein bestimmtes Ereignis generiert wurde.*
  Hier markiert der Anwender ein ihn besonders interessierendes Ereignis in *PARvis*. Dies könnte beispielsweise der Beginn einer Region mit besonders vielen Seitenfehlern sein. In *OPAL* wird die *Unit* geladen, die diese Region enthält, und der dazugehörende Quelltext wird dem Anwender angezeigt.

- *Sprung an bestimmte Punkte in der Trace-Datei.*
  Die Auswahl eines instrumentierten Bereiches im Quelltext geschieht in *OPAL*. Je nach durchgeführter Instrumentierung kann dies zum Beispiel eine parallele Region sein. In *PARvis* wird dann die Anzeige auf den entsprechenden Bereich in der Trace-Datei ausgerichtet.

- *Identifizierung von Daten auf Speicherseiten mit ihren Variablennamen.*

- *Lokalisierung von Variablen auf Speicherseiten.*
  Diese beiden Punkte sind in ihrer Funktionalität jeweils die Umkehrung des anderen. In *PARvis* wählt der Anwender dazu eine Speicherseite aus, woraufhin ihm in *OPAL* die auf dieser Speicherseite vorhandenen Variablen angezeigt werden. Geht der Anwender den umgekehrten Weg, wählt er zunächst in *OPAL* eine Variable aus. *PARvis* hält dann die notwendigen Informationen für ihn bereit, damit er erkennen kann, auf welchen Speicherseiten sich diese Variable befindet.

In Abbildung 5.3 sind diese Punkte noch einmal zusammenfassend dargestellt.

Nachdem nun die Kriterien für eine Schnittstelle definiert worden sind, erfolgt im nächsten Abschnitt die Beschreibung der daraus entwickelten Schnittstelle zwischen *PARvis* und *OPAL*.

5.4 Beschreibung der Realisierung der Schnittstelle zwischen PARvis und OPAL

Die Schnittstelle zwischen *PARvis* und *OPAL* läßt sich realisieren, da beide Werkzeuge auf eine gemeinsame Datenbasis zugreifen, nämlich die Trace-Daten eines

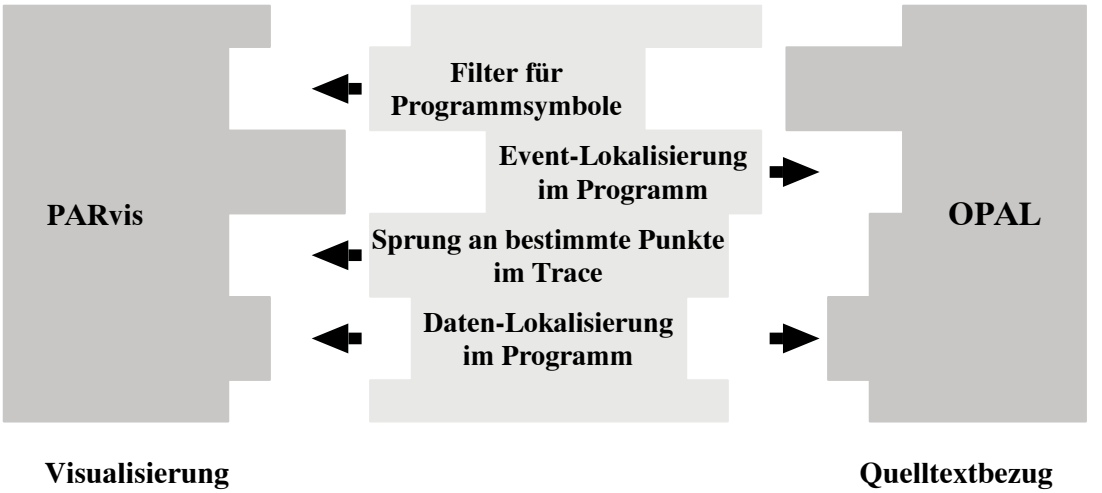


Abbildung 5.3: Konzept einer Schnittstelle zwischen *PARvis* und *OPAL*

SVM-Fortran-Programms. Auch wenn in *OPAL* bei der Darstellung der Trace-Informationen die Zeit herausgefiltert wird, ist sie dennoch in den einzelnen Ereignissen weiterhin auch für *OPAL* sichtbar. Ein einzelnes Ereignis ist über seinen Zeitstempel und die Nummer des Knotens, von dem es erzeugt wurde, eindeutig zu identifizieren.

Will der Anwender von einem Werkzeug zusätzliche Informationen von einer Komponente des anderen Werkzeuges über die Schnittstelle erhalten, müssen die folgenden Informationen zwischen den beiden Werkzeugen ausgetauscht werden:

- Eine Identifikation der Komponente, die aktiviert werden soll, und
- die sich aus dem Zeitstempel und der Knotennummer des Ereignisses ergebende physikalische Position in der Trace-Datei.

Um eine eigenständige Nutzung der beiden Werkzeuge zu erhalten, bietet sich als einfacher Kommunikationsmechanismus zwischen den beiden Programmen die Verwendung einer nach dem First In First Out-Prinzip *(FIFO)* organisierten Warteschlange an. Solch eine Warteschlange wird von dem Betriebssystem `UNIX` als sogenannte *FIFO-Pipe* zur Verfügung gestellt. Diese wird von einem neuen Modul des Werkzeuges in bestimmten Zeitabständen abgefragt. Ist eine neue Nachricht vorhanden, wird sie ausgewertet und die entsprechenden Aktionen ausgeführt. Diese Vorgehensweise entspricht einer Ereignissteuerung der Programme.

Sind zusätzlich Daten (beispielsweise einzelne Ereignisse) zwischen den beiden Performance-Analyse-Werkzeugen auszutauschen, ist dieses am effizientesten über ein gemeinsames Segment *(Shared Segment)* innerhalb des Arbeitsspeichers möglich. Diese gemeinsamen Segmente stehen dem Programmierer seit dem Release 3.2 des `UNIX` System V zum Datenaustausch zwischen einzelnen Prozessen zur Verfügung.

Sollen die beiden Werkzeuge *PARvis* und *OPAL* auf zwei verschiedenen Rechnern ausgeführt werden, kann mit Hilfe der *Interclient Communication* eine Verbindung zwischen den beiden Applikationen und ihren Host-Rechnern aufgebaut werden. Den Mechanismus dazu kann das X11-Protokoll durch seine Client-Server-Architektur zur Verfügung stellen. Falls die Rechenleistung des Auswertungsrechners zum gleichzeitigen Betrieb der beiden Werkzeuge nicht ausreichen sollte, kann auf eine solche „verteilte" Performance-Analyse ausgewichen werden. Dieser Kopplungsmechanismus ist allerdings wegen seiner Komplexität im Rahmen dieser Arbeit nicht implementiert worden.

Entsprechend der in Abschnitt 5.3 aufgeführten Kriterien ergeben sich die folgenden Aktionen, die über die Schnittstelle bei dem jeweiligen Werkzeug ausgelöst werden. Zunächst erfolgt eine Beschreibung der von *PARvis* bei *OPAL* auslösbaren Aktionen.

- *Anzeige, wo im Programm ein bestimmtes Ereignis generiert wurde.*
  PARvis übermittelt dazu den Zeitstempel, die Knotennummer sowie die aus dem gewählten Ereignis generierbaren Sprachinformationen. Dazu zählen, je nach Ereignis, die Unterprogrammnummer (`Subprogram No`), die Nummer der Region (`Region No`) oder die Variablennummer (`Variable No`). Gegebenenfalls werden diese Informationen über den Kontext, in dem sich das ausgewählte Ereignis befindet, ermittelt. Durch die Versendung des Zeitstempels ist eine eindeutige Identifikation der Inkarnation einer Programmregion möglich. Diese Region wird dann im Quelltext markiert und dem Anwender angezeigt.

- *Identifizierung von Daten auf Speicherseiten mit ihren Variablennamen.*
  Hierzu wird in *PARvis* der Adreßbereich der ausgewählten Speicherseite ermittelt und in dem gemeinsamen Speichersegment abgelegt. An *OPAL* wird, wie bei allen Aktionen, der Zeitstempel und die Knotennummer übergeben. Dies ist notwendig, da die Verteilung von Variablen auf Speicherseiten während der Programmausführung nicht konstant ist. Aus diesen Angaben bestimmt *OPAL* die aktuelle Belegung der Speicherseite mit Variablen. Durch eine Markierung dieser Variablen im Quelltext erhält der Benutzer die gewünschten Informationen.

Durch die Präsentation der Ereignisse über die Zeit bietet *PARvis* dem Anwender mehr Möglichkeiten zur Verfolgung des Programmablaufes als *OPAL*. Somit ergeben sich drei Aktionen, bei denen *PARvis* *OPAL* in der Visualisierung unterstützen kann (siehe auch Abschnitt 5.3):

- *Einschränkung der Visualisierung auf bestimmte Programmsymbole.*
  Durch Verwendung der Filterfunktion in *PARvis* war dieses Kriterium der Schnittstelle sehr leicht zu implementieren. *OPAL* legt die im Quelltext markierten Programmsymbole in dem gemeinsamen Speichersegment ab, woraufhin *PARvis* damit seine Filterfunktion aktiviert. Der zuletzt in *PARvis* visualisierte Ausschnitt aus der Trace-Datei wird beibehalten, da an *PARvis* nur

Dummy-Argumente für den Zeitstempel und die Knotennummer übermittelt werden.

- *Sprung an bestimmte Punkte in der Trace-Datei.*
  Zur Bestimmung der in *OPAL* ausgewählten Programmregion erhält *PARvis* die entsprechenden Unterprogramm- und Regionennummern mitgeteilt. Um bei mehrfach ausgeführten Regionen den richtigen (Zeit-)Punkt in der Trace-Datei lokalisieren zu können, werden die dazu benötigten Informationen aus den Sprachereignissen (`pdo-iteration-space` usw. [Kru95]) gewonnen. Da diese Ereignisse bisher noch nicht durch den *SAM* erzeugt werden, wird zunächst immer der Zeitpunkt der ersten Inkarnation einer Region bestimmt und an *PARvis* übermittelt. Hier wird das enstprechende Ereignis in der Trace-Datei lokalisiert und angezeigt.

- *Lokalisierung von Variablen auf Speicherseiten.*
  Aus dem vom Benutzer ausgewählten Variablennamen wird in *OPAL* die entsprechende Variablennummer bestimmt. Aus dem Kontext muß nun noch die Region ermittelt werden, in welcher die Variable vom Benutzer ausgewählt wurde. Diese Informationen sind für *PARvis* jedoch zu einer eindeutigen Lokalisierung noch nicht ausreichend, da die zu dieser Variable gehörenden Daten beispielsweise in einer Schleife noch auf eine andere Speicherseite (einen anderen Prozessor) migrieren können. An *PARvis* wird daher der Zeitpunkt übermittelt, der zu dem Beginn der Region gehört. *PARvis* markiert dann in der Seitenübersicht *(Page Overview)* die Speicherseiten, die Daten zu der Variable enthalten. Dies geschieht schrittweise für jede Inkarnation der Region, so daß der Anwender ein eventuelles Migrieren der Daten durch den Speicher verfolgen kann.

Gerade die Identifizierung und Lokalisierung von Variablen auf Speicherseiten kann dem Anwender bei der Performance-Analyse wichtige Hinweise auf Fehler in seinem Programm liefern. Dazu gehört unter anderem die Visualisierung einer fehlerhaften Anordnung von Daten auf Speicherseiten, welche zu erheblichen Performance-Einbußen führt.

5.5 Die Lokalisierung von Daten im Speicher und im Programm

Neben dem in Abschnitt 4.1.5 angesprochenen Problem des Seitenflatterns *(Page Thrashing)* gibt es durch die Einteilung des Speichers in Seiten noch ein weiteres Problem, welches die Performance eines Programms sehr negativ beeinflussen kann. Es wird als *False Sharing* bezeichnet. Dieses Problem wird durch die von dem ASVM-Kernel der Intel Paragon verwendete Größe einer Speicherseite von 8 KByte

begünstigt. Auf Grund dieser Seitengröße können auf einer Speicherseite viele verschiedene Variablen der Speicherklasse `SHARED` vorhanden sein. Greift ein Prozessor schreibend auf eine dieser Variablen zu, müssen wegen der verwendeten starken Kohärenz sämtliche anderen Kopien dieser Speicherseite invalidiert werden. Möchte nun nach diesem Schreibzugriff ein anderer Prozessor lesend auf eine der nicht veränderten Variablen dieser Speicherseite zugreifen, bekommt er trotzdem eine neue Kopie dieser Speicherseite zugeschickt. Dabei ist die Versendung einer neuen Kopie der Speicherseite an den Prozessor eigentlich unnötig [GKÖ95].

Im Gegensatz zum *False Sharing* spricht man auch noch vom *True Sharing*. Hier möchte der andere Prozessor ebenfalls auf die Variable mit neuem Inhalt zugreifen. Das Versenden der Speicherseite ist also notwendig, damit der Prozessor auf eine aktuelle Kopie der Variable mit der Speicherklasse `SHARED` zugreifen kann [GKÖ95].

Die in der Schnittstelle zwischen *PARvis* und *OPAL* implementierten Aktionen – *Lokalisierung von Variablen auf Speicherseiten* sowie *Identifizierung von Daten auf Speicherseiten mit ihren Variablennamen* – ermöglichen es dem Benutzer, das Problem des *False Sharing* schnell und eindeutig zu erkennen. Dazu wählt er in *PARvis* in der Seitenübersicht *(Page Overview)* eine Speicherseite aus, von der er vorher in einem Statistik-Fenster gesehen hat, daß bei ihr unverhältnismäßig viele Seitenfehler aufgetreten sind. In *OPAL* werden ihm die zu dem in *PARvis* betrachteten Zeitpunkt vorhandenen Variablen auf der Speicherseite im Quelltext angezeigt. In Abbildung 5.4 ist dieser Vorgang dargestellt.

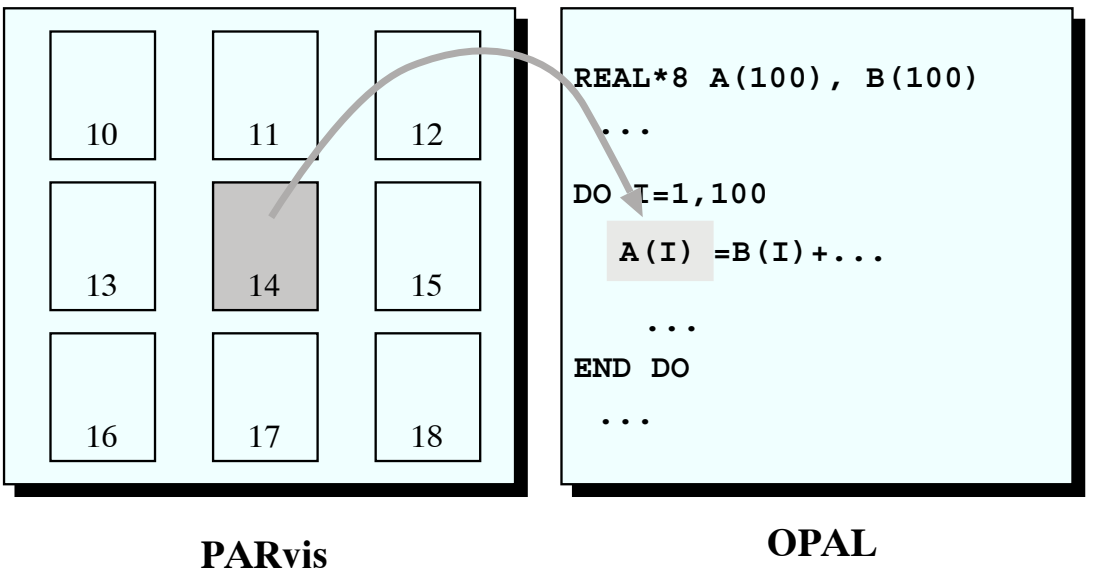


Abbildung 5.4: Identifizierung von Daten auf Speicherseiten mit ihren Variablennamen

Möchte der Anwender feststellen, an welche Stellen im Speicher die zu einer Variablen gehörenden Daten während der Programmausführung migriert sind, kann er

in *OPAL* im Quelltext die entsprechende Variable der Speicherklasse `SHARED` markieren. In *PARvis* erfolgt gegebenenfalls eine Animation der Migration der Daten durch den Speicher (siehe Abschnitt 5.4). Abbildung 5.5 zeigt diesen Vorgang.

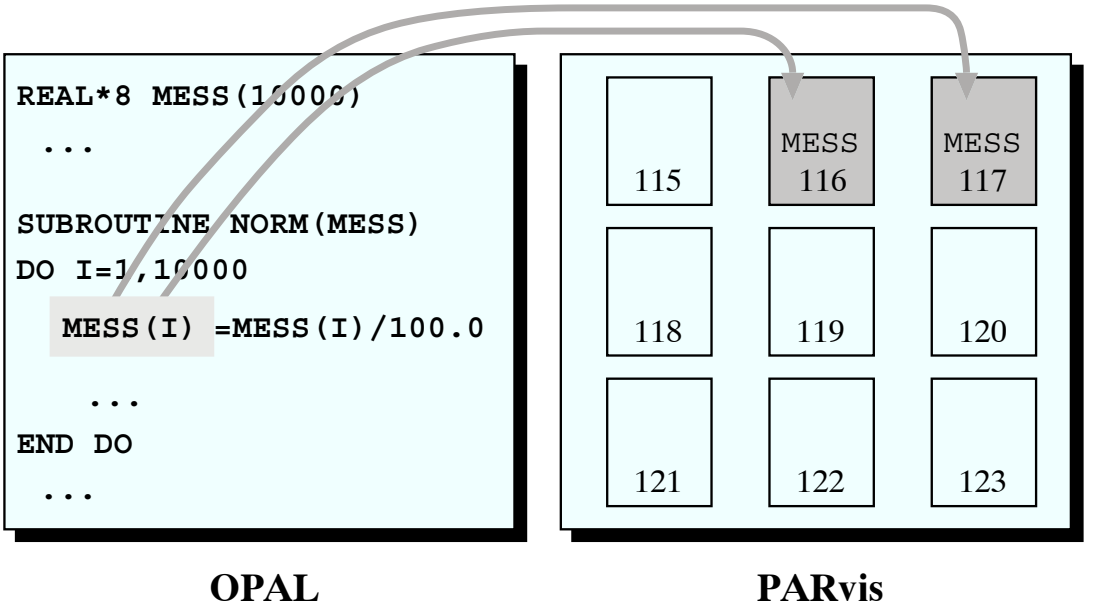


Abbildung 5.5: Lokalisierung von Variablen auf Speicherseiten

Mit den im Rahmen dieser Arbeit entwickelten Komponenten kann der Benutzer in einer integrierten Umgebung die Performance-Analyse eines *SVM-Fortran*-Programms durchführen. Damit ist die Arbeit im Vergleich zu den gewohnten Entwicklungsumgebungen für sequentielle Programme recht ähnlich. Die integrierte Umgebung stellt dem Anwender dazu die Visualisierung des Programmablaufes, die Lokalisierung und Identifizierung von Variablen auf Speicherseiten und eine Zuordnung zwischen Trace-Datei und einer bestimmten Stelle im Quelltext zur Verfügung.

Kapitel 6

Zusammenfassung und Ausblick

Im ersten Teil dieser Arbeit wurden Methoden zur Performance-Vorhersage beschrieben, deren Ziel es ist, unter Berücksichtigung spezifischer Architekturmerkmale des Parallelrechners eine möglichst genaue Vorhersage über den Programmablauf zu geben. Schwerpunkt beim Einsatz dieser Methoden sind Programmiersprachen, die auf dem Programmiermodell des *Compiler-Bridged Shared Memory* aufbauen. Dazu zählen unter anderem *Vienna Fortran* und *Fortran D*. Entscheidend für die Performance eines Programms ist hier die Vorgabe einer Datenverteilung, bei deren Auswahl ein Werkzeug zur Performance-Vorhersage dem Compiler verläßliche Hinweise liefern soll. Bei den beiden eben genannten Programmiersprachen werden solche Werkzeuge als Prototypen entwickelt und eingesetzt. Es hat sich herausgestellt, daß die Vorhersagen zu brauchbaren Ergebnissen führen, wenn die untersuchten Programme eine sehr einfache Struktur aufwiesen. Insbesondere irreguläre Probleme entziehen sich jedoch derzeit (unter Umständen sogar dauerhaft) einer Handhabung durch die beschriebenen Methoden. Die Performance-Vorhersage wird sicherlich ein wichtiger Baustein zukünftiger autoparallelisierender Compiler werden. Heute jedoch stellt sie keine Alternative zur Post Mortem-Analyse, vor allen Dingen der Performance-Analyse durch Ereignis-Protokollierung, dar.

Das Performance-Analyse-Werkzeug *PARvis* visualisiert Trace-Daten, die durch die Instrumentierung von Programmen erzeugt wurden. Die Instrumentierung kann entweder auf Quelltextebene oder Binär-Code-Ebene erfolgen. Für die Programmiersprache *SVM-Fortran* steht das Werkzeug *OPAL* zur Verfügung, über das die Instrumentierung des Quelltextes sehr feingranular vorgenommen werden kann. Der *SVM-Fortran Application Monitor (SAM)* wertet die Instrumentierung zur Laufzeit des Programms aus und sorgt für die Protokollierung der Ereignisse. Die abgespeicherten Trace-Daten liegen im SDDF-Format vor, welches sich durch seine Maschinenunabhängigkeit und die große Flexibilität bei der Definition von Ereignissen auszeichnet. Für die Manipulation der Trace-Daten im SDDF-Format stand, nach erheblichen Anpassungsschwierigkeiten, eine C++-Klassenbibliothek zur Verfügung.

Die in *PARvis* vorhandenen Komponenten zur Visualisierung von SVM-Aktivitäten konnten für Trace-Daten von *SVM-Fortran*-Programmen bisher nicht genutzt wer-

den, da für eine Konvertierung in das *PARvis*-Format kein Werkzeug vorhanden war. Im Rahmen dieser Arbeit wurde daher ein effizienter und stabiler Treiber für *PARvis* realisiert, mit dessen Hilfe jetzt auch die Trace-Daten von *SVM-Fortran*-Programmen in *PARvis* einer Performance-Analyse unterzogen werden können.

PARvis erwartet Trace-Daten in chronologischer Reihenfolge. Da bei *Read Page Faults* und *Write Page Faults* der Zeitpunkt des Auftretens eines Seitenfehlers und der Befriedigung der Seitenanforderung in einem Ereignis kombiniert ist, nimmt der Treiber eine Umformung dieser Ereignisse vor. Dazu wird innerhalb des Treibers eine chronologisch sortierte Liste verwaltet, in der sämtliche Einzelereignisse in zeitlich aufsteigender Reihenfolge einsortiert werden. Auf die Problematik und deren Lösung des zeitlichen Aufwandes für das Mischen und Sortieren der generierten p Trace-Dateien in eine einzelne Trace-Datei wurde hingewiesen.

Neben diesen sehr wichtigen Arbeiten konnte die bereits vorhandene Funktionalität der SVM-Komponenten auch der Analyse der Trace-Daten von *SVM-Fortran*-Programmen zugänglich gemacht werden. Im Bereich der statistischen Auswertungen von *PARvis* wurde die Darstellung der Summen von *Read Page Faults* und *Write Page Faults* für eine ausgewählte Inkarnation einer *SVM-Fortran*-Region ermöglicht. Treten in dieser Region überdurchschnittlich große Summen für Seitenfehler auf, können durch eine neue Komponente die zu dieser Region gehörenden Daten des `page-travel`-Ereignisses betrachtet werden. Das Problem des „Seitenflatterns“ *(Page Thrashing)* ist so direkt erkennbar. Für die Darstellung von ausgelagerten Speicherseiten (andere Knoten oder Auslagerungsbereich des Betriebssystems) wurde ein neues Konzept eingeführt. Bei der Kommunikationszeitlinien-Darstellung ist die Anzeige der Verbindungslinien ermöglicht worden. Um eine aussagekräftige ASCII-Darstellung der Ereignisse in einer Trace-Datei zu erhalten, werden diese intern im Treiber entsprechend aufbereitet. Dazu erfolgt eine Kombination der Ereignisdefinition mit den aktuellen Daten des Ereignisses. Die Präsentation der Symbolnamen eines *SVM-Fortran*-Programms in *PARvis* ist möglich. Durch die Auswertung der in den *Stream Attributes* der Trace-Datei gespeicherten Namen der in dem Programm vorhandenen *SVM-Fortran*-Regionen ist der Quelltextbezug bei der Visualisierung der Trace-Daten erweitert worden.

Das Werkzeug *OPAL* geht bei der Visualisierung der Trace-Daten einen anderen Weg als *PARvis*. Hier erfolgt die Anzeige der Trace-Informationen in numerischer Form entsprechend dem instrumentierten Quelltext. Diese und weitere Fähigkeiten von *OPAL* sind zu Beginn des Kapitels 5 beschrieben worden. Nach einer Gegenüberstellung der den beiden Werkzeugen zugrundeliegenden Konzepte wurden hieraus Kriterien für eine Schnittstelle von *PARvis* zu *OPAL* entwickelt. Das Ziel bei Einführung dieser Schnittstelle war ein erweiterter Quelltextbezug bei der Performance-Analyse mit *PARvis*. Durch die Protokollierung von *SVM-Fortran*-Sprachereignissen durch den *SAM* ist ein Bezug zum Quelltext und eine Kopplung zu *OPAL* herstellbar. Diese Schnittstelle wurde als weitere neue Komponente von *PARvis* implementiert. Wichtigster Teil ist die Lokalisierung von Variablen auf Speicherseiten.

Ausgehend von dem implementierten SDDF-Treiber für *SVM-Fortran*-Programme können Treiber für weitere SDDF-basierte Trace-Formate realisiert werden. Die Auswertung von Trace-Daten des Debuggers der Intel Paragon *(ipd)* oder der *Pablo* Trace Capture-Bibliothek mit *PARvis* ist damit problemlos möglich.

Die im Rahmen dieser Arbeit entwickelte Statistik-Komponente von *PARvis* zeigt dem Anwender die ausgewählte Inkarnation einer Region. Hier wäre eine Erweiterung denkbar, die auf Wunsch sämtliche Inkarnationen der gewählten Region anzeigt. Zusätzliche Informationen können aus den vorhandenen Trace-Daten gewonnen werden, wenn diese durch interne Zähler (beispielsweise für den Zugriff auf bestimmte Speicherseiten) ausgewertet werden. Eine Filterung der Informationen fände dann statt. Dabei ist eine hierarchische Vorgehensweise anzustreben. Bestimmte Vorgänge wie *Page Thrashing* oder *False Sharing*, die die Performance der Programmausführung negativ beeinflussen, können so unmittelbar erkannt werden.

Langfristig wünschenswert wäre dabei der Einsatz von statistischen Methoden, die es gestatten, derartige Vorgänge aus den Trace-Informationen herauszufiltern. Denkbar wäre hier ein Ansatz für die statistische Analyse der Trace-Daten durch die Abbildung dieser Daten auf kontinuierliche Markov-Ketten. Dabei wird ein diskreter Zustandsraum genutzt. In diesem Zustandsraum können die an der Programmausführung beteiligten Prozessoren auf Zustände abgebildet werden. Die Betrachtung der Zustandsänderungen einzelner Speicherseiten wird dann möglich. Abbildung 6.1 verdeutlicht dies.

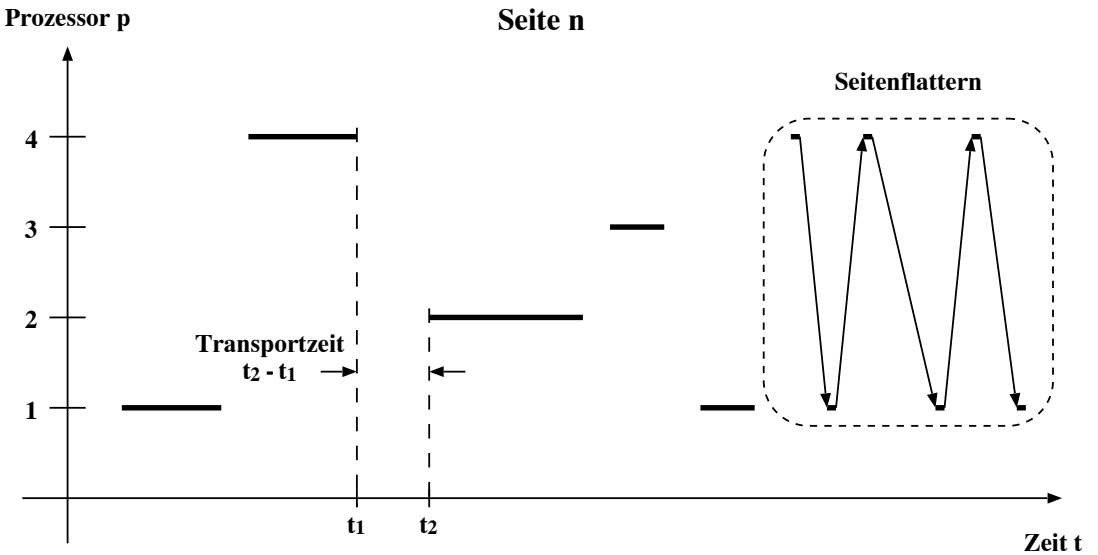


Abbildung 6.1: Zustandsänderungen einer Speicherseite

Ein *SVM-Fortran*-Programm kann mit einer guten Performance ablaufen, wenn die Datenlokalität hoch ist. Übertragen auf die Markov-Kette bedeutet dies, daß die Übergangsrate einer Speicherseite von einem Prozessor zu einem anderen klein sein

soll. Die Übergangsraten erhält man durch Auswertung der Trace-Daten. Punkte mit einer hohen Übergangsrate weisen auf eine schlechte Datenlokalität hin. Betrachtet man zusätzlich noch den neuen Zustand, sind beispielsweise auch Fehler wie das Seitenflattern (zwei Prozessoren beanspruchen eine Speicherseite sozusagen gleichzeitig, Abbildung 6.1) erkennbar. Die Zeit, die eine Speicherseite im Kommunikationsnetzwerk des Parallelrechners verbringt, ist aus der Zeit zwischen zwei Zuständen ablesbar. Durch die Diskretisierung des Zustandsraumes ist ein Effekt wie das Seitenflattern auch bei sehr unterschiedlichen Transportzeiten erkennbar. Eine automatische Auswertung von Trace-Daten, die durch eine Wissensbasis gestützt wird, ist somit denkbar. Damit können in der Benutzerunterstützung bzw. -führung neue Wege gegangen werden. Bedingt durch die Ergänzbarkeit der Wissensbasis erhalten die Werkzeuge eine gewisse „Eigenintelligenz“, wodurch sie sich adaptiv an die jeweilige Problemstellung anpassen können. Der Anwender soll dann nicht nur auf Performance-Engpässe in seinem Programm hingewiesen werden, das Werkzeug zur Performance-Analyse kann ihm durch seine Wissensbasis auch Vorschläge zur Verbesserung seines Programm-Codes machen.

In der Visualisierung von Trace-Informationen sind ebenfalls neue Konzepte zu entwickeln, um die Fülle der in den Trace-Daten vorhandenen Informationen in geeigneter Weise dem Benutzer zugänglich zu machen. Ob allerdings der in [RSTS95] beschriebene Weg der Verwendung von *Virtual Reality*-Methoden zur Visualisierung in die richtige Richtung geht, bleibt abzuwarten.

Literaturverzeichnis

[Adv93] V.S. Adve, *Analyzing the Behavior and Performance of Parallel Programs*, Computer Sciences Technical Report #1201, University of Wisconsin-Madison, 1993

[AdVe93] V.S. Adve and M.K. Vernon, *The Influence of Random Delays on Parallel Execution Times*, Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Performance Evaluation Review, Vol. 21, Iss. 1, 1993, p. 61–73

[Arn93] A. Arnold, *PARvis: Eine X-basierte Umgebung zur Visualisierung von parallelen Programmen in Multiprozessorsystemen*, Bericht des Forschungszentrums Jülich Jül-2848, 1993

[Arn95] A. Arnold, *User's Guide to PARvis*, Version 2.7, to be published, Central Institute for Applied Mathematics, Research Centre Jülich, 1995

[ADN95] A. Arnold, U. Detert, and W.E. Nagel, *Performance Optimization of Parallel Programs, – Tracing, Zooming, Understanding –*, Proc. 35th Semi-Annual Cray User Group Meeting (Spring), Denver, CO, 1995, p. 252–258

[Ayd94] R.A. Aydt, *The Pablo Self-Defining Data Format*, Department of Computer Science, University of Illinois, Urbana, IL, 1994

[BFKK90] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, *An Interactive Environment for Data Partitioning and Distribution*, 5th Distributed Memory Computing Conference, Charleston, SC, 1990, p. 1160–1170

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, *A Static Performance Estimator to Guide Data Partitioning Decisions*, 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Williamsburg, VA, 1991, p. 213–223

[BeGe95] R. Berrendorf and M. Gerndt, *SVM-Fortran Reference Manual Version 1.4*, Internal Report KFA-ZAM-IB-9510, Central Institute for Applied Mathematics, Research Centre Jülich, 1995

[BGM95] R. Berrendorf, M. Gerndt and M. Mairandres, *Programming Shared Virtual Memory on the Intel Paragon^{TM} Supercomputer*, Internal Report KFA-ZAM-IB-9509, Central Institute for Applied Mathematics, Research Centre Jülich, 1995

[BGNP93] R. Berrendorf, M. Gerndt, W.E. Nagel, and J. Prümmer, *SVM Fortran*, Internal Report KFA-ZAM-IB-9322, Central Institute for Applied Mathematics, Research Centre Jülich, 1993

[BCVY91] J. Bruner, H. Cheong, A. Veidenbaum, and P. Yew, *Chief: A Parallel Simulation Environment for Parallel Systems*, 5th Int'l Parallel Processing Symposium, Anaheim, CA, 1991, p. 568–575

[CoSm85] R.P. Cody and J.K. Smith, *Applied Statistics and the SAS Programming Language*, North-Holland, New York · Amsterdam · Oxford, 1985

[Cro94] M. Crovella, *Birds of a Feather Session on Performance Prediction*, Summary Report, Supercomputing '94, Washington, DC, 1994

[CBLM92] M. Crovella, R. Bianchini, T. LeBlanc, E. Markatos, and R. Wisniewski, *Using Communication-to-Computation Ratio in Parallel Program Design and Performance Prediction*, Proc. 4th IEEE Symposium on Parallel and Distributed Processing, IEEE, New York, 1992, p. 238–245

[DFK94] J. Docter, K. Fälski, and R. Knecht, *Paragon XP/S at KFA Jülich*, Technische Kurzinformation KFA-ZAM-TKI-0230, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich, 1994

[Dud88] *Duden «Informatik», ein Sachlexikon für Studium und Praxis*, hrsg. von H. Engesser, Dudenverlag, Mannheim, 1988

[Fah93] T. Fahringer, *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*, Dissertation, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, 1993

[Fah94] T. Fahringer, *Evaluation of Benchmark Performance Estimation for Parallel Fortran Programs on Massively Parallel SIMD and MIMD Computers*, Department of Software Technology and Parallel Systems, University of Vienna, to be published in: IEEE Proc. 2nd Euromicro Workshop on Parallel and Distributed Processing, Malaga, Spain, 1994

[FHKK90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, *Fortran D language specification*, Technical Report TR90-141, Dept. of Computer Science, Rice University, Houston, TX, 1990; and Technical Report SCCS-42c, Syracuse Center for Computational Science, Syracuse University, Syracuse, NY, 1991

[GaHu94] J. García, R. Hughey, *Scalable Visualization of Parallel Systems*, UCSC-CRL-94-29, Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz, CA, 1994

[Gem93] A.J.C. van Gemund, *Performance Prediction of Parallel Processing Systems: The Pamela Methodology*, Proc. 7th ACM Int'l Conference on Supercomputing 1993 (ICS '93), Tokyo, Japan, 1993, p. 318–327

[Ger94] M. Gerndt, *Performance Analysis Environment for SVM-Fortran Programs*, Internal Report KFA-ZAM-IB-9417, Central Institute for Applied Mathematics, Research Centre Jülich, 1994

[GKÖ95] M. Gerndt, A. Krumme and S. Özmen, *Performance Analysis for SVM-Fortran with OPAL*, to appear in: Proc. of the Int'l Conference on Parallel and Distributed Processessing Techniques and Applications (PDPTA '95), Athens, GA, USA, 1995

[GeBe95] M. Gerndt and R. Berrendorf, *Parallelizing Applications with SVM-Fortran*, Conference Paper High Performance Computing and Networking (HPCN '95), Milano, Italy, 1995, to appear in: Lecture Notes in Computer Science, Springer-Verlag

[Gir88] G.R. Gircys, *Understanding and Using COFF*, 1st Edition, O'Reilly & Associates, 1988

[Göb95] J. Göbel, *Evaluation of Shared Virtual Memory on the Paragon Supercomputer*, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, 1995

[HaMe92] F. Hartleb and V. Mertsiotakis, *Bounds for the Mean Runtime of Parallel Programs*, in R. Pooley and J. Hillston (Eds.), Edits Vol. 10, Computer Performance Evaluation '92: Modelling Techniques and Tools, Edinburgh University Press, 1993, p. 143–154

[HeFi94] M.T. Heath and J. Etheridge Finger, *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*, Users Guide, Oakridge National Laboratory, 1994

[HPF93] High Performance FORTRAN Forum, *High Performance FORTRAN Language Specification – PART I, Version 1.0*, ACM Fortran Forum, Vol. 12, No. 4, 1993

[HPF94] High Performance FORTRAN Forum, *High Performance FORTRAN Language Specification – PART II, Version 1.0*, ACM Fortran Forum, Vol. 13, No. 2, 1994

[HKKK91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.W. Tseng, *An Overview of the Fortran D Programming System*, in: U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Eds.),

Proc. 4th Int'l Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA, 1991, p. 18–34

[Int94] Intel Corporation, *Paragon Interactive Parallel Debugger Reference Manual*, Order No. 312547-003, Intel Supercomputer Systems Division, 1994

[Kar87] A.H. Karp, *Programming for Parallelism*, Computer, Vol. 20, No. 5, 1987, p. 43–57

[Kru95] A. Krumme, *SVM-Fortran Traces*, Revision: 0.13, Central Institut for Applied Mathematics, Research Centre Juelich, 1995

[Lin94] M.A. Linn, *Untersuchungen zur Ablaufplanung bei Parallelrechnern mit virtuell gemeinsamen Speicher*, Bericht des Forschungszentrums Jülich Jül-2931, 1994

[Lov93] D.B. Loveman, *High Performance Fortran*, IEEE Parallel and Distributed Technology, Vol. 1, No. 1, 1993, p. 25–42

[MaZe94] M. Mairandres and S. Zeisset, *Scalable and Efficient Management of Pages in Shared Virtual Memory Systems*, Intel European Supercomputer Development Center (ESDC), 1994

[MiSc90] H. Mierendorff and R. Schwarzwald, *LAPAS: A Performance Evaluation Tool for Large Parallel Systems*, 11. ITG/GI Fachtagung: Architektur von Rechnersystemen, VDE-Verlag, München, 1990

[Mül94] C.M. Müllender, *Visualisierung der Speicheraktivitäten von parallelen Programmen in Systemen mit virtuell gemeinsamem Speicher*, Bericht des Forschungszentrums Jülich Jül-2911, 1994

[NaAr93] W.E. Nagel und A. Arnold, *PARvis: Ein Werkzeug zur Visualisierung von parallelen Prozessen auf Mehrprozessorsystemen*, 7. ITG/GI-Fachtagung: Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, Aachen, 1993, p. 178–187

[NaAr94] W.E. Nagel and A. Arnold, *Performance Visualization of Parallel Programs: The PARvis Environment*, Proc. 1994 Intel Supercomputer Users Group Conference (ISUG '94), San Diego, CA, 1994, p. 24–31

[NaLi93] W.E. Nagel and M.A. Linn, *Benchmarking Parallel Programs in a Multiprogramming Environment: The PAR-Bench System*, in: J.J. Dongarra and W. Gentzsch (Eds.), Advances in Parallel Computing, Vol. 8: Computer Benchmarks, Elsevier Science Publishers B.V., The Netherlands, 1993, p. 303–322

[Ott95] H. Otto, *Entwicklung und Analyse von dynamischen Loop-Scheduling-Algorithmen für SVM-Fortran*, Diplomarbeit an der RWTH Aachen, to be published, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich, 1995

[Özm95] S. Özmen, *SAM: Performance-Analyse-Monitor für SVM-Fortran*, Bericht des Forschungszentrums Jülich Jül-3116, 1995

[Pab95] The Pablo Research Group, *A Description of the Classes and Methods of the Pablo SDDF Interface Library*, Department of Computer Science, University of Illinois, Urbana, IL, 1995

[PFTV88] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C; The Art of Scientific Computing*, Cambridge University Press, 1988

[PrKi92] J.P. Prost and S. Kipnis, *A Multilevel Trace-Driven Simulation Approach for Performance Analysis of Distributed-Memory Programs*, RC 17612 (#77650), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, 1992

[RAMN92] D.A. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, and B.W. Schwartz, *An Overview of the Pablo Performance Analysis Environment*, Department of Computer Science, University of Illinois, Urbana, IL, 1992

[RSTS95] D.A. Reed, W.H. Scullin, L.F. Tavera, K.A. Shields, and C.L. Elford, *Virtual Reality and Parallel Systems Performance Analysis*, IEEE Computer, to appear in November 1995

[Sar89a] V. Sarkar, *Determining Average Program Execution Times*, ACM Int'l Conference on Supercomputing (ICS '89), Reno, NV, 1989

[Sar89b] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, The MIT Press, Cambridge, MA, 1989

[Shi94] K.A. Shields, *iPablo User's Guide*, Department of Computer Science, University of Illinois, Urbana, IL, 1994

[SWG92] J.P. Singh, W. Weber, and A. Gupta, *SPLASH: Stanford Parallel Applications for Shared-Memory*, Computer Architecture News, Vol. 20, No. 1, 1992, p. 5–44

[Söt89] F. Sötz, *A Method for Performance Prediction of Parallel Programs*, Proc. IV. Int'l Conference on Vector and Parallel Processing CONPAR 90-VAPP, Zurich, Switzerland, 1990, p. 98–107

[SVS88] P. Stenström, D.F. Vrsalovic, and Z.Z. Segall, *Shared Data Structures in a Distributed System – Performance Evaluation and Practical Considerations*, Proc. Int'l Seminar on Performance of Distributed and Parallel Systems, Kyoto, Japan, 1988, p. 15–30

[Vos95] M. Voss, *Performance-Analyse mit Pablo Version 4.0*, Studienarbeit an der RWTH Aachen, to be published, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich, 1995

[VSSG84] D.F. Vrsalovic, D.P. Siewiorek, Z.Z. Segall, and E.F. Gehringer, *Performance Prediction for Multiprocessor Systems*, Proc. 1984 Int'l Conference on Parallel Processing, IEEE, 1984, p. 139–46

[Wan90] K.Y. Wang, *A Performance Prediction Model for Parallel Compilers*, Technical Report CSD-TR-1041, CAPO Report CER-90-43, Computer Science Dept., Purdue University, 1990

[YaVi91] N. Yazici-Pekergin and J.M. Vincent, *Stochastic Bounds on Execution Times of Parallel Programs*, IEEE Transactions on Software Engineering, Vol. 17, No. 10, 1991, p. 197–217

[ZBCM92] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, *Vienna Fortran – a Language Specification*, Technical Report, ICASE, ICASE Interim Report 21, Hampton, VA, 1992

Danksagung

Die vorliegende Diplomarbeit in Elektrotechnik ist am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule (RWTH) Aachen erstellt worden.

Dem Lehrstuhlinhaber Herrn Prof. Dr. F. Hoßfeld danke ich für die Möglichkeit, die Arbeit am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich (KFA) anfertigen zu können. Mein besonderer Dank gilt Herrn Dr. W.E. Nagel für die Betreuung der Arbeit und viele konstruktive Anregungen. Herr A. Arnold hat mir sehr bei der Einarbeitung in PARvis geholfen. Den Herren Dr. M. Gerndt und A. Krumme möchte ich für die Unterstützung bei der Gestaltung der Schnittstelle zwischen PARvis und OPAL danken. Sie haben mich ebenfalls, neben Herrn S. Özmen, bei der Programmierung des PARvis-Treibers für SVM-Fortran-Ereignisse beraten. Bei Herrn H. Bast von der Firma Intel bedanke ich mich für die Beantwortung aller Fragen hinsichtlich des ASVM-Kernels von Intel Paragon.

Ohne die Unterstützung und die Geduld meiner Frau und meiner Eltern könnte diese Arbeit nicht den Abschluß meines Studiums darstellen. Deswegen spreche ich Ihnen an dieser Stelle meinen ganz persönlichen Dank aus.