Diss. ETH Nr. 9880

Uebersetzung funktionaler Sprachen für den ADAM-Parallelrechner

ABHANDLUNG
zur Erlangung des Titels
DOKTOR DER TECHNISCHEN WISSENSCHAFTEN
der
EIDGENÖSSISCHEN TECHNISCHEN HOCHSCHULE
ZÜRICH

vorgelegt von
Stephan B. Murer
Dipl. Informatik-Ing. ETH
geboren am 26. Februar 1963
von Beckenried (NW) und Zürich

angenommen auf Antrag von
Prof. Dr. A. Kündig, Referent
Prof. Dr. H. Burkhart, Korreferent
1992

Leer - Vide - Empty

Übersetzung funktionaler Sprachen für den ADAM-Parallelrechner

Stephan Murer

Leer - Vide - Empty

Vorwort

Das ADAM-Projekt ist eine Team-Arbeit und ich möchte mich bei allen Mitgliedern der ADAM-Forschungsgruppe (Richard Bührer, Philipp Färber, Hansjürg Lips, Olivier Maquelin, Reto Marti, Srdjan Mitrovic, Patrick Schibli, Milan Tadjan und Jürg Wyttenbach) für die lange, gute Zusammenarbeit bedanken. Ohne diese fruchtbare Zusammenarbeit wäre die Arbeit nie zu Stande gekommen. Mein besonderer Dank gilt J. Wyttenbach, der die Grundlagen für das ADAM-Projekt schuf, Olivier Maquelin, auf dessen Simulator alle experimentellen Resultate dieser Arbeit entstanden sind, und Reto Marti, mit dem ich während der ganzen Zeit am Institut das Büro und viele Ideen teilte. Neben den Mitarbeitern des ADAM-Projekts trug auch eine Anzahl von Studenten mit Studienarbeiten zur Realisierung des Projekts bei. Namentlich möchte ich dabei Th. Wüest, P. de Marchi und R. Boldini erwähnen.

Einen grossen Dank verdient auch Hans-Jörg Brundiers, der immer mit grossem aber unbürokratischem Einsatz unsere Computer-Infrastruktur aufrechterhalten hat, was bei all unseren Spezialwünschen nicht immer eine einfache Aufgabe war

Ebenfalls danken möchte ich Prof. A. Kündig, dem Betreuer meiner Doktorarbeit, der mir bei meiner Arbeit immer die notwendige Freiheit liess, und Prof. H. Burkhart für die Uebernahme des Korreferats.

Besonderer Dank gebührt meinen Eltern, die mir erst die Ausbildung ermöglichten, welche schliesslich zu dieser Arbeit führte. Sie haben zusammen mit meinem Bruder auch die ganze Arbeit sorgfältig gelesen und so noch manchen Druckfehler eliminiert.

Schliesslich möchte ich meiner Lebenspartnerin Monika Landolt für ihre liebevolle Geduld, Unterstützung und Toleranz während der ganzen Zeit danken.

Abstract

In this dissertation the necessary concepts to generate code for a new class of parallel computers, i.e. coarse-grain dataflow machines (or multi-threaded architectures, as they are called more recently), are demonstrated on the basis of a simple functional programming language. In a first, introductory chapter, an overview is given over some general requirements for parallel computers (scalability, programmability, general-purpose applicability). The analysis of these requirements has led to the concept of coarse-grain dataflow machines or multi-threaded architectures, as they have been called more recently. In the next two chapters the ADAM architecture as the target and the functional language MFL as the source for the compilation are defined as far as necessary. Finally, the concept of hierarchical acyclic dataflow graphs as intermediate code in the compiler is presented.

In the central part of the thesis we discuss the several novel code generation methods for coarse-grain dataflow computers. In this context the following problems have been solved:

1. The choice of a reasonable trade-off between parallelism and overhead for management and synchronization of parallel activities.

2. The efficient implementation of large data structures for functional languages on a parallel machine.

3. The choice of an optimum instruction sequence to fully exploit the potential parallelism of the machine.

Chapters in this part share the same structure: First the problem is introduced, then methods to solve it are shown and finally the solutions are validated with simulation experiments and compared to similar results in the literature.

In the following chapter, we discuss the implementation of the MFL compiler and its integration into a new kind of programming environment. After some conclusions and a number of proposals for further research, three appendices may be found, containing the syntax of MFL in the usual form, all source codes of the MFL benchmark programs and an example assembler code of a runtime routine for the management of large data structures.

Keywords: functional languages, data-flow architectures, parallel processors, multi-threaded architectures, code generation, optimization

Kurzfassung

In dieser Dissertation werden auf der Basis einer einfachen funktionalen Programmiersprache die notwendigen Konzepte aufgezeigt, um Code für eine neuartige Klasse von Parallelrechnern - die grobgranularen Datenflussmaschinen - zu generieren. In einem ersten, einführenden Kapitel wird ein Ueberblick über allgemeine Anforderungen an Parallelrechner (Skalierbarkeit, Programmierbarkeit und allgemeine Anwendbarkeit) gegeben. Die Analyse dieser Anforderungen hat zum Konzept der grobgranularen Datenflussrechner geführt. Die ADAM-Architektur als Zielarchitektur und die funktionale Sprache MFL als Quellensprache für die Uebersetzung werden soweit nötig beschrieben. Es wird dann das Konzept hierarchischer, azyklischer Datenflussgraphen als Zwischencode für funktionale Programme eingeführt.

Im zentralen Teil der Arbeit wird eine Reihe von neuen Codegenerierungstechniken für grobgranulare Datenflussrechner diskutiert. In diesem Kontext wurden die folgenden Probleme gelöst:

1. Die Wahl eines vernünftigen Kompromisses zwischen Parallelität und dem notwendigen Zusatzaufwand zur Verwaltung und Synchronisation paralleler Aktivitäten,

2. Die effiziente Implementation grosser Datenstrukturen für funktionale Sprachen auf einer parallelen Maschine,

3. Die Wahl einer optimalen Sequenz von Instruktionen, so dass das Parallelitätspotential der Maschine möglichst gut ausgenutzt werden kann.

In allen Kapiteln dieses Teils werden am Anfang die Probleme eingeführt, dann die Methoden zu deren Lösung gezeigt, die Lösungen anhand von Simulationsexperimenten validiert und schliesslich mit ähnlichen Arbeiten aus der Literatur verglichen.

Im nächsten Kapitel wird die Implementation des MFL-Compilers und seine Integration in eine neuartige Programmierumgebung diskutiert. Nach einigen Schlussfolgerungen und Hinweisen auf mögliche Folgearbeiten folgen noch drei Anhänge mit einer Syntax von MFL in der üblichen Notation, den MFL-Quellenprogrammen aller in den Experimenten verwendeten Beispielen und dem Assembler-Code für einen Teil des Laufzeitsystems.

Schlüsselbegriffe: Funktionale Sprachen, Datenfluss-Architekturen, Parallelrechner, Codegenerierung, Optimierung

Leer - Vide - Empty

Inhaltsverzeichnis

1. Einleitung

1.1. Warum Parallelrechner?

"In the world of computers and computation there are two phenomena that should be in balance but that are not: the supply of versus the demand for computing power." [Böhm83]

In der Physik, der Chemie und auch der Biologie wird heute in zunehmendem Masse das Experiment durch die computergestützte Simulation ersetzt. Dabei gibt es auch heute noch viele Probleme, die mit den schnellsten zur Verfügung stehenden Computern nicht gelöst werden können, obwohl die Algorithmen prinzipiell bekannt wären. Die Figur 1.1 gibt einen Ueberblick über einige wichtige Probleme, sowie Speichergrösse und Rechenleistung, die zur Lösung notwendig wären.

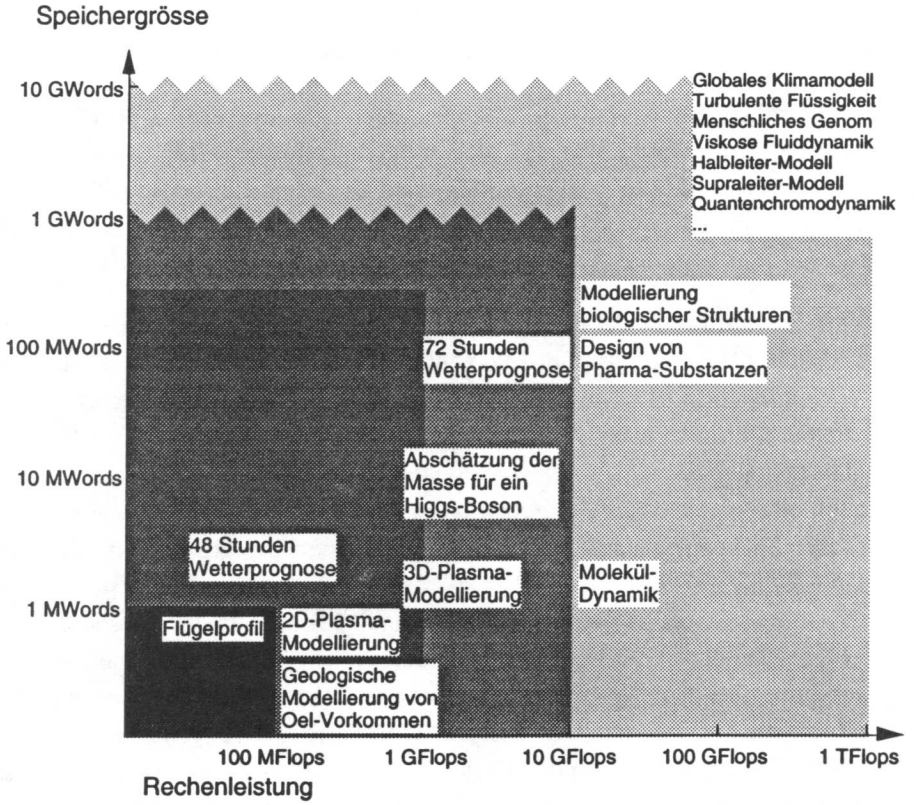


Fig. 1.1: Einige grosse Rechenprobleme mit Speicher- und Rechenbedarf [Parsyt92]

Auf der anderen Seite zeichnen sich bei der Verbesserung der klassischen Von-Neumann-Architektur mit einem Prozessor trotz Pipelining, überlappenden Speicherzugriffen und anderen Verbesserungen zur Ausnützung der Ressourcen unüberwindliche physikalische Grenzen ab [Ware72]. Die Zykluszeit der Cray-2 beträgt beispielsweise noch 4.2 Nanosekunden. Eine einfache Rechnung zeigt, dass die Signale ohne Berücksichtigung der Torlaufzeiten in dieser Maschine noch etwa einen Meter pro Zyklus zurücklegen. Berücksichtigen wir noch die zur Abstrahlung der Verlustleistung nötige Fläche, so wird klar, dass diese Entwicklung ein absehbares Ende hat.

Sollen um Grössenordnungen höhere Rechenleistungen erreicht werden, so ist dies nur noch mittels *Parallelprozessoren* möglich. Theoretisch wäre mit n identischen Prozessoren eine Beschleunigung um den Faktor n gegenüber der 1-Prozessor-Maschine möglich. In der Praxis ist allerdings die Beschleunigung viel kleiner und abhängig von Maschinenarchitektur und verwendetem Algorithmus.

1.2. Anforderungen an parallele Architekturen

An einen Parallelrechner müssen wir grundsätzlich die drei Anforderungen *Skalierbarkeit*, *Programmierbarkeit* und möglichst *universelle Anwendbarkeit* stellen. In den folgenden drei Abschnitten wird zu jedem dieser Begriffe erklärt, was damit gemeint ist, welche Auswirkungen die Anforderung auf die Systemarchitektur hat und warum die Anforderung wirtschaftlich bedeutend ist.

1. Zunächst muss ein Parallelcomputer skalierbar sein. Skalierbarkeit bedeutet, dass für ein Programm mit genügend algorithmischer Parallelität die Rechenleistung proportional mit der Anzahl Prozessoren ansteigt. Selbstverständlich ist dieser Anstieg nicht bis ins Grenzenlose möglich. Wenn also angegeben wird, dass eine Maschine skalierbar sei, so muss immer auch eine obere Grenze gegeben werden, für welche Skalierbarkeit gilt. In dieser Dissertation soll weder von Maschinen die Rede sein, die diese Grenze bei wenigen Dutzend Prozessoren ansetzen, noch von solchen, die Skalierbarkeit bis zu vielen Tausenden bis Millionen von Prozessoren postulieren. Im ersten Fall genügt die bereits reife Technologie, mehrere Prozessoren über einen gemeinsamen Bus auf den gleichen Speicher zugreifen zu lassen. Für den zweiten Fall sind weder die grundlegendsten Konzepte noch die zu verwendende Technologie klar. Möglicherweise werden dereinst dem menschlichen Gehirn nachempfundene Parallelrechner heute noch unvorstellbare Leistungen vollbringen.

Skalierbarkeit bedeutet, dass im ganzen System keine gemeinsamen globalen Ressourcen vorhanden sein dürfen, weil diese mit zunehmender Prozessorenzahl zum Flaschenhals im System werden. Insbesondere dürfen die Prozessoren nicht auf einen gemeinsamen, physikalischen Speicher zugreifen. Die gesamte Nutzbandbreite des Kommunikationssystems zwischen den Prozessoren muss proportional mit der Anzahl Prozessoren wachsen. In [MurFär92] werden diese Probleme am Beispiel eines gemeinsamen, verteilten Speichers für die ADAM-Architektur diskutiert.

Skalierbarkeit ergibt auch wirtschaftliche Vorteile, da die gleichen Komponenten (Mikroprozessoren) und Software-Werkzeuge in einer ganzen Computerfamilie mit Modellen unterschiedlicher Leistungsklassen verwendet werden können. Wenn das Problem der Programmierbarkeit gelöst wäre, wäre es schon heute in vielen Fällen billiger, eine Parallelmaschine mit konventioneller Technologie an Stelle eines teuren Supercomputers mit Spitzentechnologie einzusetzen.

2. Die zweite, zentrale Anforderung an einen Parallelrechner ist die *Programmierbarkeit*: Programmierbarkeit bedeutet, dass die Details der Hardware-Architektur eines Parallelrechners dem Applikationsprogrammierer verborgen bleiben. Der Programmierer sollte sich genau so wenig mit dem Problem befassen müssen, wie sein Programm auf mehreren Prozessoren ausgeführt, wie er sich heute im Normalfall nicht darum kümmern muss, auf welche Speicherstellen die Variablen eines Programms zu liegen kommen. Wir brauchen eine abstraktere Sicht von Parallelmaschinen, als wir heute haben.

   Die Forderung nach Programmierbarkeit hat einige Konsequenzen für die Systemarchitektur: Obwohl ein physisch gemeinsamer Speicher aus Gründen der Skalierbarkeit unmöglich ist, muss konzeptionell ein gemeinsamer Speicher angeboten werden. Explizite Meldungsübermittlung ist ein zu primitives Konzept für die allgemeine Programmierung. Ein Ausweg aus diesem Dilemma bietet ein physisch verteilter, logisch gemeinsamer [MurFär92] Speicher, wobei man den Preis bezahlt, dass die Speicherzugriffszeit nicht mehr konstant ist. Dies ist aber nicht so schlimm, da lange Speicherzugriffszeiten mit sinnvoller Arbeit an anderen Teilen des Programms versteckt werden können. Eine weitere, zentrale Anforderung ist eine effiziente Methode zur Synchronisation verschiedener paralleler Aktivitäten, da diese Grundoperation sehr häufig vorkommt.

Bei den Gesamtkosten eines Informatiksystems wird der Anteil der Softwarekosten (inklusive Software-Wartung) immer höher. Die Forderung nach Programmierbarkeit ist deshalb wirtschaftlich von äusserster Bedeutung. Wir sind der Meinung, dass hochparallele Maschinen erst dann wettbewerbsfähig werden können, wenn diese Anforderung erfüllt ist.

3. Die dritte Anforderung an Parallelmaschinen ist deren *universelle Anwendbarkeit*. Damit ist gemeint, dass jede Art von parallelem Programm mit vernünftiger Effizienz auf der gleichen Maschine ausgeführt werden kann.

   Es gibt parallele Algorithmen, bei denen sich die Arbeiten nicht a priori auf die verschiedenen Prozessoren verteilen lassen, da Verteilung der Arbeit auf die einzelnen Teile des Programms stark von den Eingabedaten abhängig ist. Diese Anforderung bedeutet für das System, dass es die Möglichkeit des dynamischen Lastausgleiches unterstützen muss. Die Kommunikationsbandbreite und die Rechenleistung müssen aufeinander abgestimmt sein. Heutige Multiprozessoren bieten in der Regel eine zu kleine Kommunikationsleistung im Vergleich zur vorhanden Rechenleistung. Dies ist nicht erstaunlich, da es einfacher ist, die Prozessorleistung innerhalb eines Chips als die Kommunikationsleistung zwischen den Komponenten zu steigern. Die fortschreitende Integration optischer und elektronischer Technologien lässt für die Zukunft hoffen. Parallelität verschiedener Granularität muss effizient unterstützt werden.

   Die Idee des Computers als universelle Informationsverarbeitungsmaschine ist der eigentliche Grund für die aussergewöhnliche wirtschaftliche Bedeutung der Informatik. Diese Idee sollte für Parallelrechner nicht aufgegeben werden, auch wenn es sich für gewisse Applikationen lohnt, Spezialmaschinen zu entwerfen und herzustellen. Die universelle Anwendbarkeit einer Computerfamilie ist die Voraussetzung dafür, dass diese in hohen Stückzahlen und somit in guter Qualität zu günstigen Preisen produziert werden kann. Ausserdem ist die massive Verbreitung von Standard-Software heute der einzig gangbare Weg, gute Software-Qualität zu geringen Kosten zu erzielen. Das Beispiel des PC-Marktes zeigt diese wirtschaftlichen Zusammenhänge exemplarisch.

1.3. Architektur, die diese Anforderungen erfüllt

Heutige Parallelrechnerarchitekturen erfüllen diese Anforderungen nicht oder nur teilweise. Maschinen mit *physisch gemeinsamem Speicher* sind zwar mit geeigneten Werkzeugen programmierbar und universell anwendbar. Allerdings sind sie nur in sehr begrenztem Masse skalierbar. In der nächsten Zukunft werden wahrscheinlich solche Maschinen den Markt dominieren.

Heutige Maschinen mit verteiltem Speicher, deren Prozessoren nur *Meldungen austauschen* können, erweitern die Grenze der Skalierbarkeit deutlich nach oben. Leider sind sie nicht programmierbar im geforderten Sinne, da die Architektur zuwenig stark vor dem Programmierer verborgen werden kann. Zudem ist bei derartigen Architekturen die Kommunikationsleistung in der Regel zu klein, als dass sie universell einsetzbar wären.

Die dritte Klasse von verbreiteten Maschinen sind die sogenannten *SIMD-Rechner (Single Instruction Multiple Data [Flynn72])*, bei denen eine grosse Anzahl von Prozessoren dasselbe Programm ausführen. Dadurch, dass sie auf allen Prozessoren synchron die gleiche Instruktion ausführen, vereinfacht sich die Struktur stark. In dieser Familie sind deshalb jene Maschinen mit den meisten Prozessoren anzutreffen. Dementsprechend werden die heutigen "Gigaflop-Rekorde" auch auf solchen Maschinen erzielt. Es gibt Programmiersysteme für derartige Maschinen, die eine recht abstrakte Programmierung erlauben. Das Problem liegt aber bei der universellen Anwendbarkeit. SIMD-Maschinen erlauben nur die Ausführung von regelmässigen, datenparallelen Algorithmen. Diese Form der Parallelität ist zwar sehr wichtig, aber bei weitem nicht die einzige.

Schon seit einiger Zeit gibt es, zumindest theoretisch, grundsätzliche Alternativen zu diesen Rechnerarchitekturen. In einem vielbeachteten Artikel [Backus78] wird argumentiert, dass sich das durch die von Neumann-Architektur mit ihrem einzigen Ort der Kontrolle (Instruktionszähler) implizierte Programmiermodell aus konzeptionellen Gründen nicht für Parallelcomputer eignet. Eine weitere Beschleunigung der Rechner ist nur mit Hilfe einer radikal anderen Architektur und Programmierung möglich; z. B. der Datenflussarchitektur und entsprechenden funktionalen Sprachen. Die Programme für einen solchen Datenflussrechner sind nicht mehr Instruktionsfolgen, sondern Datenflussgraphen mit Knoten und Kanten. Die Knoten entsprechen dabei den Operationen auf den Daten und die Kanten den Datenabhängigkeiten. Im Gegensatz zur Von-Neumann-Architektur wird eine Datenflussmaschine nicht vom Kontrollfluss, sondern vom Datenfluss gesteuert. Die

Grundoperation der Maschine heisst nicht mehr: "Führe die Operation aus, auf die der Instruktionszähler zeigt und erhöhe diesen!" Neu heisst sie: "Führe jede Operation aus, an deren Eingängen alle Daten vorhanden sind, und generiere an deren Ausgang jeweils ein neues Datum." Dies kann natürlich auch bei mehreren Knoten gleichzeitig der Fall sein: Wir haben implizite Parallelität!

Leider ist die Implementierung des oben beschriebenen Prinzips in der Praxis nicht so einfach wie in der Theorie. Es wird heute allgemein akzeptiert, dass feingranulare Datenflussmachinen mit Synchronisation für jede Instruktion nicht effizient realisierbar sind. Die wichtigsten Kritikpunkte am Prinzip des feingranularen Datenflusses sind in [GaPaKK82] und im Kapitel 2 "Die ADAM-Architektur" zusammengefasst. Die Tatsache, dass alle wesentlichen Datenfluss-Forschungsgruppen auf der Welt auf die Entwicklung hybrider Architekturen umgestellt haben, stützt diese These zusätzlich (vgl. Abschnitt 2.8 "Arbeiten mit ähnlichen Ansätzen"). Einen Ausweg aus diesem Dilemma bieten Datenflussarchitekturen mit variabler Granularität. Dabei wird das grundsätzliche Datenfluss-Operationsprinzip beibehalten. Die einzelnen Knoten im Graph entsprechen aber nicht mehr einzelnen Instruktionen, sondern ganzen Instruktionssequenzen. Man erspart sich so unnötige Synchronisation einzelner Instruktionen, die ohnehin sequentiell ablaufen müssen. Die Balance zwischen Rechenarbeit und Kommunikation kann vom Compiler oder vom Programmierer durch gezielte Wahl der Granularität für solche Instruktionssequenzen beeinflusst werden. Im Kapitel 2 "Die ADAM-Architektur" wird eine konkrete derartige Architektur, soweit für das Verständnis dieser Dissertation nötig, eingeführt.

Datenflussmaschinen sind im Sinne der theoretischen Informatik gleich mächtig wie Turingmaschinen [Böhm83]. Sie können also die gleiche Problemklasse lösen wie die heute gebräuchlichen Computer. Dennoch lassen sich nicht alle gebräuchlichen Programmiersprachen einfach zu Datenflussgraphen übersetzen. Wir werden uns in den Kapiteln 3 "MFL - Eine einfache Einfachzuweisungssprache" und 4 "Datenflussgraphen als Zwischencode für MFL" eingehend mit dem Problem befassen, welche Eigenschaften Programmiersprachen aufweisen müssen, damit sie leicht zu Datenflussgraphen übersetzt werden können.

1.4. Ziel, Fragestellung und Methode dieser Dissertation

Das übergeordnete Ziel der vorliegenden Arbeit ist, einen Beitrag zur besseren Entkoppelung zwischen Hardware-Architektur und Anwendungen auf parallelen Maschinen zu leisten. Wie schon diskutiert, ist Programmierbarkeit eine der zu

erfüllenden Hauptanforderungen, um einen ökonomischen Einsatz von parallelen Rechnern in Zukunft rechtzufertigen. Aus diesem Ziel ergab sich mit der Zeit eine Reihe von zunehmend konkreteren Fragestellungen:

- Wie müssen Programmiersprachen aussehen, so dass mit ihrer Hilfe ein Algorithmus genügend abstrakt dargestellt werden kann, dass er sich für eine parallele Ausführung unabhängig von der präzisen Architektur des verwendeten Computers eignet, sich dann aber doch in effizienten Code übersetzen lässt?

- Wie müssen Programmiersprache und Maschinenarchitektur aufeinander abgestimmt werden, damit eine effiziente Ausführung möglich ist?

- Aus welchen wesentlichen Schritten besteht der Uebersetzungsprozess von einem funktionalen Programm zu einem Maschinenprogramm?

- Welche Datenstrukturen sollen für die Zwischenergebnisse im Uebersetzungsprozess verwendet werden?

- Wie partitioniert man ein Programm so in parallele Abschnitte, dass der durch die parallele Ausführungsweise entstehende Zusatzaufwand die nutzbringende Rechenarbeit nicht übersteigt?

- Wie sollen grosse Datenstrukturen in einer funktionalen Sprache verwaltet werden, die aus Benutzersicht keine Zeigervariablen erlaubt?

- Wie müssen Datenstrukturen auf die verschiedenen Prozessoren verteilt werden, so dass ein effizienter Zugriff möglich ist?

- Wie muss innerhalb der sequentiellen Abschnitte des Programms die Reihenfolge der Instruktionen gewählt werden, damit einerseits unterschiedliche Speicherlatenzen effizient verborgen werden können und andererseits die Parallelität des Algorithmus möglichst erhalten bleibt?

Methodisch wurde der Weg gewählt, diese Fragen am Beispiel der Uebersetzung der primitiven, funktionalen Sprache MFL, zu beantworten. Es wurde ein Compiler-Prototyp für die Programmiersprache MFL konstruiert und implementiert, in welchem alle in dieser Dissertation gezeigten Konzepte eingebaut wurden. Die Bedeutung der Konzepte wurde, wenn immer möglich, durch simulierte Ausführung von Beispielcodes auf unserer grobgranularen Datenflussarchitektur ADAM überprüft. Wo nötig und möglich, werden die Konzepte auch theoretisch untermauert.

Allerdings stehen die konkreten Experimente auf dem Simulator im Vordergrund. Es handelt sich also nicht um eine vorwiegend theoretische, sondern um eine weitgehend experimentelle Arbeit.

1.5. Ueberblick über den Inhalt

Der Aufbau dieser Arbeit ist entsprechend der Struktur des MFL-Compilers (Figur 1.2) gewählt worden.

An vorderster Stelle nach der Einführung, im Kapitel 2 "Die ADAM-Architektur" folgt eine kurze Beschreibung der verwendeten Architektur, soweit sie für das Verständnis der weiteren Arbeit notwendig ist. Es wird auch kurz auf die Entwicklungsgeschichte der Architektur, die Entwurfsprinzipien und auf den Simulator, die konkrete Realisierung der Architektur, eingegangen. Dieses Kapitel definiert quasi die untere Schnittstelle dieser Arbeit.

Am Anfang des Uebersetzungsprozesses steht das Programm in lesbarer Form. Im Kapitel 3 "MFL - Eine einfache, funktionale Sprache" werden Syntax und Semantik der Programmiersprache MFL definiert.

Aus den syntaktischen Strukturen wird anschliessend ein hierarchischer, azyklischer Datenflussgraph generiert, der als gemeinsame Datenstruktur für alle weiteren Phasen des Compilers dient. In Kapitel 4 "Datenflussgraphen als Zwischencode für MFL" wird die Struktur dieser Graphen anhand von MFL-Beispielen dokumentiert.

Kapitel 5 "Partitionierung von Datenflussgraphen" ist das erste Kapitel des eigentlichen Kerns der Arbeit. Es wird dort zuerst das allgemeine Problem der optimalen Partitionierung paralleler Aktivitäten bei gegebenen Kommunikations- und Rechenkosten studiert. Wir setzen uns dann kritisch mit einigen bestehenden Ansätzen auseinander und schlagen eine einfachere, wenn auch nicht so "optimale" Lösung vor, die sich auf eine Verbesserung an den wirklich kritischen Stellen im Programm konzentriert.

In Kapitel 6 "Effiziente Verwaltung von Datenstrukturen" wird das Problem diskutiert, wie sich grosse Datenstrukturen in funktionalen Sprachen verwalten lassen. Eine konventionelle Implementation, die jeder strukturierten Variablen einen festen, eigenen Speicherbereich zuordnet, wäre unter diesen Umständen äusserst ineffizient. Deshalb werden Datenstrukturen grundsätzlich nur als Heap-Objekte angelegt. Wenn möglich, werden nur Referenzen auf die Datenstrukturen

kopiert. Die effiziente Verwaltung solcher Datenstrukturen bietet, insbesondere in einem parallelen Umfeld, eine Reihe von Problemen, die durch das MFL-Laufzeitsystem und die ADAM-Hardware in Funktionsteilung gelöst werden.

Das Kapitel 7 "Sequentialisierung und Codegenerierung" bespricht die Frage, wie man innerhalb eines sequentiellen Codeblocks von der durch die Datenflussgraphen gegebenen Halbordnung zu einer, für die sequentielle Ausführung notwendigen, totalen Ordnung von Instruktionen gelangt. Dabei ist darauf zu achten, die Instruktionen so zu ordnen, dass die Möglichkeiten zur parallelen Ausführung auf der ADAM-Architektur optimal ausgeschöpft werden.

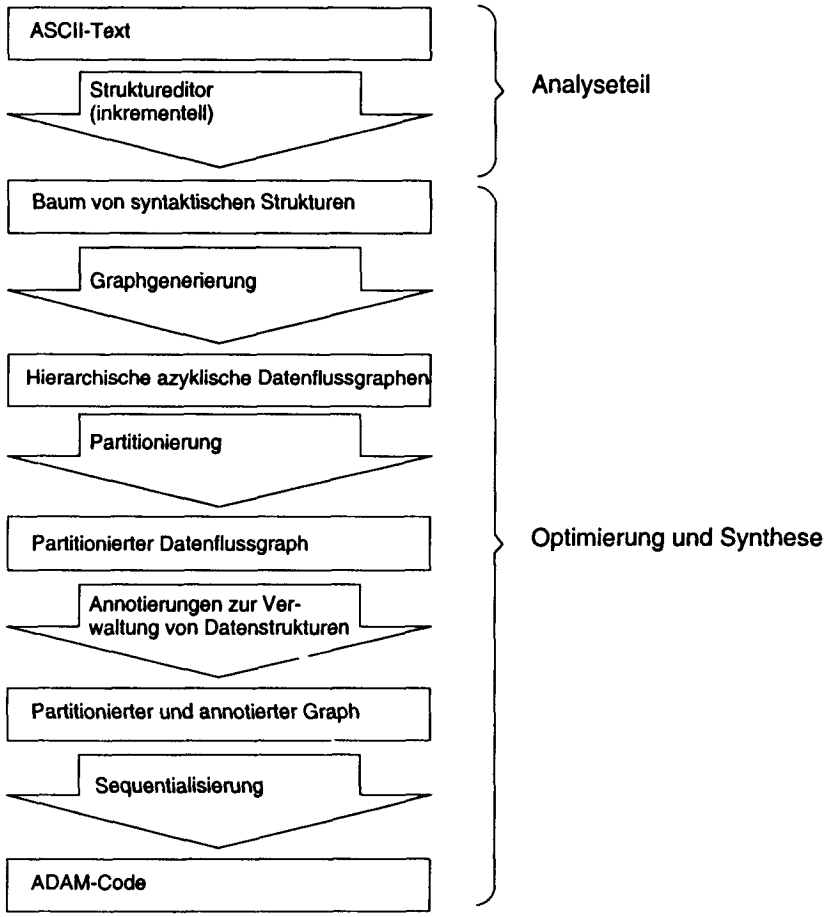


Fig. 1.2: Generelle Struktur des MFL-Compilers

Zusätzlich zu den in den Kapiteln 5 bis 7 beschriebenen Teilen der Codegenerierung werden auf den Graphen auch eine Reihe von konventionellen Optimierungstransformationen durchgeführt. In der beschriebenen Version des Compilers sind dies "Constant folding" (Konstante Ausdrücke werden schon zur Uebersetzungszeit berechnet), Entfernung von schleifeninvarianten Berechnungen aus dem Schleifenkörper und Entfernung von redundanten Teilen in Ausdrücken ("Common Subexpression Elimination"). Man beachte, dass sich solche Transformationen auf Datenflussgraphen besonders leicht durchführen lassen. Analoge Algorithmen wurden für SISAL implementiert [SkeWel85].

Die beiden abschliessenden Kapitel 8 "Implementation" und 9 "Ausblick und Evaluation" geben einen Ueberblick über die technische Realisierung der beschriebenen Konzepte, fassen die erreichten Ziele noch einmal zusammen und stecken den Rahmen für weiterführende Arbeiten ab.

1.6. Verwandte Arbeiten am ADAM-Projekt

Die vorliegende Arbeit wurde im Rahmen des Projekts ADAM geschrieben, dessen globales es Ziel ist, einen skalierbaren, programmierbaren und universell verwendbaren Parallelrechner zu konstruieren. Dabei wurde von allem Anfang an die Devise verfolgt, dass Rechnerarchitektur keine reine Hardware-Frage, sondern eine gemeinsame Optimierung von Theorie, Programmiermodell, Programmiersprachen, Programmierumgebungen, Simulation, Hardwarekonzepten und Technologie ist. All diese Gebiete beeinflussen sich gegenseitig stark und es ist das ehrgeizige Ziel des ADAM-Projekts, eine innovative, durchgehende Lösung für alle Bereiche anzubieten.

An den verschiedenen Fragestellungen des ADAM-Projekts sind verschiedene Mitarbeiter in unserer Forschungsgruppe tätig. Es ist deshalb wichtig, einen gewissen Ueberblick über den Kontext dieser Arbeit zu haben.

Die ADAM-Gruppe hat bereits eine lange Tradition im Bereich der Parallelrechner-Forschung. Im EMPRESS-Projekt [BBBBFH82] wurde schon früh ein vollständiger Prototyp eines Multiprozessors implementiert. Schon bald wurde jedoch klar, dass sich Computerarchitektur nicht als reines Hardware-Problem betrachten lässt. So legte eine weitere wichtige Arbeit [BühEka87] den Grundstein zur Idee, traditionelle Ansätze der Computerarchitektur mit Datenflussprinzipien zu verknüpfen.

In der Dissertation Wyttenbach [Wytten90] wurde dann das Konzept für eine erste Version des ADAM-Rechners und der dafür notwendigen Programmierwerkzeuge vorgelegt. In der Zwischenzeit wurden die Arbeiten am ADAM Projekt fortgesetzt und vertiefte Resultate aus den Bereichen Architektur/Simulation, Codegenerierung für die Sprache SISAL auf der ADAM-Architektur und Konzepte für eine moderne Programmierumgebung werden im Rahmen einer Reihe von weiteren Dissertationen erarbeitet. An einer Hardware-Implementation des ADAM-Konzepts wird noch gearbeitet.

Leer - Vide - Empty

2. Die ADAM-Architektur

In diesem Kapitel wird zuerst kurz auf den geschichtlichen Hintergrund und dann auf die verschiedenen Komponenten der ADAM-Zielarchitektur eingegangen, die auf die Codegenerierung einen Einfluss haben. Dann wird ein Blockdiagramm der gesamten Architektur gegeben, um eine Idee über das Zusammenwirken der verschiedenen Elemente zu vermitteln. Zuletzt folgt eine kurze Beschreibung des ADAM-Simulators, auf dem alle Experimente für diese Arbeit durchgeführt wurden. Die ADAM-Architektur ist die untere Schnittstelle für den in dieser Arbeit beschriebenen Compiler.

Die Konzepte der ADAM-Architektur werden in diesem Kapitel nur fragmentarisch und informell beschrieben. Vorkenntnisse auf dem Gebiet der Rechnerarchitektur und der Datenflussidee im allgemeinen erleichtern die Lektüre dieses Kapitels. Es wird nur auf jene Konzepte eingegangen, bei denen sich die ADAM-Architektur wesentlich von anderen Computerarchitekturen unterscheidet. Für eine detaillierte und formale Beschreibung der Architektur und der dahinterstehenden Ideen sei der interessierte Leser auf andere Dissertationen aus dem ADAM-Projekt verwiesen [Wytten90, Maquel92]. Da die Struktur der Maschine grundsätzlich zyklisch ist, d.h. alle Komponenten der Architektur von den meisten anderen abhängen, war es schwierig, eine Reihenfolge für die Beschreibung der Komponenenten zu wählen. Mit Hilfe verschiedener Querverweise sollte man trotzdem einen Weg durch das Kapitel finden.

2.1. Datenflussrechner: Weg vom von Neumann-Konzept

Schon früh erkannte man, dass das von Neumann-Programmiermodell nicht unbedingt die geeignete Grundlage für die Konstruktion paralleler Algorithmen ist. In einem berühmten Artikel [Backus78] wird auf die grundlegenden Probleme dieses Modells hingewiesen: Jede Zustandsänderung verursacht einen grossen Verkehr von einzelnen Datenworten zwischen Speicher und Prozessor (Adressen, Instruktionen, Operanden und Resultate)[1], und der Kontrollfluss ist inhärent sequentiell, vorgegeben durch den Programmzähler, der nach jeder Instruktion inkre-

---

[1]) Man redet in diesem Zusammenhang auch vom "von Neumann-Flaschenhals" und meint damit die nur ein Wort breite Verbindung zwischen Speicher und Prozessor, die sich in Programmiersprachen beim Zuweisungsoperator zeigt.

mentiert wird. Diese Beschränkungen schlagen sich auch in den Konzepten der meisten, heute verwendeten Programmiersprachen nieder.

In der Theorie des λ-Kalküls [Stoy77] war schon lange bekannt, dass es verschiedene (auch parallele) Berechnungsstrategien für mathematische Ausdrücke gibt, die zum gleichen Ergebnis führen. In funktionalen Programmiersprachen und entsprechenden Datenflussgraphen (vgl. Kap. 3 "MFL - Eine einfache, funktionale Sprache" und 4 "Datenflussgraphen als Zwischencode für MFL") wird diese Eigenschaft benutzt und ein Programm als eine partielle Ordnung unter Operatoren dargestellt. Mit Datenflussgraphen ist eine Darstellungsweise für funktionale Programme gemeint, bei der die Knoten den Operationen und die Kanten den Datenabhängigkeiten entsprechen (vgl. auch Figur 2.1).

Das entsprechende Maschinenmodell ist sehr einfach und geht auf [Dennis75] zurück. Ein Datenfluss-Knoten kann feuern, sobald an allen seinen Eingängen Marken vorhanden sind. Die Marken tragen Datenwerte. Beim Feuern wird an jedem Eingang eine Marke konsumiert, aus den entsprechenden Datenwerten das Resultat berechnet und dieses auf der Ausgangskante ausgegeben. Kanten können eine beliebige Anzahl Marken in FIFO-Speichern zwischenspeichern, ohne dass sich die Semantik des Programms ändert [Kahn74].

Das Modell hat zwei wichtige Eigenschaften: Parallelität - mehrere Knoten können gleichzeitig feuern, falls sie nicht voneinander abhängig sind - und Determiniertheit - die Resultate sind nicht von der Berechnungsreihenfolge abhängig. Es gibt statische und dynamische Datenflussarchitekturen. Dynamische Datenflussarchitekturen unterstützen Rekursion, d. h., es können mehrere Inkarnationen desselben Graphen abgearbeitet werden, indem man den Marken zusätzlich zu einem Datenwert noch eine Farbe für die Inkarnation zuweist [ArvNik89].

Als Beispiel für eine moderne Datenflussarchitektur soll kurz die ETS[2]/Monsoon-Architektur vorgestellt werden [Papado88]. Die Figur 2.1 zeigt einen Datenflussgraphen und das entsprechende Maschinenprogramm für die Monsoon-Maschine.

---

[2]) ETS heisst "Explicit Token Store". Dies deutet auf den Unterschied dieser moderneren Datenflussarchitektur zu älteren Ideen [TTDA, Manchester] hin: Die Operanden müssen nicht mehr durch eine "Matching Unit" mit teurem Assoziativspeicher zu Paaren geformt werden, sondern finden sich an einem bestimmten Platz im "Frame", eben explizit.

Jede Instruktion besteht aus der auszuführenden Operation, dem Operandenregister und einer oder zwei Zielinstruktionen für das Resultat.

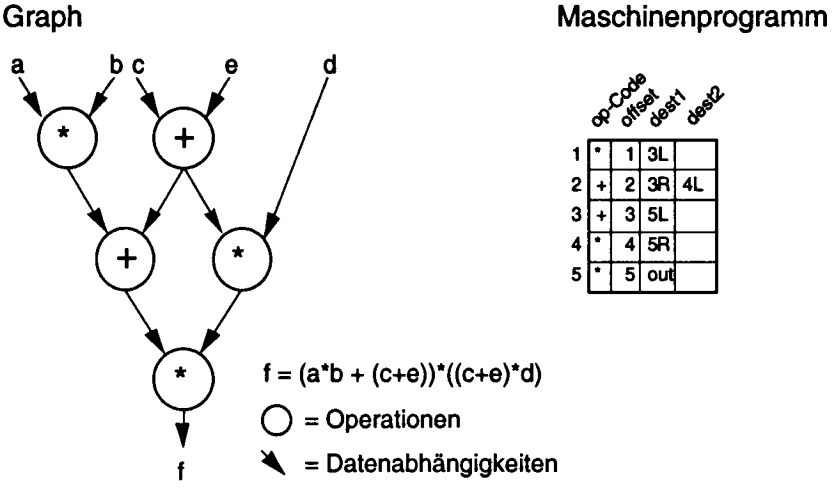


Fig. 2.1: Graph und Maschinenprogramm

Die Maschine besteht aus einer ringförmigen Pipeline (vgl. Fig. 2.2). In dieser zirkulieren Marken ("Token"), bestehend aus einem Rahmenzeiger, der auf den Anfang des entsprechenden Aktivationsrahmens zeigt (Es können mehrere Inkarnationen des gleichen Graphen im System existieren), einem Instruktionszeiger, dem Wert und dem Port, der anzeigt, ob der Wert der linke oder rechte Operand für die Instruktion ist. In der "Instruction-fetch"-Einheit wird die entsprechende Instruktion aus dem Programmspeicher geholt und mit Hilfe des Framezeigers die absolute Position des Operandenregisters im "Frame-Store" bestimmt. Nun gibt es zwei Fälle: Wenn sich bereits ein Operand im Register befindet, so ist der neue Wert der zweite Operand und die Instruktion kann direkt ausgeführt werden. Ist das Operandenregister noch leer, so wird der Wert als erster Operand dort gespeichert. Nach dem Rechnen werden aus dem Resultat, dem Ziel und dem aktuellen Framezeiger wieder ein oder zwei neue Token gebildet, die wieder in die Warteschlange zurückgehen.

Ein echter Multiprozessor wird realisiert, indem mehrere solche Ringe über ein Netzwerk verbunden werden, über das Token ausgetauscht werden können. Es gibt bereits eine Reihe von Prototyp-Implementationen für derartige Maschinen [GuKiWa85, Papado88, HiNSSY87, SaYHKY89]

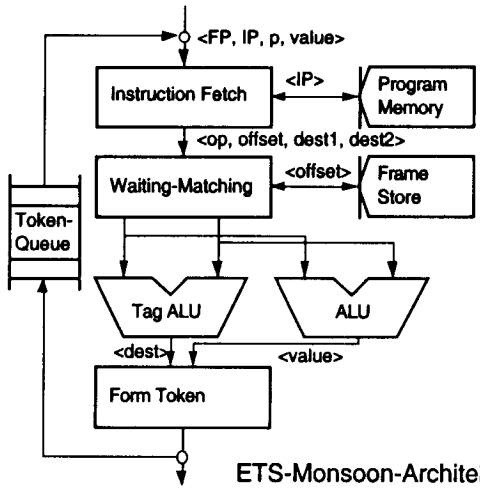


Fig. 2.2: ETS/Monsoon-Datenflussarchitektur, nach [Papado88]

2.2. Codeblöcke: Datenfluss mit variabler Granularität

Das Datenflussprinzip ist ein bestechender Ansatz, um die ganze Parallelität eines Programmgraphen auszunützen. Es gibt aber einige gewichtige Probleme:

1. Sequentielle Programmteile werden, wegen der langen Pipeline, sehr ineffizient ausgeführt.

2. Man hat eher zuviel als zuwenig Parallelität, so dass die Maschine mit Marken überschwemmt werden kann[3].

3. Der Synchronisationsaufwand ist gross im Vergleich zum eigentlichen Rechenaufwand, weil auf der Ebene der einzelnen Instruktionen synchronisiert werden muss.

4. Lokalität des Speichers kann nur schlecht ausgenützt werden, da die einzelnen Tokens stochastisch verteilt werden.

Dennoch lösen Datenflussarchitekturen zwei grundsätzliche Probleme des parallelen Rechnens [ArvIan87]:

---

3) Dieses Problem ist in der englischen Fachliteratur zum Thema unter dem Begriff "Throttling problem" bekannt.

1. Die *Speicherlatenz*, die in grossen Multiprozessoren prinzipiell auftritt, kann versteckt werden, indem asynchron auf den Speicher zugegriffen wird und während der Latenzzeit des Speichers der Prozessor durch andere Arbeiten ausgelastet werden kann.

2. Der *Namensraum für die Synchronisation* ist gross genug[4].

Die Vorteile des Datenflusskonzepts lassen sich mit den Vorteilen der klassischen Rechnerarchitektur kombinieren, indem man von der Idee abgeht, nur einzelne Instruktionen als Einheiten der Synchronisation zu betrachten [Babb84, BühEka87, Ianucc88]. Man lässt die Maschine ganze Instruktionssequenzen abarbeiten, bevor wieder über die Daten synchronisiert wird.

In der ADAM-Architektur werden solche Instruktionssequenzen Codeblöcke genannt. Codeblöcke haben ähnlich wie Prozeduren in klassischen Sprachen eine statische Natur, den Code, und eine dynamische Natur, den Kontext. Ein Codeblock ist in der Regel ähnlich gross wie eine Prozedur (10 - 1000 Instruktionen), wird aber auf der Maschine eher wie ein Prozess in einem klassischen Betriebssystem behandelt. Dies bedeutet, dass sich mehrere Codeblöcke den Prozessor, auf dem sie laufen, teilen. Codeblöcke werden nach Regeln, auf die noch zurückgekommen wird, gestartet, suspendiert und terminiert. Die richtige Grösse der Codeblocks, bzw. das optimale Verhältnis zwischen Rechenarbeit und Kommunikation ist ein wichtiges Problem für den Codegenerator und wird in Kap. 5 "Partitionierung von Datenflussgraphen in Codeblöcke" eingehend diskutiert.

2.3. Lebenszyklus eines Codeblocks

Es gibt zwei Maschinenstruktionen um einen neuen Codeblock zu erzeugen: CALL und PARCALL. Die CALL-Instruktion entspricht ungefähr dem Fork im bekannten Fork-Join-Konstrukt [PetSil85]. Es wird ein *Token* erzeugt, das alle wesentlichen Angaben zur Ausführung eines Codeblocks enthält. Bei diesen Daten handelt es sich um die Referenz auf das *Parameter-Objekt* für diesen Aufruf, die Adresse für das *Resultat-Objekt* und einen Zeiger auf den entsprechenden Code[5]. Dieses

---

4) Es wird prinzipiell auf jedes Speicherwort synchronisiert.

5) In der Praxis untersuchen wir auch Methoden, die mehr als einen Parameter direkt in das Token verpacken. Ausserdem kann für jeden CALL eine Priorität angegeben werden, die sich dann

Token kann sich nun frei zwischen den Prozessoren bewegen, bis ein Prozessor freie Kapazität zur Ausführung des entsprechenden Codeblocks hat. Ist ein solcher Prozessor gefunden, so wird das Token expandiert. Das bedeutet, dass dem Codeblock ein entsprechender Kontext, bestehend aus einem *Register-Frame* und einem Instruktionszähler zugeordnet wird. Im Anschluss daran, wird der Codeblock in den Zustand "ready" versetzt.

Sobald der aktuell laufende Codeblock suspendiert oder terminiert wird, kommt einer der bereiten (im Zustand "ready") Codeblöcke zum Laufen. Suspendierte Codeblöcke kommen in den Zustand "waiting", aus dem sie nach endlicher Zeit wieder aufgeweckt werden und in den Zustand "ready" wechseln. Die Operationen, die zur Suspendierung bzw. zum Aufwecken eines Codeblocks führen können, werden im Abschnitt 2.5 "Der Objekt-Manager: Ein verteilter, gemeinsamer Speicher" erklärt.

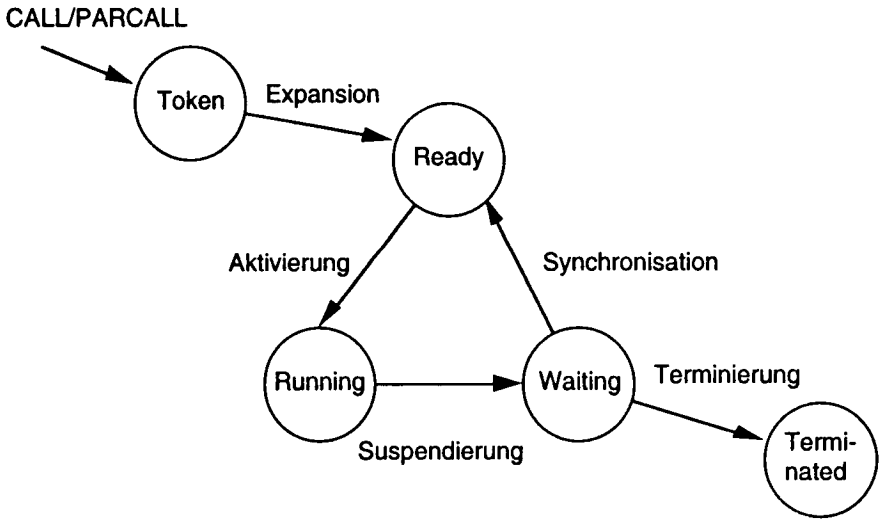


Fig. 2.3: Lebenszyklus eines Codeblocks

Die PARCALL-Instruktion funktioniert nur geringfügig anders. Anstelle eines einzelnen werden gerade mehrere Tokens gleichen Inhalts erzeugt. Ausserdem wird nicht eine Adresse für das Resultat-Objekt, sondern ein zählendes Semaphor mitgegeben, damit auf die Terminierung aller gemeinsam gestarteten Codeblöcke

---

auf die Schedulingstrategie auswirkt. Auf die Codegenerierung haben aber diese Details keinen Einfluss, so dass hier von einem vereinfachten Modell ausgegangen wird.

gewartet werden kann. Die PARCALL-Instruktion wird gebraucht, um den Schleifenkörper von parallelen Schleifen effizient parallel auszuführen.

Im aufrufenden Codeblock wird auf das Vorhandensein des Resultats synchronisiert (vgl. Abschnitt 2.4 "Das Register-Frame: Synchronisation über Daten"), um das Ende eines Untercodeblocks zu erkennen. Codeblöcke werden also explizit gestartet, und es wird implizit auf deren Terminierung gewartet.

Interessant an diesem Konzept ist, dass sich ein Codeblock als Token noch frei zwischen den Prozessoren bewegen kann und erst nach seiner Expandierung endgültig einem Prozessor zugeordnet werden kann. Diese Idee erlaubt den *dynamischen Lastausgleich* zwischen den Prozessoren nach folgendem Muster: Solange auf einem Prozessor noch Codeblöcke bereit (im Zustand "ready") sind, werden keine neuen Tokens expandiert. Zudem wird die Anzahl Tokens zwischen benachbarten Prozessoren durch Austausch ausgeglichen, so dass sich die Arbeit nach dem Muster der Wärmeleitungsgleichung auf der ganzen Maschine verteilt, nahe verwandte[6] Codeblöcke aber auf nahe beieinander liegenden Prozessoren ausgeführt werden. Die genauen Details dieses Lastverteilungsmechanismus sind recht komplex und genauer in [Maquel90] beschrieben.

2.4. Das Register-Frame: Synchronisation über Daten

Sobald ein Codeblock expandiert wird, bekommt er auf seinem Prozessor ein "Register-Frame", bestehend aus einer fixen Anzahl von Allzweckregistern, zugeteilt. In diesem geschieht die eigentliche Synchronisation über Daten[7]. Neben den eigentlichen Registern enthält ein solches Frame noch drei zusätzliche Bits pro Register.

Es gibt ein *Valid-Bit*, das angibt, ob im zugehörigen Register ein gültiger Wert abgespeichert ist. Sobald während des Programmablaufs ein Laufzeitfehler auf

---

6) Dabei bedeutet "Verwandtschaft" zwischen zwei Codeblöcken, dass der eine den anderen aufgerufen hat.

7) Der dabei ablaufende Mechanismus gleicht dem Frame-Konzept in der ETS/Monsoon-Architektur. Wegen dieser Daten-Synchronisation können wir bei ADAM überhaupt von einer Datenflussmaschine sprechen.

tritt[8], wird das Valid-Bit des zugehörigen Resultat-Registers zurückgesetzt. Damit sich der Fehler nicht fortpflanzen kann, werden vor der Ausführung jeder Operation die Valid-Bits der Operanden überprüft. Das Resultat wird immer ungültig, falls ein Operand ungültig ist (vgl. 2.6 "Der Exekutor: Ein spezieller RISC-Prozessor").

Present und *Wait-Bit* dienen der Synchronisation. Vor der Ausführung einer Instruktion werden die Present-Bits aller Operanden überprüft. Falls ein nicht gesetztes Present-Bit gefunden wird, wird das Wait-Bit des entsprechenden Registers gesetzt und der Codeblock suspendiert. Vom Objekt-Manager (vgl. Abschnitt 2.5 "Der Objekt-Manager: Ein verteilter, gemeinsamer Speicher") treffen nun im Hintergrund die erwarteten Daten ein, und die entsprechenden Present-Bits können gesetzt werden. Sobald die Daten für jenes Register eingetroffen sind, dessen Wait-Bit gesetzt ist, wird der Codeblock wieder in den Zustand "ready" gesetzt und kann bei Gelegenheit seine Arbeit fortsetzen. Die Figur 2.4 zeigt die wesentlichen Phasen einer solchen *asynchronen Transaktion.*

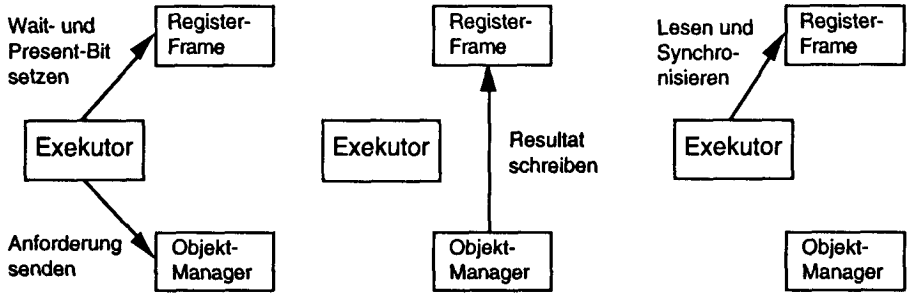


Fig. 2.4: Der Ablauf einer asynchronen Aktion [Maquel90]

Dieser Mechanismus erlaubt es, prinzipiell jede Instruktion asynchron auszuführen und auf deren Resultate zu warten. Diese Möglichkeit wird aber nur von jenen Instruktionen ausgenützt, die nicht direkt auf dem Exekutor ausgeführt werden können. Dabei handelt es sich einerseits um CALLs und PARCALLs, die die Ausführung eines neuen Codeblocks auf einem anderen Prozessor initieren und andererseits um alle Befehle, die vom Objekt-Manager ausgeführt werden.

---

[8]) Man denke dabei beispielsweise an eine Division durch Null oder einen arithmetischen Ueberlauf.

Da ein Kontextwechsel (Suspendierung des laufenden Codeblocks) nicht kostenlos ist, ist es leicht zu verstehen, dass die Sequenz der Instruktionen eine wichtige Rolle bei der Codegenerierung spielt. Vereinfacht gesagt geht es darum, die Instruktion, welche das Present-Bit zurückgesetzt hat (Produzent) und jene, welche die entsprechenden Daten als Operand braucht (Konsument), in der Sequenz möglichst weit auseinander zu positionieren, damit ein Kontextwechsel lange verzögert oder gar verhindert werden kann. Das Kapitel 8 "Sequentialisierung und Codegenerierung" befasst sich eingehend mit diesem Problem.

2.5. Der Objekt-Manager: Ein physisch verteilter, logisch gemeinsamer Speicher

Bei der ADAM-Architektur handelt es sich um einen Multiprozessor mit verteiltem Speicher. Es gibt also auf der Ebene der Hardware keinen gemeinsamen Speicher, wie etwa in busgekoppelten Multiprozessoren üblich. Dennoch kann man sich auf einer höheren Ebene den Speicher als eine Menge von *Objekten* ansehen, die von allen Prozessoren aus transparent zugreifbar sind.

Objekte sind mit einem Referenzzähler versehene Speicherbereiche beliebiger Länge. Alle Objekte sind durch eine Objektidentifikation eindeutig gekennzeichnet. Verschiedene Operationen sind auf einem Objekt möglich:

1. ALLOCATE: Ein Objekt kann unter Angabe der gewünschten Objektgrösse und Objektklasse alloziert werden. Der Objekt-Manager reserviert den entsprechenden Speicherbereich und liefert die Objektidentifikation des neuen Objekts, falls Platz vorhanden ist.

2. REF, DEREF, DEALLOCATE: Mit REF bzw. DEALLOCATE/DEREF kann der Referenzzähler des Objekts um eins erhöht bzw. verkleinert werden. Nach ALLOCATE steht der Referenzzähler auf 1. Sinkt er durch ein DEALLOCATE auf 0 ab, so wird der vom Objekt besetzte Speicherplatz freigegeben. DEREF dekrementiert den Referenzzähler und liefert als Resultat den neuen Wert zurück. DERF gibt ein Objekt jedoch nicht frei, die eigentliche Freigabe muss am Schluss durch DEALLOCATE erfolgen.

3. LOAD: Der Wert einer Speicherzelle kann unter Angabe der Objektidentifikation und des Indexes im Objekt gelesen werden[9].

4. STORE: Der Wert einer Speicherzelle kann unter Angabe der Objektidentifikation, des Indexes im Objekt und des Werts geschrieben werden[9].

Es gibt drei Klassen von Objekten: Lokale Objekte, verteilte Objekte und replizierte Objekte:

Der Speicher für *lokale Objekte* wird auf dem eigenen Prozessor alloziert. Lese- und Schreibzugriffe vom eigenen oder benachbarten Prozessoren sind schnell, und es wird nur wenig Speicher verbraucht. Der Nachteil dieser Objekte ist, dass die Bandbreite der Zugriffe durch den einen, verwaltenden Objekt-Manager beschränkt ist, was zu einem Systemengpass führt, falls viele Prozessoren gleichzeitig auf ein Objekt zugreifen wollen.

Verteilte Objekte werden gleichmässig verteilt auf allen Prozessoren alloziert. Aus dem Index eines Zugriffs lässt sich berechnen, auf welchem Prozessor das gewünschte Datenwort zu finden ist[10]. Bei diesen Objekten ist die Lokalität nicht auszunützen, da die meisten Zugriffe nicht auf dem eigenen Prozessor passieren. Dafür ist die Zugriffsbandbreite zum Schreiben und zum Lesen nur durch die Bandbreite des gesamten Netzwerkes beschränkt.

Bei *replizierten Objekten* wird eine Kopie des ganzen Objekts auf jedem Prozessor angelegt. Solche Objekte haben die grösste Lesebandbreite, benötigen aber zum Schreiben den speziellen Broadcast-Bus, was die Schreibzugriffe sequentialisiert. Ausserdem benötigen duplizierte Objekte viel mehr Speicher als Objekte der anderen Klassen.

---

9) Neben Valid-Bits haben die einzelnen Worte auch Present-Bits. Es gibt einen speziellen Schreibmodus für Einfachzuweisungssprachen, der einen Fehler liefert, falls eine Zelle überschrieben werden soll. Dies ist notwendig, da für Array-Zugriffe die Einfachzuweisungsregel nicht zur Kompilationszeit garantiert werden kann.

10) In der aktuellen Implementation sind die einzelnen Felder entsprechend den hintersten Stellen des Index auf die Prozessoren verteilt. "i MOD nrOfProc" ergibt also den Prozessor und "i DIV nrOfProc" den Index des gewünschten Felds auf diesem. Selbstverständlich sind auch andere Verteilungsstrategien möglich.

Die Figur 2.5 illustriert die Speicherbelegung durch die verschiedenen Objektklassen. In [MurFär92] werden die Bandbreiten und Speicherlatenzen für alle Objektklassen im Detail quantitativ analysiert.

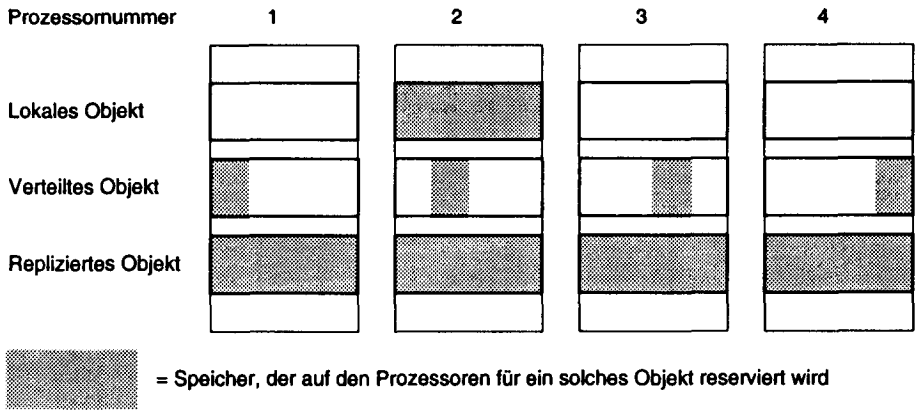


Fig. 2.5: Die verschiedenen Klassen von Objekten

Alle Operationen sind für jedes Objekt von jedem Prozessor aus transparent ausführbar. Trotz gleicher Funktionalität gibt es, abhängig von der verwendeten Objektklasse, bedeutende Unterschiede in der Rechenleistung. Die richtige Verwendung der Objektklassen ist ein wesentliches Problem bei der Codegenerierung für die ADAM-Architektur und wird in Kap. 6 "Effiziente Verwaltung von Datenstrukturen" eingehend besprochen.

Alle Objekt-Manager der Maschine sind untereinander durch ein Paket-Netzwerk verbunden, auf dem Befehle und Datenpakete für Zugriffe auf nichtlokale Objekte ausgetauscht werden. Zusätzlich gibt es eine effiziente Möglichkeit, auf duplizierte Objekte zu schreiben: Den *Broadcast-Bus*. Ein Prozessor kann auf ihn schreiben und alle anderen können mithören.

Alle Objekt-Manager-Befehle werden asynchron ausgeführt. Zusammen mit den Synchronisationsfähigkeiten des Register-Frames können die unterschiedlichen Latenzzeiten der nicht lokalen Speicherzugriffe versteckt werden[11].

---

11) Dieses Verstecken der Latenzzeiten ist eine zentrale Forderung an skalierbare Multiprozessoren, wie in [Arvind und Ianucci] schön erklärt wird.

2.6. Der Exekutor: Ein spezieller RISC-Prozessor

Alle Befehle, die nicht vom Objekt- oder vom Context-Manager ausgeführt werden, laufen auf dem Exekutor ab. Er unterstützt den üblichen arithmetisch-logischen Befehlssatz, inklusive Fliesskomma-Arithmetik. Es gibt Sprungbefehle für Schleifen. Es kann nur über die LOAD/STORE-Befehle des Objekt-Managers auf den Hauptspeicher zugegriffen werden. Alle anderen Befehle verwenden Operanden- und Zielregister aus dem Register-Frame. Es handelt sich demzufolge um eine typische LOAD/STORE-Architektur. Mittels Instruktionszähler steuert der Exekutor den gesamten Programmablauf.

Speziell ist, dass vor jeder Operation die Present-Bits der Operanden überprüft werden, was zum Abbruch einer Operation in der "Fetch-Data"-Phase führen kann. Ausserdem kann Pipelining wegen der vielen Kontextwechsel nicht effizient auf die übliche Art implementiert werden.

2.7. Der Simulator: Die verwendete Implementation der ADAM-Architektur

Alle in den späteren Kapiteln dieser Arbeit gezeigten Messungen wurden auf einem Simulator der oben beschriebenen Maschine durchgeführt. Die Maschine wird darin Zyklus für Zyklus auf der Ebene des Register-Transfers simuliert. Bei allen vorgenommenen Messungen wird die Zeit als Anzahl verstrichene Zyklen angegeben. Die Dauer eines solchen Zyklus ist von der verwendeten Technologie zur Implementation abhängig. Zudem gibt es für die einzelnen Baublöcke verschiedene Implementationsvarianten, die eine reale Maschine noch beschleunigen oder verlangsamen könnten. Die Zahlen geben aber sicher bis auf eine Grössenordnung verlässliche Vorhersagen über das Verhalten einer echten Maschine. Dies unterscheidet den verwendeten Simulator grundsätzlich von anderen Datenfluss-Simulatoren, die auf bedeutend höherer Ebene arbeiten.

In der aktuellen Implementation (vgl. Figur 2.6) besteht ein ADAM-Prozessor aus den vier selbständigen Unterprozessoren Objekt-Manager (OM), Sequencer und aritmetisch-logische Einheit (S und ALU), Context-Manager (CM) und Token-Manager (TM) die über zwei Busse miteinander kommunizieren. Ueber den C-Bus verschickt der Sequencer Anforderungen an seine Koprozessoren. Ueber eigene Caches (OC, SC, FC, CC und TC) und den D-Bus greifen die Prozessoren auf den lokalen Speicher des Prozessors (M) zu. Eingabe und Ausgabe geschehen über entsprechende Speicherstellen (I/O).

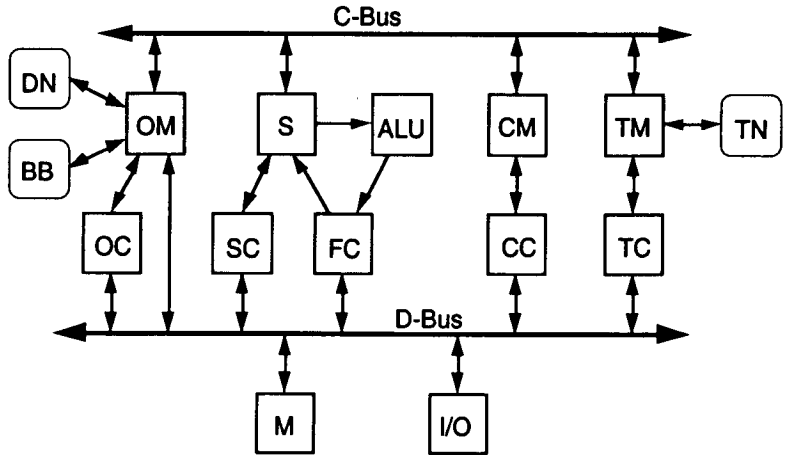


Fig. 2.6: Gesamtstruktur der ADAM-Architektur [Maquel92]

Der Objekt-Manager ist über ein Netzwerk (DN) und den Broadcast-Bus (BB) mit den Objekt-Managern auf den anderen Prozessoren verbunden. Alle Objekt-Manager, das Netzwerk, der Broadcast-Bus und die lokalen Speicher bilden gemeinsam die abstrakte Schicht eines physisch verteilten, aber logisch gemeinsamen Speichers. Die Architektur des Simulators ist nicht vollständig festgelegt. Verschiedene Parameter, wie Netzwerk-Topologie, verschiedene Latenzen, Kosten für einzelne Befehle, etc. können vom Benutzer gewählt werden. Wo dies für die Experimente eine Rolle spielt, wird direkt auf die Wahl der Parameter hingewiesen.

Der Simulator ist als interaktives Werkzeug auf dem MacIntosh-Arbeitsplatz-Computer implementiert. Es stehen dem Benutzer verschiedene Beobachtungsmöglichkeiten zur Verfügung. Um den Simulationskern zu beschleunigen, kann dessen Ausführung auf einen 32-Transputer-Multiprozessor ausgelagert werden. Es stehen somit Simulationsleistungen (ca. 10000 Zyklen pro Sekunde) zur Verfügung, die es erlauben, grössere Testprogramme auf sehr vielen (maximal 256) Prozessoren auszuführen. Die Details zur ADAM-Architektur und zum Simulator findet man in einer anderen Dissertation aus dem ADAM-Projekt.

2.8. Arbeiten mit ähnlichen Ansätzen

Die gezeigten Ideen werden nicht nur in der ADAM-Architektur verwendet. Im folgenden wird kurz eine Reihe von Computerarchitekturen hingewiesen, bei denen teilweise ähnliche Konzepte verwendet wurden. Selbstverständlich lassen sich unsere Ideen zur Codegenerierung auch auf diese Maschinen übertragen.

Schon der erste, kommerziell erhältliche Multiprozessor [HwaBri84], der Denelcor HEP (Heterogeneous Element Processor) unterstützte mehrere Codeblöcke, Datensynchronisation auf Registern und einen sehr schnellen Kontextwechsel in Hardware, um unterschiedliche Latenzen der parallelen, funktionalen Einheiten und des Speichers zu verstecken. Das grösste Problem bei dieser Maschine war, dass die Anzahl der Codeblöcke durch die Hardware auf eine tiefe Limite von 128 pro Prozessor beschränkt wurde. In der ADAM Architektur ist zwar die Anzahl der Codeblöcke, die im Frame-Cache Platz finden, ebenfalls beschränkt, jedoch können bei Bedarf Kontexte in den Hauptspeicher ausgelagert werden. Ausserdem werden Codeblöcke gar nicht expandiert, sondern in Tokenform aufbewahrt, solange alle Prozessoren genügend Arbeit haben.

Inzwischen haben mehr oder weniger alle Forschungsgruppen auf der Welt, die sich mit Datenflussarchitekturen beschäftigen, einsehen müssen, dass eine feingranulare Architektur nicht effizient implementierbar ist. Einen wichtigen Anstoss zu dieser Entwicklung hat sicher eine Arbeit gegeben, an der auch ein ehemaliges Mitglied unserer Forschungsgruppe beteiligt war [BühEka87]. Entsprechende Arbeiten wurden an allen Zentren für Datenflussforschung in Angriff genommen. Die frühen Arbeiten [GauErc84], [Babb84], [Ianucc88] waren eher theoretischer Natur. Dagegen wurden in der neuesten Zeit auch sehr konkrete Implementationsvorschläge für solche Maschinen vorgestellt [NiPaAr91]. Diese sind dem ADAM-Implementations-Konzept sehr ähnlich, denn es basiert ebenfalls auf den RISC-Mikroprozessoren der Reihe 88000 von Motorola [Motoro88].

Die ADAM-Architektur unterscheidet sich von den genannten Ansätzen hauptsächlich durch folgende Punkte:

- Der ADAM-Ansatz ist umfassend, was bedeutet, dass Architektur und Programmierwerkzeuge gemeinsam entwickelt wurden.

- In der ADAM-Architektur wird deutlich gezeigt, wie die Last wirklich auf mehrere Prozessoren verteilt werden kann [Maquel90]. In vielen Datenfluss-Projekten bleiben die Aussagen zu dieser Frage unklar.

- Im ADAM-Projekt ist nicht nur die Parallelisierung der Rechenarbeit, sondern auch die Parallelisierung der Daten ein Thema [MurFär92].

- Der Simulator der ADAM-Hardware implementiert ein hardwarenahes, realitätsbezogenes Modell.

3. MFL - Eine einfache, funktionale Sprache

Wie in Kapitel 1 "Einführung" erwähnt, wird im ADAM-Projekt Computerarchitektur nicht als reine Hardware-Aufgabe angesehen. Wir haben deshalb die einfache Programmiersprache MFL mit dem Ziel möglichst direkter Abbildbarkeit auf die ADAM-Zielmaschine entworfen. Dieses Kapitel enthält eine kurze Einführung in die Programmiersprache MFL mit Syntaxdefinitionen und kleinen Beispielen. Alle folgenden Kapitel dieser Arbeit widmen sich dem Problem der effizienten Uebersetzung von MFL für die ADAM-Architektur. Während in Kapitel 2 "Die ADAM-Architektur" die Zielmaschine für die in der vorliegenden Arbeit definierte Codegenerierung vorgestellt wird, erläutert dieses Kapitel die übersetzte Programmiersprache.

Ausserdem wird ein allgemeiner Ueberblick über die Klasse der Einfachzuweisungssprachen, deren Vorteile und Grenzen gegeben. Am Schluss des Kapitels werden zwei unterschiedliche Möglichkeiten diskutiert, die Grenzen solcher Programmiersprachen zu überwinden. Anschliessend wird jene Lösung skizziert, die wir für unser Projekt ausgewählt haben.

Eine Reihe grösserer MFL-Programme finden sich in Anhang B. Der Anhang A enthält die vollständige Syntax von MFL.

3.1. Einfachzuweisungssprachen

Mit *Einfachzuweisungssprachen* ist, wie der Name sagt, eine Klasse von Programmiersprachen gemeint, deren wesentlichste Eigenschaft ist, dass jede Variable im Verlaufe der Programmausführung nur einmal einen Wert zugewiesen bekommt. Der Name "Einfachzuweisungssprache" stammt aus einem französischen Projekt [Plas76], bei dem die Programmiersprache LAU ("Language à Assignement Unique") definiert wurde.

Ausserdem haben in Einfachzuweisungssprachen alle Kontrollkonstrukte (auch Schleifen) funktionalen Charakter. Sie sind Ausdrücke wie in der Mathematik, die auf Grund einer Reihe von definierten Variablen ein- oder mehrwertige Resulate liefern.

Durch diese zwei Einschränkungen gewinnt man eine Reihe von Vorteilen für die parallele Ausführung von Programmen:

1. Man gewinnt die *referentielle Transparenz* von Namen und Werten. Die Bedeutung dieses Begriffs liegt darin, dass ein Variablenname fest an einen Wert gebunden ist, also immer denselben Wert referenziert. Diese Bindung ist stärker als in gewöhnlichen Programmiersprachen, bei denen ein Name in der Regel an eine feste Speicherstelle gebunden ist und durch wiederholte Zuweisungen verschiedene Werte annehmen kann oder sogar Zeigervariablen zugelassen sind, die mit mehreren Namen auf die gleiche Speicherstelle zeigen. Konkret bedeutet dies, dass sich ein Programm in einer Einfachzuweisungssprache auf seine reinen Datenabhängigkeiten reduzieren lässt.

2. Die einzige Einschränkung für die *Ausführung* von Einfachzuweisungs-Programmen ist, dass jeder Wert berechnet und zugewiesen sein muss, bevor er gebraucht wird. Anders ausgedrückt kann man sagen, dass das ganze Parallelitätspotential, das in einem Algorithmus steckt, auch wirklich ausgenutzt werden kann. Der Programmierer kann durch seine Auswahl einer Instruktionssequenz die Parallelität eines Algorithmus nicht zerstören.

3. Es gibt in solchen Programmen keinen *globalen Zustand* und auch keine Möglichkeit von *Seiteneffekten*. Diese Eigenschaft ist natürlich für die parallele Ausführung sehr wünschenswert, hat aber auch positive Konsequenzen für die Softwaretechnik. Eine Funktion wird, mit den gleichen Eingabewerten gespiesen, immer die gleichen Ausgabewerte liefern.

4. Die Eigenschaft, mit gleichen Eingabewerten immer die gleichen Ausgabewerte zu liefern, erstreckt sich auch auf die parallele Ausführung auf einer beliebigen Anzahl von Prozessoren. Diese *Determiniertheit* kann bei der Programmierung ausgenutzt werden, indem man das Programm auf einem Prozessor sequentiell testet, von Fehlern befreit und erst dann parallel laufen lässt. Unter der Voraussetzung, dass alle zu Grunde liegenden Systeme (Compiler, Hardware, etc.) korrekt laufen, ist dies ein möglicher Lösungsansatz zum Problemkreis "Debugging paralleler Programme".

5. In solchen Sprachen lassen sich nur *totale Funktionen* schreiben. Mit Einfachzuweisungssprache ist nicht nur die Einschränkung gemeint, dass jeder Name nur einmal definiert werden darf, sondern auch, dass jeder Name einmal zugewiesen werden muss. Es gibt also keine Probleme mit uninitialisierten Variablen.

Neben diesen unbestritten positiven Eigenschaften von Einfachzuweisungssprachen, gibt es auch eine Reihe von Nachteilen. Auf diese Nachteile und mögliche Auswege kommen wir am Schluss des Kapitels noch ausführlich zurück.

Doch zuerst wird nun die im ADAM-Projekt verwendete Einfachzuweisungssprache MFL beschrieben. Der Name *MFL* [Wytten90] bedeutet "Modular Functional Language" und weist damit auf den syntaktischen Ursprung der Sprache, nämlich Modula-2 [Wirth85], und ihre wichtigsten Baublöcke, die Funktionen, hin.

3.2. Funktionen als Baublöcke

3.2.1. Einfache Funktionen

Die wichtigsten Bausteine eines MFL-Programms sind die Funktionen. Eine Funktion besteht immer aus einem *Funktionskopf*, der die Schnittstelle der Funktion gegen aussen definiert, und einem *Funktionskörper*, der die Rechenvorschrift für die Funktion enthält. Das folgende Beispiel zeigt eine Funktion, die zwei ganzzahlige Werte austauscht:

```
FUNCTION Xchg(a, b: INTEGER RETURNS INTEGER, INTEGER); (* Kopf *)
BEGIN                                                  (* Körper *)
RETURNS
  b, a
ENDFUNCTION Xchg;
```

An diesem Beispiel sieht man schon, dass die Syntax von MFL in der Tradition von Modula-2 mit vielen Schlüsselwörtern und wenigen Sonderzeichen gehalten ist. Die Sprache kennt ein strenges Typenkonzept, auf welches später noch im Detail eingegangen wird. Für dieses Beispiel genügt es zu sehen, dass die Funktion zwei ganze Zahlen (`INTEGER`) als Input erwartet und zwei ganze Zahlen als Resultate liefert. Im Gegensatz zu vielen imperativen Sprachen können MFL-Funktionen mehrwertige Resultate liefern. Die Eingabeparameter werden in gewohnter Weise als formale Parameter-Liste definiert. Der Typ des Resultats wird nach dem Schlüsselwort `RETURNS` als Liste von Typenbezeichnern spezifiziert.

Der Funktionskörper wird durch die Schlüsselwörter `BEGIN`, `RETURNS` und `ENDFUNCTION` gegliedert. Zwischen `BEGIN` und `RETURNS` können optional eine Reihe von Werten durch Ausdrücke definiert werden. Nach `RETURNS` folgt der Resultatausdruck für die Funktion.

Alle Funktionen haben einen Namen, der im Funktionskopf deklariert und am Schluss, nach `ENDFUNCTION`, wiederholt wird. Für solche *Bezeichner* gelten die

gleichen Regeln wie in Modula-2: Bezeichner dürfen aus einer beliebig langen Folge von Buchstaben und Zahlen ohne Sonderzeichen bestehen. Gross- und Kleinschreibung werden unterschieden.

3.2.2. Funktionen als Sichtbarkeitsbereiche

Wie üblich in blockstrukturierten Sprachen, haben die Funktionen auch die Bedeutung von *Sichtbarkeitsbereichen* für programmiersprachliche Objekte. Im folgenden Beispiel der Funktion `QuadEqu` zur Berechnung der beiden Lösungen der quadratischen Gleichung, sieht man, wie lokal zur Funktion `QuadEqu` die Funktion `Discriminant` und die *Variable*[1] `d` definiert werden. Solche Sichtbarkeitsbereiche können beliebig verschachtelt werden. Neben Variablen und Funktionen können auch Typen und Konstanten mit dem gleichen Sichtbarkeitsbereich definiert werden. Eine Besonderheit in MFL ist, dass alle Parameter einer Funktion explizit übergeben werden müssen und somit Variablen nur auf der Ebene der Definition, nicht aber in den verschachtelten Funktionen, sichtbar sind. Die Variable d wäre also in diesem Beispiel in der Funktion `QuadEqu`, nicht aber in `Discriminant` sichtbar.

```
FUNCTION QuadEqu(a, b, c: REAL RETURNS REAL, REAL);
VAR d: REAL;
  FUNCTION Discriminant(a, b, c: REAL RETURNS REAL);
  BEGIN
  RETURNS SQRT(b*b - 4.0*a*c)
  ENDFUNCTION Discriminant;
BEGIN
  d := Discriminant(a,b,c);
RETURNS
  (-b + d) / (2.0 * a), (-b - d) / (2.0 * a)
ENDFUNCTION QuadEqu;
```

Ganze MFL-Programme bestehen aus einer Reihe von Funktionen, die auf der obersten Ebene sichtbar sind. Solche Programme werden nicht selbständig aktiv. Es gibt keinen Code, der nach dem Laden automatisch ausgeführt wird. Sie ma-

---

[1]) In MFL werden benannte Werte als Variablen bezeichnet, obwohl sie keine Variablen im herkömmlichen Sinne der Programmiertechnik sind, da sie nur einmal zugewiesen werden dürfen. Der Grund dafür liegt darin, dass in den prozeduralen und in den funktionalen Teilen von FOOL (vgl. Abschnitt 3.6 "Grenzen von MFL und weiterführende Konzepte") soweit als möglich übereinstimmende Begriffe verwendet werden sollen.

chen nur Sinn im Zusammenhang mit einem grösseren Programmiersystem, von wo aus die einzelnen Funktionen aufgerufen werden können.

3.3. Deklarationen

Neben Funktionen und Variablen lassen sich in MFL noch eine Reihe von weiteren programmiersprachlichen Objekten definieren. Die oben besprochenen Regeln über Defintionsbereiche gelten für alle Objekte in diesem Abschnitt.

3.3.1. Basistypen

MFL unterstützt eine Reihe von *Basistypen*: In der heutigen Version sind dies die Typen `REAL`, für 32-Bit-Gleitkommazahlen, `INTEGER` für 32-Bit-Ganzzahlen, `CHAR` für 8-Bit-Zeichen und `BOOLEAN` für die beiden Wahrheitswerte `TRUE` und `FALSE`. Entsprechend dieser Basistypen ist die Schreibweise für *konstante Literale* definiert. In der jetzigen Version sind noch keine Grundtypen für Zeichenketten, Systemtypen (ADDRESS, etc.) und Mengen enthalten. Diese lassen sich ohne weiteres in eine definitive Version der Sprache einbauen.

Typ:	Regulärer Ausdruck für Literale: (d = beliebige Dezimalziffer, c = beliebiges Zeichen)	Beispiele:	Bemerkungen:
INTEGER	d{d}	134, 12, 0	Im Moment sind noch keine Möglichkeiten für die einfache Darstellung von Zahlen anderer Zahlensysteme (binär, oktal, hexadezimal) vorgesehen. Die Vorzeichen '+' und '-' sind nicht Teil der Literale. Sie erscheinen über die Syntax.
REAL	d{d}'.' d {d} | d{d}'E'{'+'|'-'}d{d}	2.0, 0.0, 123455.234E-23	Im Moment sind noch keine REALS mit höherer Genauigkeit vorgesehen, obwohl dies für eine praktisch verwendbare Sprache notwendig wäre.
CHAR	'"' c '"'	"a", "X"	Es gibt noch keine einfache Möglichkeit Sonderzeichen zu definieren.
BOOLEAN	'TRUE' | 'FALSE'	TRUE, FALSE	

Tabelle 3.1: Grundtypen in MFL

3.3.2. Konstanten

Analog zu Modula-2 erlaubt MFL die Definition von Konstanten. Auf der linken Seite einer *Konstantendefinition* steht immer ein Bezeichner und auf der rechten Seite ein beliebiger, skalarer Ausdruck. Es gilt die Bedingung, dass alle im Definitionsausdruck verwendeten Werte entweder konstante Literale oder bereits definierte Konstanten sind. Der Typ einer Konstanten wird durch den Typ des Definitionsausdrucks bestimmt.

Beispiel:

```
CONST  pi = 3.1415;      firstC = "a";  minInt = -32767;
       debug = TRUE;     rangeCheck = debug;
       piSquare = pi * pi;
```

Der Unterschied zwischen Konstanten und Variablen in MFL ist nicht so gross wie bei konventionellen Programmiersprachen, da jede Variable nur einmal zugewiesen werden darf und danach als Konstante betrachtet werden kann. Eigentlich be-

steht der Unterschied nur darin, dass Variablen dynamisch (zur Laufzeit) und Konstanten statisch (zur Uebersetzungszeit) instanziert werden.

Strukturierte Konstanten sind in der aktuellen Version von MFL noch nicht vorgesehen. Als Ausdrücke zur Definition von Konstanten dürfen beliebige, einfache Ausdrücke verwendet werden, bei denen alle Ausgangswerte konstant sind. Komplexe Ausdrücke (vgl. Abschnitt "Komplexe Ausdrücke") sind für Konstantendefinitionen nicht vorgesehen.

3.3.3. Tupel

Es gibt in MFL auch die Möglichkeit, Verbundtypen zu deklarieren. Wir gebrauchen in diesem Zusammenhang das Schlüsselwort TUPLE, da wir den Verbundtyp eher mit dem mathematischen Konzept eines Tupels als mit dem Konzept eines Datensatzes auf einem Sekundärspeicher assoziieren.

Ein Tupel für die Verwaltung von kompexen Zahlen würde folgendermassen definiert:

```
TYPE Complex =   TUPLE
                   re, im : REAL;
                 ENDTUPLE;
```

Verschachtelte Tupel können in MFL nicht direkt deklariert werden, weil MFL nur die Typenkompatibilität über Namen, nicht aber die strukturelle Typenkompatibilität kennt. Man kann demzufolge keine anonymen Typen verwenden, wie dies in Pascal oder Modula-2 möglich ist. Ein Tupel für ein Paar von komplexen Zahlen macht den Unterschied deutlich:

```
(* Korrekte Version *)
TYPE Complex     = TUPLE
                     re, im : REAL;
                   ENDTUPLE;
     ComplexPair = TUPLE
                     first, second: Complex;
                   ENDTUPLE;
(* Falsche Version *)
TYPE ComplexPair = TUPLE
                     first: TUPLE
                                re, im : REAL;
                            ENDTUPLE;
                     sec:   TUPLE
                                re, im : REAL;
                            ENDTUPLE;
                   ENDTUPLE;
```

Ein weiterer Unterschied zu gewohnten Sprachen ist, das man keine Varianten-Records kennt. Dieses Konzept ist auf der Ebene der objekt-orientierten Erweiterung von MFL (vgl. Teil 3.6 "Grenzen von MFL und weiterführende Ideen") durch die Möglichkeit von Vererbung bei Objekttypen ersetzt und verbessert. Varianten-Records sind bekanntlich ein ziemlich problematisches Konzept für die Typenüberprüfung, werden sie doch häufig dazu verwendet die strengen Typenregeln in Pascal zu umgehen.

3.3.4. Arrays

Arrays definiert man in MFL folgendermassen:

```
CONST n = 25;
TYPE
     Vector = ARRAY [1..n] OF REAL ENDARRAY;
```

Mehrdimensionale Arrays lassen sich in der heutigen Version von MFL auf zwei Arten deklarieren:

1. Man deklariert direkt einen *mehrdimensionalen Array*, der auf der Maschine in ein einzelnes Objekt (vgl. Kapitel 2 "Die ADAM-Architektur") abgebildet wird. Der Vorteil dieser Art von Array-Definition ist, dass die einzelnen Elemente direkt über einstufige Zugriffe erreicht werden können. Dem steht der Nachteil gegenüber, dass Teile des Arrays nicht als ganzes zugewiesen werden können, da sie keinen expliziten Typ haben und somit nicht zuweisungskompatibel zu einem anderen Typ sein können. Eine derart deklarierte Matrix und Zugriffe auf ihre Elemente sehen folgendermassen aus:

```
TYPE
    Matrix = ARRAY [1..n],[1..n] OF Vector ENDARRAY;

... a[i, j] ... (* erlaubter Designator *)
... a[i] ...    (* unmöglicher Designator *)
```

2. Die andere Möglichkeit ist, einen mehrdimensionalen Array über eindimensionale Arrays, deren Elemente selbst Arrays sind, zu deklarieren. In diesem Fall wird jede Stufe auf der Maschine durch eigene Objekte realisiert, falls die Unterfelder nicht expandiert werden (vgl. Abschnitt 6.3 "Objekte mit Unterobjekten"). Auf die atomaren Elemente kann deshalb nur noch über mehrstufige Ladeoperationen zugegriffen werden, wobei auch die Unterstrukturen einen eigenen Typ haben und somit als ganzes zugreifbar werden. Die entsprechende Matrix sieht etwas anders aus:

```
TYPE
    Vector = ARRAY [1..n] OF REAL ENDARRAY;
    Matrix = ARRAY [1..n] OF Vector ENDARRAY;
... a[i] ... (* Gültiger Zugriff auf eine Zeile *)
... a[i][j]  (* Zugriff auf ein einzelnes Element *)
```

Nebenbei zeigt das Beispiel auch, dass definierte Konstanten nicht nur im operationellen Teil des Programms, sondern auch in Typdeklarationen verwendet werden können.

Arrays sind in MFL mit Absicht sehr primitiv gehalten. Für eine Produktionssprache wäre es sicherlich wünschenswert, dynamische Arrays oder zumindest offene Arrays als Funktionsparameter im Sinne von Modula-2 vorzusehen. Die Kosten und Techniken für solche unbestritten komfortablen Erweiterungen der Sprache sind jedoch bekannt und es gibt keinen Grund, anzunehmen, dass man für funk-

tionale Sprachen grundlegend andere Techniken benötigen würde. Aus Gründen der Einfachheit wurde deshalb vorläufig auf derartige Möglichkeiten verzichtet.

3.3.5. Aufzählungs- und Unterbereichstypen

Aufzählungstypen können in MFL wie folgt definiert werden:

```
TYPE Farbe = (rot, gruen, blau, gelb, weiss);
     BOOLEAN = (TRUE, FALSE);
```

Für Aufzählungstypen sind nur die Operatoren ":=", "=" und "<>" definiert. Bei Vergleichen und Zuweisungen sind sie nur gegenüber Werten vom gleichen Aufzählungstyp typenverträglich. Eine Ausnahme ist der eingebaute Aufzählungstyp BOOLEAN mit den üblichen Operatoren.

Unterbereichstypen definieren durch Angabe eines maximalen und eines minimalen Elements einen Unterbereich des Typs INTEGER:

```
TYPE TagImMonat = [1..31]
```

Vorläufig sind noch keine Unterbereiche über Aufzählungstypen zugelassen. Alle Unterbereichstypen sind untereinander und mit dem vordefinierten Typ INTEGER zuweisungs- und operationskompatibel. Durch das Laufzeitsystem wird sichergestellt, dass einer Variablen eines Unterbereichstyps nur gültige Werte zugewiesen werden können.

3.4. Einfache Ausdrücke und Wertzuweisungen

Jeder Funktionskörper enthält eine Reihe von Wertzuweisungen (StatementSeed) und einen Resultatausdruck. Wie üblich wird eine Variable auf der linken Seite über den Operator ":=" durch einen Ausdruck auf der rechten Seite zugewiesen. Jede Zuweisung wird durch ein Semikolon abgeschlossen[2].

Wertzuweisungen haben lokale Bedeutung, zugewiesene Werte gelten nur für den direkt zugehörigen Resultatausdruck.

---

[2]) Dies ist ein Unterschied zu PASCAL und MODULA-2, bei denen der Strichpunkt zwei Anweisungen voneinander separiert.

3.4.1. Mehrfache Zuweisungen

Im Unterschied zu den meisten imperativen Sprachen kennt MFL mehrfache Zuweisungen. Eine Wertzuweisung (Assignment) besteht dabei aus einer Reihe von Variablennamen auf der linken Seite und einer Reihe von Ausdrücken auf der rechten Seite. Die Anzahl der Variablen und der Ausdrücke muss nicht übereinstimmen, da in MFL mehrwertige Ausdrücke möglich sind. Eine typische Anwendung mehrfacher Ausdrücke ist die Zuweisung von Werten durch eine mehrwertige Funktion, wie im folgenden Beispiel gezeigt:

```
FUNCTION Gugus(x, y: INTEGER RETURNS INTEGER, INTEGER);
.....
x, y := Gugus(a, b);
```

Man kann aber von der gleichen Möglichkeit Gebrauch machen, um einer Reihe von Variablen einfache Ausdrücke (z. B. Konstanten) zuzuweisen.

```
i, j, x := 0, 1, 0.0;
```

Eine weitere typische Anwendung dieser mehrwertigen Zuweisungen werden wir im Zusammenhang mit *komplexen Ausdrücken* kennenlernen.

3.4.2. Regeln für die Einfachzuweisung

Da MFL eine *Einfachzuweisungssprache* ist, spielen die Regeln für die Zuweisung von Variablen eine zentrale Rolle. Die Regeln sind:

1. Grundsätzlich darf jede Variable innerhalb ihres Definitionsbereiches nur einmal zugewiesen werden.

Diese Regel ist für eine *skalare Variable* sehr einfach zu überprüfen: Ein Variablenname darf nur einmal auf der linken Seite einer Zuweisung erscheinen. Schwieriger wird das Ganze für *strukturierte Variablen*. Es gilt dort die folgende Regel:

2. Alle Felder einer strukturierten Variable müssen innerhalb der gleichen Reihe von Wertzuweisungen (StatementSequence) zugewiesen werden.

Im Anschluss an diese Wertzuweisungen gilt die Variable als vollständig zugewiesen und kann in einem Resultatausdruck verwendet werden. Für einzelne *Array-Felder* können zur Uebersetzungszeit keine Einfachzuweisungs-Ueberprüfungen gemacht werden, da die Indices dynamisch berechnet werden. In der ADAM-Architektur wird zur *Laufzeit* überprüft, welche Felder eines Arrays schon zuge-

wiesen sind, und ein entsprechender Fehler signalisiert, falls mehrfach das gleiche Feld zugewiesen wird.

Es liessen sich durch symbolische Analyse für einfache Index-Funktionen schon zur Uebersetzungszeit Zuweisungskonflikte erkennen. Die entsprechenden Techniken sind schon lange aus dem Gebiet der vektorisierenden FORTRAN-Compiler bekannt [PadWol86]. Da die einschlägigen Techniken einerseits bekannt, andererseits aber aufwendig zu implementieren sind und ausserdem nicht zum zentralen Gebiet unserer Forschungsinteressen gehören, wurde auf eine solche Analyse im MFL-Compiler verzichtet.

Für *Tupel* kann die Einfachzuweisungsregel zur Uebersetzungszeit mit den folgenden Regeln überprüft werden.

2.1 Unterfelder zugewiesener Tupelfelder gelten als zugewiesen.

2.2 Tupelfelder, von denen einzelne Unterfelder zugewiesen sind, gelten als zugewiesen und können nicht mehr als Ganzes zugewiesen werden.

2.3 Felder ohne gegenseitige Unterfeld-Beziehung in der Feldhierarchie eines Tupels können unabhängig voneinander zugewiesen werden.

Das folgende Beispiel sollte den Gebrauch dieser Regeln verdeutlichen:

```
TYPE DoubleMatrix =  TUPLE
                       a : Matrix;
                       b : Matrix;
                     ENDTUPLE;
     Complex =       TUPLE
                       x : CARDINAL;
                       y : DoubleMatrix
                     ENDTUPLE;
VAR a: Complex;

a.x, a.y := ....      ok nach Regel 2.3
a.y, a.y.b := ....    Fehler nach Regel 2.1
a.x, a := ....        Fehler nach Regel 2.2
```

Diese Regeln sind stark genug, um eine Mehrfachzuweisung, ausser im Falle der Arrays, sicher zu verhindern. Dagegen sind die Regeln in gewissen, pathologischen Fällen strenger als notwendig. Man kann sich zum Beispiel den Fall vorstellen, dass ein Tupel als Ganzes durch ein unvollständig zugewiesenes Tupel definiert

wird. In diesem Fall kann der undefinierte Teil des Tupels nachträglich nicht mehr zugewiesen werden, ohne gegen die Einfachzuweisungsregeln zu verstossen, obwohl er noch nicht zugewiesen ist. Das folgende Beispiel zeigt das Problem:

```
a.x := ...
b := a
b.y := ...          Fehler, obwohl a.y und somit b.y nie zugewiesen wurden
```

Dieses Problem liesse sich nur lösen, wenn auch für die Tupel von der statischen Ueberprüfung der Einfachzuweisungsregeln abgewichen würde und ähnlich wie bei den Arrays die Regeln zur Laufzeit überprüft würden. Wir haben uns aber gegen diese Lösung entschieden, weil uns der mögliche Gewinn an Flexibilität weniger wertvoll erschien, als die Aussage, dass wir auch für Tupel mehrfache Zuweisung schon zur Uebersetzungszeit wirkungsvoll verhindern können. Zusätzliche Ueberprüfungen zur Laufzeit sind nämlich nicht kostenlos zu haben.

3.4.3. Einfache Ausdrücke und Operatoren in MFL

MFL kennt in *einfachen, arithmetischen Ausdrücken* vier Klassen von Operatoren mit nach unten zunehmender Bindung an die Operanden:

1. Die *Vergleichsoperatoren* (RelOp)
2. Die *Additionsoperatoren* (AddOp)
3. Die *Multiplikationsoperatoren* (MulOp)
4. Die *Vorzeichenoperatoren* (Sign)

In der Tabelle 3.2 sind in der ersten Spalte die Klasse des Operators, in der zweiten und dritten Spalte der Operator und seine Arität und in den letzten vier Spalten der Resultattyp des Operators mit Operanden der vier Basistypen angegeben. Nicht alle Operatoren sind für alle Basistypen definiert. Undefinierte Kombinationen sind mit *** markiert. Die Semantik der einzelnen Operatoren entspricht den üblichen Gepflogenheiten bei PASCAL-ähnlichen Sprachen.

Operator -Klasse	Operator	Arität	Operanden: INTEGER	REAL	CHAR	BOOLEAN
RelOp	"="	2	BOOLEAN	BOOLEAN	BOOLEAN	BOOLEAN
	"<>", "#"	2	BOOLEAN	BOOLEAN	BOOLEAN	BOOLEAN
	"<"	2	BOOLEAN	BOOLEAN	BOOLEAN	BOOLEAN
	">"	2	BOOLEAN	BOOLEAN	BOOLEAN	BOOLEAN
	"<="	2	BOOLEAN	BOOLEAN	BOOLEAN	BOOLEAN
	">="	2	BOOLEAN	BOOLEAN	BOOLEAN	BOOLEAN
AddOp	"+"	2	INTEGER	REAL	***	***
	"-"	2	INTEGER	REAL	***	***
	"OR"	2	***	***	***	BOOLEAN
MulOp	"*"	2	INTEGER	REAL	***	***
	"/", "DIV"	2	INTEGER	REAL	***	***
	"MOD"	2	INTEGER	***	***	***
	"^"	2	INTEGER	REAL	***	***
	"AND", "&"	2	***	***	***	BOOLEAN
Sign	"-"	1	INTEGER	REAL	***	***
	"+"	1	INTEGER	REAL	***	***
	"NOT"	1	***	***	***	BOOLEAN

Tabelle 3.2: Operatoren in einfachen MFL-Ausdrücken

Neben den gezeigten Operatoren kann jede *Funktion*, die nur ein skalares Resultat liefert als *Designator* in einem einfachen Ausdruck verwendet werden. Funktionen, die mehrere Resultate oder ein Resultat mit komplexem Typ (ARRAY oder TUPLE) liefern, dürfen nur direkt rechts der Zuweisung stehen, da für komplexe Typen keine Operationen ausser der Zuweisung definiert sind.

Eine Reihe von Funktionen ist in der Sprache MFL bereits vordefiniert. Die Auswahl dieser Funktionen steht im direkten Zusammenhang mit der ADAM-Architektur: Jede Funktion lässt sich direkt als Instruktion auf der ADAM-Maschine ausführen. Die Funktionen `MIN`, `MAX` und `ABS` sind für jeden skalaren Typ ausser REAL definiert.

```
FUNCTION MIN(x: INTEGER RETURNS INTEGER);
FUNCTION MINR(x: REAL RETURNS REAL);
```

```
FUNCTION MAX(x: INTEGER RETURNS INTEGER);
FUNCTION MAXR(x: REAL RETURNS REAL);
FUNCTION ABS(x: INTEGER RETURNS INTEGER);
FUNCTION ABSR(x: REAL RETURNS INTEGER);
FUNCTION SIN(x: REAL RETURNS REAL);
FUNCTION COS(x: REAL RETURNS REAL);
FUNCTION TAN(x: REAL RETURNS REAL);
FUNCTION ASIN(x: REAL RETURNS REAL);
FUNCTION ACOS(x: REAL RETURNS REAL);
FUNCTION ATAN(x: REAL RETURNS REAL);
FUNCTION SINH(x: REAL RETURNS REAL);
FUNCTION COSH(x: REAL RETURNS REAL);
FUNCTION TANH(x: REAL RETURNS REAL);
FUNCTION ATANH(x: REAL RETURNS REAL);
FUNCTION SQRT(x: REAL RETURNS REAL);
FUNCTION EXP(x: REAL RETURNS REAL);
FUNCTION LOG(x: REAL RETURNS REAL);
FUNCTION TRUNC(x: REAL RETURNS INTEGER);
FUNCTION ROUND(x: REAL RETURNS INTEGER);
FUNCTION FLOAT(x: REAL RETURNS INTEGER);
```

Die Semantik der Funktionen ist auf Grund der Namen klar. Zusätzliche Details findet man in der Beschreibung der ADAM-Architektur [Maquel92].

3.4.4. Typenkompatibilität

In einer Sprache mit strengen Typen stellt sich die Frage nach *Zuweisungs-* und *Operationskompatibilität* der verschiedenen Typen. Grundsätzlich wendet MFL dabei das Prinzip der Typenkompatibilität über Namen an:

1. Gleiche Typen (d. h. gleich über den Namen) sind zueinander zuweisungs- und operationskompatibel.

2. Unterbereichstypen sind mit allen anderen Unterbereichstypen und dem Basistyp INTEGER zuweisungs- und operationskompatibel.

3. Aufzählungstypen sind nur zu sich selbst zuweisungskompatibel. Als Operationen sind für sie nur die Vergleiche "=" und "<>" zugelassen.

Ueber die entsprechenden Standardfunktionen lassen sich Werte der Basistypen INTEGER und REAL ineinander konvertieren.

3.5. Komplexe Ausdrücke

Aus konventionellen Sprachen bekannte Konstrukte, wie Schleifen und Alternativen, erscheinen in MFL als *komplexe Ausdrücke*. Je nach Art des komplexen Ausdruckes besteht dieser aus mehreren, verschiedenartigen Substrukturen. Allen komplexen Ausdrücken gemeinsam ist jedoch, dass sie über mindestens einen *Resultatausdruck* verfügen. Aehnlich den Funktionen darf dieser Resultatausdruck *mehrwertig* sein. Komplexe Ausdrücke dürfen deshalb nur direkt in einer Wertezuweisung oder dem Resultatausdruck eines übergeordneten komplexen Ausdrucks bzw. einer Funktion erscheinen. Alle Resultate eines komplexen Ausdrucks werden über den Resultatausdruck zurückgegeben. Somit wird erreicht, dass auch die komplexen Ausdrücke, entsprechend den Funktionen, *seiteneffektfrei* sind.

3.5.1. Bedingter Ausdruck

Bei der Auswertung eines *bedingten Ausdruckes* wird zuerst die erste Bedingung (nach dem IF) ausgewertet. Ist diese erfüllt, so wird der entsprechende Resultatausdruck (nach dem THEN) ausgewertet und die entsprechenden Resultate werden zurückgegeben. Trifft die Bedingung nicht zu, so wird die erste ELSIF-Bedingung berechnet und entsprechend dem Resultat ein Resultatausdruck zurückgegeben. Trifft keine der Bedingungen zu, so wird der letzte Resultatausdruck (nach dem ELSE) zum Resultat. Der *Typ* des ganzen bedingten Ausdruckes entspricht dem Typ einer seiner Resultatausdrücke, die deshalb alle denselben Typ haben müssen.

Das Beispiel der Ackermann-Funktion zeigt die praktische Anwendung eines bedingten Ausdruckes:

```
FUNCTION A (x,y : INTEGER RETURNS INTEGER);
BEGIN
  RETURNS
    IF x = 0 THEN
      RETURNS y + 1
    ELSIF (y = 0) THEN
      RETURNS A(x-1, 1)
    ELSE
      RETURNS A(x-1, A(x, y-1))
    ENDIF
ENDFUNCTION A;
```

Alle *Einfachzuweisungsregeln* sind für den ganzen Sichtbarkeitsbereich einer Variable gültig. Im folgenden Beispiel könnte i ohne weiteres in beiden Zweigen des bedingten Ausdrucks verwendet werden, was aber von den strengen Regeln

verboten wird. Ebenso wäre es problemlos möglich, die gleiche Variable nicht nur einmal pro Funktion, sondern einmal pro Reihe von Wertzuweisungen (Statement-Seed) neu zu definieren, da wir wissen, dass Wertzuweisungen im Innern der seiteneffektfreien, komplexen Ausdrücke nur lokale Bedeutung haben. Im folgenden Beispiel wird "i" einmal ausserhalb des komplexen Ausdruckes und einmal pro Reihe von Anweisungen im bedingten Ausdruck zugewiesen. Dies hätte zur Folge, dass jede Reihe von Anweisungen, gemeinsam mit ihrem Resultatausdruck ein eigener Gültigkeitsbereich für jede Variable der Funktion wäre. Die Lösung hätte den Vorteil, dass man in grösseren MFL-Programmen Variablen-Namen sparen könnte. Wir haben uns jedoch gegen diese Lösung entschieden, da dadurch das Prinzip der einfachen Zuweisung auf dem Sichtbarkeitsbereich verletzt gewesen wäre.

```
...
i := IF b=2 THEN
       i := b;
       RETURNS i*i
     ELSE
       i := c;
       RETURNS i+i
     ENDIF;
...
```

In bedingten MFL-Ausdrücken ist der ELSE-Zweig obligatorisch, also nicht optional wie in den entsprechenden Anweisungen der konventionellen Programmiersprachen. Der Resultatausdruck hat somit immer einen definierten Wert. Als Konsequenz davon sind MFL-Funktionen immer *total*. Auch bei den anderen komplexen Ausdrücken wurde darauf geachtet, dass diese keine undefinierten Ergebnisse liefern können.

Der bedingte Ausdruck entspricht im wesentlichen dem *McCarthy-Schema* [EngLäu88] aus der theoretischen Informatik. Dieses Schema gemeinsam mit der Rekursion erlaubt schon die Darstellung aller partiell rekursiven und somit berechenbaren Funktionen. MFL wäre allein mit dem bedingten Ausdruck schon eine, im Sinne der theoretischen Informatik, mächtige Sprache. Tatsächlich gibt es funktionale Sprachen, die über keine anderen Kontrollstrukturen verfügen. In MFL sollten es aber sowohl der Programmierer, als auch der Compiler einfacher haben, effiziente Programme zu schreiben bzw. zu erzeugen. Deshalb ist zusätzlich eine Reihe von iterativen komplexen Ausdrücken vorgesehen.

3.5.2. Iterative Ausdrücke

Im Gegensatz zu vielen rein funktionalen Sprachen, aber ganz in der Tradition von anderen Einfachzuweisungssprachen, sieht MFL auch iterative Konstrukte vor. Zu jeder Iteration gehören in MFL eine Menge von *Iterationsvariablen*. Bei jedem Schleifendurchlauf wird aus der aktuellen Generation der Iterationsvariablen und den schleifeninvarianten Werten eine neue Generation von Schleifenvariablen berechnet. Die neue Generation von Iterationsvariablen unterscheidet sich von der aktuellen syntaktisch dadurch, dass dem Variablennamen das Schlüsselwort "NEW" vorangestellt wird.

Wenn man so will, kann man die Iterationsvariablen als begrenzten und kontrollierten Verstoss gegen das Einfachzuweisungsprinzip in MFL betrachten. So betrachtet, entsprechen die Iterationsvariablen dem variablen Zustand der Schleife. Auf der Ebene der statischen Semantik ist das Prinzip jedoch nicht verletzt, wenn man "NEW x" und "x" als zwei unterschiedliche Namen betrachtet.

Es gibt auch eine Betrachtungsweise, die das Einfachzuweisungsprinzip auch dynamisch aufrecht erhält, indem man annimmt, dass jeder Durchlauf der Schleife einer neuen Instanz von Schleifenvariablen entspricht und dass über den Variablennamen mit beziehungsweise ohne vorangestelltes "NEW" auf die nächste beziehungsweise die aktuelle Instanz einer Iterationsvariable zugegriffen werden kann. In dieser Betrachtungsweise gleicht der ganze iterative Ausdruck einer rekursiven Funktion, wobei die Iterationsvariablen den lokalen Variablen der Funktion entsprechen.

Iterative, komplexe Ausdrücke bestehen in MFL aus drei wesentlichen Teilen:

1. Im *Initialisierungsteil* bekommen alle Iterationsvariablen ihre Startwerte.

2. Im *Iterationsteil* oder *Schleifenkörper* (loop body) wird eine alte Generation von Iterationsvariablen in eine neue Generation transformiert.

3. Im *Resultatausdruck* wird aus der letzten Generation von Iterationsvariablen und anderen Werten das Resultat des ganzen iterativen Ausdrucks gebildet und zurückgegeben.

Die verschiedenen iterativen Ausdrücke in MFL unterscheiden sich durch die Art der Schleifensteuerung. Es gibt die *Iteration mit Abbruchbedingung* (LOOP), wobei die Abbruchbedingung (EXIT) jeweils direkt nach oder direkt vor dem

Schleifenkörper stehen kann und entsprechend vor oder nach der Ausführung des Schleifenkörpers überprüft wird. Syntaktisch wäre die Abbruchbedingung an einer beliebigen Stelle im Schleifenkörper möglich. Diese Allgemeinheit ist für die sequentielle Erweiterung von MFL vorgesehen (vgl. Abschnitt 3.6 "Grenzen von MFL und weiterführende Ideen"). Das folgende Beispiel zeigt das MFL-Programm für die iterative Annäherung an die Quadratwurzel nach der Newton-Methode, welches im wesentlichen aus einer Iteration mit Test der Abbruchbedingung nach dem Schleifenkörper besteht:

```
FUNCTION Sqrt(x, eps: REAL RETURNS REAL);
VAR sqrt: REAL;
BEGIN
RETURNS
  LOOP
    sqrt := x / 2.0;
  DO
    NEW sqrt := (x / sqrt + sqrt) / 2.0;
    EXIT (sqrt * sqrt - x) < eps;
  RETURNS sqrt
  ENDLOOP
ENDFUNCTION Sqrt;
```

Neben der Iteration mit Abbruchbedingung gibt es auch einen iterativen Ausdruck mit vorausberechneter Anzahl Schleifendurchläufe (FOR). Im Schleifenkörper kann die *Laufvariable* als spezielle Iterationsvariable verwendet werden, die nach jedem Schleifendurchlauf automatisch um Eins erhöht wird. Die Laufvariable muss wie andere Variablen deklariert werden und immer vom Typ INTEGER oder einem Unterbereichstyp sein.

Im nächsten Beispiel wird mit Hilfe des FOR-Ausdrucks ein Array mit Zufallszahlen gefüllt:

```
CONST n = 512;
TYPE ArrayToSort = ARRAY [1..n] OF INTEGER ENDARRAY;

FUNCTION NewSample(RETURNS ArrayToSort);

  VAR a:          ArrayToSort;
      i:          INTEGER;
      lastRandom: INTEGER;

  FUNCTION Random(lastRandom: INTEGER RETURNS INTEGER);
  BEGIN
    RETURNS ((lastRandom * 521) + 17) MOD 256
  ENDFUNCTION Random;

BEGIN
RETURNS
  FOR i := [1..n] INIT
    lastRandom := 13;
  DO
    NEW lastRandom := Random(lastRandom);
    a[i] := NEW lastRandom;
  RETURNS a
  ENDFOR
ENDFUNCTION NewSample;
```

Das obige Beispiel zeigt, dass im Schleifenkörper nicht nur Zuweisungen zu Iterationsvariablen, sondern auch Zuweisungen zu anderen Variablen möglich sind, falls das Einfachzuweisungsprinzip eingehalten wird. Falls in einem Schleifenkörper ein *Array elementweise zugewiesen* wird, dieses aber keine Schleifenvariable ist (keine Zuweisung mit NEW), so erfolgen alle Zuweisungen zum gleichen Array. Alle anderen, im Schleifenkörper zugewiesenen Variablen haben nur eine lokale Bedeutung, sind also nicht über die Grenzen eines Schleifendurchlaufs hinaus definiert.

3.5.3. Paralleler Ausdruck

Aehnlich den iterativen Ausdrücken gibt es auch den *parallelen Ausdruck*. Beim parallelen Ausdruck ist wie beim FOR-Ausdruck die Anzahl der Wiederholungen des Schleifenkörpers im voraus bestimmt. Anders als beim FOR-Ausdruck werden

die einzelnen Schleifendurchläufe in einer nicht definierten Reihenfolge, möglicherweise auch parallel, durchlaufen. Es dürfen also keine Datenabhängigkeiten unter den Schleifendurchläufen bestehen. Solche Abhängigkeiten entsprechen genau den Iterationsvariablen, welche somit in parallelen Ausdrücken verboten sind. Mit den Iterationsvariablen fällt auch die Notwendigkeit eines Initialisierungsteils für diese weg.

Die einzige Möglichkeit, im Schleifenkörper festzustellen, welches die gerade bearbeitete Iteration ist, bieten die *Laufvariablen*. Im Gegensatz zum FOR-Ausdruck dürfen dafür mehrere Laufvariablen über das Schlüsselwort "CROSS" zu einem komplexeren Index-Ausdruck verknüpft werden. Insgesamt wird dann das kartesische Produkt der Laufbereiche aller Laufvariablen durchlaufen.

Am folgenden Beispiel der Matrix-Multiplikation lässt sich der Unterschied zwischen dem parallelen und dem iterativen Ausdruck schön zeigen:

```
CONST n = 25;

TYPE Range = [1..n];
     Matrix = ARRAY MRange, MRange OF REAL ENDARRAY;

FUNCTION MatMul(a, b: Matrix RETURNS Matrix);
VAR i,j,k: MRange;
    sum: REAL;
    c: Matrix;
BEGIN
  RETURNS
    FORALL i:= [1..n] CROSS j:= [1..n] DO
      c[i,j] :=
        FOR k := [1..n] INIT
          sum := 0.0;
        DO
          NEW sum := sum + a[i,k] * b[k,j];
        RETURNS sum
        ENDFOR;
      RETURNS c
    ENDFORALL
ENDFUNCTION MatMul;
```

Parallele Ausdrücke sind die wichtigste Quelle von Parallelität in einem MFL-Programm und es ist eine entscheidende Frage bei der Uebersetzung, in welchem

Masse sie parallel oder sequentiell ausgeführt werden sollen (vgl. Kapitel 5 "Partitionierung von Datenflussgraphen in Codeblöcke").

3.6. Grenzen von MFL und weiterführende Ideen

3.6.1. Grenzen von Einfachzuweisungssprachen

Der Vorteil der impliziten Parallelität wird bei einer Einfachzuweisungssprache durch eine *Beschränkung der Ausdruckskraft* erreicht. Nicht für alle Probleme lassen sich sinnvolle Programme in einer Einfachzuweisungssprache schreiben. Generell kann man sagen, dass sich die meisten "batch"-artigen Algorithmen, bei denen alle Eingabedaten vor dem Start vorliegen, und von denen die Resultate erst nach Abschluss des ganzen Programms benötigt werden, auf natürliche Weise in einer funktionalen Sprache ausgedrückt werden können. Solche Programme haben funktionalen Charakter. Es gibt nur zwei bedeutende Ereignisse beim Programmablauf, den Start- und den Endzeitpunkt. In der Praxis kommt diese Klasse von Algorithmen recht häufig vor. Gerade numerische Algorithmen, bei denen eine implizite Parallelisierung gefragt ist, haben meistens funktionalen Charakter.

Auf der anderen Seite gibt es wichtige Klassen von Problemen, die sich funktional nur sehr schwer lösen lassen, da funktionale Programme keinen Zustand haben. Prinizipiell sind dies alle Probleme, die einen internen Zustand haben, der sich auf Grund von Ereignissen verändert. Ein typisches Beispiel dafür sind Benutzerschnittstellen, die sich in MFL nicht programmieren lassen. Aus diesem Grund wurde auf Ein- und Ausgabe-Anweisungen in MFL völlig verzichtet.

Es gibt zwei Ansätze, diese Schwächen der funktionalen Sprachen zu überwinden. Beim puristischen Ansatz versucht man, die funktionale Semantik der Sprache zu erhalten. Bei der "Ingenieur-Lösung" versucht man die positiven Seiten verschiedener Sprachparadigmen in einer hybriden Lösung zu vereinen.

3.6.2. Die puristische Lösung: Höhere, funktionale Sprache

MFL ist eine *strikte*, funktionale Sprache, was bedeutet, dass eine Funktion erst dann ausgeführt wird, wenn alle Eingabewerte berechnet sind. Dies ist zwar häufig die effizienteste, aber nicht die einzige Art, ein funktionales Programm auszuführen. Eine andere, nicht strikte Berechnungsstrategie ist beispielsweise "*lazy evaluation*". Man berechnet dort ausgehend vom Resultat nur jene Werte, die für das Resultat überhaupt eine Rolle spielen. Zusammen mit "lazy evaluation" machen auch potentiell *unendlich grosse Datenstrukturen*, sogenannte "streams", einen

Sinn, da die Elemente einer derartigen Struktur nur nach Bedarf berechnet werden und somit nicht zu einer unendlichen Schleife im Programm führen. Moderne, rein funktionale Sprachen verfügen über diese Möglichkeiten. Im Abschnitt 3.7 "Aehnliche Arbeiten" wird noch auf eine Reihe derartiger Sprachen und die Unterschiede zu MFL hingewiesen.

Mit unendlichen Datenstrukturen ist dem Programmierer, zumindest theoretisch, ein Instrument gegeben, um potentiell unendliche Folgen von Ereignissen zu verarbeiten. Boute [Boute86] hat erklärt, wie man einen primitiven Editor in einer funktionalen Sprache schreibt. Obwohl ihm daran gelegen war, zu zeigen, wie "mächtig" funktionale Sprachen selbst für derartige Anwendungen sind, zeigt allein die Tatsache, dass man eine grössere Diskussion über ein relativ primitives Problem der Informatik schreiben muss, dass funktionale Sprachen nicht das beste Ausdrucksmittel zur Programmierung von ereignisgesteuerten Applikationen sind.

Hinzu kommt, dass unseres Wissens weder auf Ein- noch auf Multiprozessoren mit prozeduralen Sprachen vergleichbar effiziente Implementationen von Laufzeitsystemen für derartige Sprachen existieren. Im Gegensatz dazu existieren solche Laufzeitsysteme für strikte, funktionale Sprachen [CanFeo90].

Zusammenfassend kann man sagen, dass sich funktionale Sprachen, trotz ihrer unbestrittenermassen positiven Eigenschaften, kaum in ihrer reinen Form werden durchsetzen können. Es muss also eine bessere Lösung zur Ueberwindung der Grenzen von MFL gefunden werden.

3.6.3. Die "Ingenieur-Lösung": Hybride Programmiersprache

Hybride Programmiersprachen sind sehr erfolgreich. Am besten lässt sich diese Aussage am Beispiel moderner, objekt-orientierter Sprachen belegen. Man hat erkannt, dass Klassen und Vererbung sehr mächtige Strukturierungshilfsmittel sind, die sich aus einer modernen Programmiersprache kaum mehr wegdenken lassen. Dennoch scheint es nicht sinnvoll zu sein, mit einer rein objekt-orientierten Sprache zu arbeiten. Der kommerzielle Erfolg von Smalltalk-80 [Goldbe83] ist vernachlässigbar gegenüber dem Erfolg von C++ [Strous86] und anderen objekt-orientierten Derivaten konventioneller Programmiersprachen. Man will die Addition "a+b" nicht als Senden einer Plus-Meldung mit Parameter "b" an das Zahlenobjekt "a" verstehen. Die übliche, mathematische Notation für artihmetische Ausdrücke ist ein zu starker Teil unserer Kultur.

Aus diesen Gründen zogen wir es bei MFL vor, den Weg Richtung *hybrider Sprache* zu gehen. Die funktionale Kernsprache MFL wird dabei in die sequentielle, moderne und objekt-orientierte Sprache FOOL [MurMar92] verpackt. Die Trennung zwischen dem sequentiellen und dem funktionalen Teil von *FOOL* verläuft zwischen Prozeduren und Funktionen. Prozeduren haben die gewohnte, sequentielle Semantik. Innerhalb von Funktionen gelten die Einfachzuweisungsregeln. Von Prozeduren aus dürfen Funktionen aufgerufen werden, aber nicht umgekehrt. Ist man einmal in die funktionale Welt eingetaucht, so bleibt man dort bis zum Ende der ersten aufgerufenen Funktion. Dementsprechend dürfen in Prozeduren lokale Funktionen deklariert werden. Wie Prozeduren können Funktionen als Methoden in Objektklassen definiert werden. Der Unterschied besteht dabei darin, dass Prozeduren die Instanzvariablen, also den Zustand eines Objekts verändern können, Funktionen aber nicht. Für den funktionalen und den prozeduralen Teil von FOOL wird dasselbe Typensystem verwendet, sodass sich Daten problemlos zwischen Prozeduren und Funktionen austauschen lassen. Eine vollständige Beschreibung von FOOL würde den Rahmen dieser Dissertation sprengen. In Anhang A ist die vollständige Syntax von FOOL inklusive MFL abgedruckt. Eine vollständige Beschreibung ist in [MarMur92a] publiziert.

3.7. Aehnliche Arbeiten

Eine Vorgänger-Version von MFL wurde schon in einer früheren Dissertation [Wytten90] beschrieben. Die tatsächlich implementierte Version unterscheidet sich aber wesentlich von dieser ersten Version, so dass es notwendig war, eine Beschreibung der aktualisierten Version von MFL in diese Arbeit aufzunehmen, obwohl der Sprachentwurf nicht das eigentliche Thema der Arbeit ist.

MFL gehört in eine Reihe von Einfachzuweisungssprachen, die in den letzten Jahren entwickelt wurden. Prominente Beispiele dafür sind LAU [Plas76], VAL [McGraw82] und SISAL [McGraw85].

Am stärksten wurde unsere Arbeit sicherlich von SISAL beeinflusst. SISAL wurde nicht für eine bestimmte Maschine entworfen. Das Format der Datenflussgraphen wurde früh publiziert [SkeGla84], so dass Codegeneratoren für möglichst viele Maschinen geschrieben werden können. In der Tat wurde eine Menge verschiede-

ner Codegeneratoren für diverse Maschinen gebaut[3]. SISAL wurde hauptsächlich als Sprache für numerische Applikationen entwickelt. Es wurde mehr Wert auf effiziente Implementierbarkeit als auf theoretische Eleganz gelegt, so dass die Sprache einige mächtige Elemente funktionaler Sprachen nicht enthält. Insbesondere unterstützt SISAL noch keine Funktionen höherer Ordnung. Die neuesten Arbeiten mit SISAL [CanFeo90] haben gezeigt, dass eine effiziente Implementation funktionaler Sprachen auf konventionellen Multiprozessoren möglich und sinnvoll ist.

Für das Datenflussprojekt [ArvNik89] am MIT wurde die Sprache Id und später Id Nouveau [Nikhil88] entwickelt. Id besteht aus einem funktionalen Kern mit Funktionen höherer Ordnung und einer Reihe von nicht funktionalen Erweiterungen. Die wichtigste Erweiterung sind die I-Strukturen [ArNiPi86], eine Art nichtstrikter Arrays. Jedes Feld einer I-Struktur hat ein zugeordnetes Präsenz-Bit. Ist dieses durch Schreiben gesetzt, so darf das Feld sofort gelesen werden, wird es dann noch einmal geschrieben, so ist dies ein Verstoss gegen die Einfachzuweisungsregel und ein Fehler wird signalisiert. Ist das Präsenz-Bit nicht gesetzt, so werden alle Lesezugriffe in eine Warteschlange eingeordnet. Sobald dann ein Wert geschrieben wird, wird das Bit gesetzt und alle wartenden Lesezugriffe werden ausgeführt. Im Gegensatz dazu kennt MFL nur strikte Datenstrukturen. Zur Laufzeit wird zwar kontrolliert, ob ein Array-Feld vor dem Gebrauch zugewiesen wurde. Ist dies nicht der Fall, wird im Gegensatz zu Id ein Fehler signalisiert und das Programm fährt weiter. Der Vorteil von I-Strukturen besteht darin, dass zusätzlich die Produzenten/Konsumenten-Parallelität ausgenützt werden kann. Dem stehen drei wesentliche Nachteile gegenüber:

1. Man kann keine Fehler entdecken, bei denen ein Wert in einer Struktur nie geschrieben wird, weil dann das Programm nie terminiert.

2. Die Implementation von I-Strukturen ist ziemlich teuer, da für jedes Element einer Struktur ganze Warteschlangen von Codeblöcken geführt werden müssen.

---

3) Auch für die ADAM-Architektur wird ein SISAL-Codegenerator entwickelt. Unter anderem wurde auch ein SISAL-Codegenerator für die Manchester-Datenflussmaschine [GuKiWa85], die erste HW- Implementation der Datenfluss-Idee, gebaut.

3. Mit I-Strukturen lassen sich Verklemmungen programmieren. Das folgende Beispiel zeigt das Problem:

```
...
i := j;
...
a[i] := a[j];
```

Abschliessend muss man sagen, dass für Sprachen mit nicht-strikter Semantik das Partitionierungsproblem (vgl. Kapitel 5 "Partitionierung von Datenflussgraphen in Codeblöcke") nicht ein reines Optimierungsproblem bleibt. Nehmen wir an, wir hätten folgendes Programmstück:

```
...
a[i] := a[j];
...
a[k] := c;
```

Führt man dieses Programmstück auf einer feingranularen Datenfluss-Maschine mit I-Strukturen aus, so wird am Ende a[i] = c gelten, falls k = j ist. Sequentialisiert man das Programm wie im Text dargestellt, was kein Verstoss gegen die statischen Datenabhängigkeiten ist (sehr wohl aber gegen die dynamischen), terminiert das Programm mit k = j nicht. Solche Fälle muss der Compiler bei der Partitionierung berücksichtigen, was zu ungünstig kleinen Codeblöcken führen kann.

Aus all diesen Gründen haben wir uns nach reiflicher Ueberlegung gegen I-Strukturen in MFL entschieden.

4. Datenflussgraphen als Zwischencode für MFL

In diesem Kapitel wird die Darstellung von funktionalen Programmen als Datenflussgraphen eingeführt und definiert. Mit Hilfe unserer Sprache MFL und einigen Beispielen wird gezeigt, wie MFL-Programme auf Datenflussgraphen abgebildet werden können. Diese Graphen bilden in den folgenden Kapiteln den Ausgangspunkt für die Diskussion von verschiedenen Problemen im Zusammenhang mit der Codegenerierung für funktionale Sprachen. Im letzten Teil diese Kapitels wird die konkrete Darstellung der diskutierten Graphen als Modula-2-Datenstruktur angegeben. Diese Datenstruktur wird vom MFL-Compiler als Zwischencode für die Codegenerierung verwendet.

Der Leser sollte mit den Grundbegriffen der Graphentheorie vertraut sein. [Läuchl88] oder [Tutte84] geben eine verständliche Einführung. Ausserdem sollte man sich mit den Grundideen von funktionalen Sprachen vertraut machen (vgl. [Ackerm82] oder Kap. 3 "MFL: Eine einfache Einfachzuweisungssprache").

4.1. Funktionale Sprachen und Datenflussgraphen

Unter *Datenflussgraphen* verstehen wir die Darstellung eines Programms in graphischer Form, wobei die Knoten den einzelnen Operationen und die Kanten den Datenabhängigkeiten unter den Operationen entsprechen.

Natürlich lassen sich nicht alle Sprachen leicht zu Datenflussgraphen übersetzen. Klassische sequentielle Sprachen enthalten neben Datenabhängigkeiten eine Reihe von zusätzlichen Abhängigkeiten, da Namen nicht Werte sondern Speicherplätze bezeichnen. Untersuchen wir beispielsweise die Abhängigkeiten in der folgenden Programmsequenz:

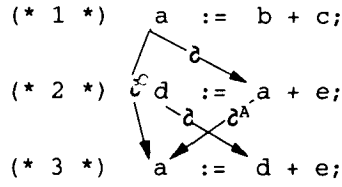


Fig. 4.1: Verschiedene Abhängigkeiten in Programmen

Wir haben hier drei verschiedene Arten von Abhängigkeiten [KuKPLW81], nämlich *Datenflussdependenzen* (∂), bei denen ein zugewiesener Wert in einem späteren Ausdruck gebraucht wird, *Zuweisungsdependenzen* (∂^O), bei denen zweimal

dieselbe Variable zugewiesen wird und *Antidependenzen* (∂^A), bei denen Variablen in Ausdrücken verwendet werden müssen, bevor sie von neuem zugewiesen werden. Wesentlich sind eigentlich nur die Datenflussdependenzen. Die anderen Abhängigkeiten können mehr oder weniger gut eliminiert werden, indem man gewissen Variablen neue Namen gibt (*Renaming*). Im obigen Beispiel müsste man einfach "a" in der dritten Zuweisung durch eine neue Variable "f" ersetzen, damit die Zuweisungs- und die Antidependenzen verschwinden. Uebrig bleiben die reinen Datenflussdependenzen (∂) und somit der Datenflussgraph.

Etwas schwieriger wird die Analyse, wenn man anstelle von skalaren Variablen in Sequenzen strukturierte Variablen in komplexen Kontrollstrukturen betrachtet. Die Forschung hat auf diesem Gebiet in den letzten 15 bis 20 Jahren einige Fortschritte erzielen können, die schliesslich zu Entwicklung und wirtschaftlichem Einsatz von vektorisierenden und bis zu einem gewissen Mass auch parallelisierenden Compilern auf der Basis konventioneller Sprachen geführt hat [PadWol86]. Die Datenflussanalyse für konventionelle Sprachen versagt aber beim Vorhandensein von Seiteneffekten über globale Variablen oder Speicherplätzen, auf die über mehrere Namen zugegriffen wird[1]. Zusammenfassend gesagt, ist man heute in der Lage, einfache Schleifen in Sprachen mit einfachem Speichermodell (FORTRAN) zu parallelisieren. Dies genügt für Maschinen mit kleinem Parallelitätspotential[2], versagt aber für massiv parallele Maschinen [Wolfe90].

Ein weiteres Problem mit der Parallelisierung konventioneller Sprachen ist, dass der Programmierer seinen vom Wesen her parallelen Algorithmus sequentiell ausdrücken muss und der Compiler die Aufgabe hat, aus diesem sequentiellen Programm die ursprüngliche Parallelität wieder zurückzugewinnen. Da der Compiler jedoch nur über eingeschränkte Analysemöglichkeiten verfügt, muss ihm entweder der Programmierer zusätzliche Hinweise zur Programmierung geben (Compilerdirektiven) oder seinen Programmierstil nach den Analysemöglichkeiten des Compilers richten. Für den Programmierer bleibt so zwar die vertraute Pro-

---

1) Im Englischen wird dieses Problem als "Aliasing" bezeichnet. In Modula-2 oder Pascal ist dies beim Vorhandensein mehrerer Zeigern auf den gleichen Speicherbereich oder VAR-Parametern der Fall.

2) Hier können wir die auf diese Weise erfolgreich zu programmierenden Vektormaschinen als Beispiel anführen, bei denen das Parallelitätspotential der Pipelinetiefe, also beispielsweise 6 oder 7 für die Fliesskomma-Pipelines in der Cray-1 [Hwang,Briggs], entspricht.

grammiersprache, nicht aber deren Programmiermodell erhalten. Obwohl dies durch die Erhaltung der alten Programmiersprachen suggeriert wird, ist es deshalb eine Illusion, dass sich aus alten Programmen durch reine Rekompilation parallele Codes gewinnen lassen.

Im Kapitel 3 "MFL - Eine einfache, funktionale Sprache" wurde ein Beispiel aus einer Klasse von Programmiersprachen vorgestellt, die sich dank geeigneter Einschränkungen in der Sprache leicht auf Datenflussgraphen abbilden lässt.

4.2. Einfache Graphen

Graphen und die entsprechenden mathematischen Werkzeuge aus der Graphentheorie werden in der Informatik und anderen Ingenieurwissenschaften für verschiedenste Probleme eingesetzt, bei denen Aktivitäten koordiniert werden sollen. Bei der Planung von komplexen Bauvorhaben werden Netzpläne erstellt, die eigentlich nichts anderes als gerichtete, azyklische Graphen sind. Die Kanten und deren Gewichte stellen dabei Aktivitäten bzw. deren Dauer dar, die Knoten bezeichnen die Präzedenzbeziehungen, oder umgekehrt.

Ein anderes Beispiel aus der allgemeinen Informatik sind die Petri-Netze [Reisig85]. Petri-Netze sind gerichtete, bipartite Graphen bestehend aus Plätzen und Transitionen und Kanten. Plätze können eine Anzahl Marken enthalten und Transitionen können feuern. Eine Transition feuert, wenn alle ihre Eingangsplätze besetzt sind. Sie konsumiert dabei von jedem dieser Plätze eine Marke und produziert auf allen Ausgangsplätzen eine neue Marke.

Im klassischen Compilerbau [AhSeUl86] werden zwei Arten von Graphen unterschieden. Einerseits haben wir Bäume oder gerichtete, azyklische Graphen zur Darstellung von arithmetischen Ausdrücken mit oder ohne gemeinsame Unterausdrücke. Knoten entsprechen dabei den Operatoren und Unterbäume entsprechen den Unterausdrücken. Bei gemeinsamen Unterausdrücken führen mehrere Kanten auf den gleichen Teilbaum. Andererseits werden allgemeine, gerichtete Graphen zur Darstellung des Kontrollflusses verwendet.

Auf den beiden oben genannten Ideen basieren Dennis' Datenflussgraphen [Dennis75]. Knoten bezeichnen dort Operationen und Kanten bezeichnen Datenabhängigkeiten wie bei den arithmetischen Ausdrücken. Jede Kante kann mit einer

Marke[1], die einen Datenwert trägt, belegt werden. Operatoren feuern, sobald alle ihre Eingangskanten mit einer Marke belegt sind, und produzieren dann auf allen Ausgangskanten eine neue Marke mit dem entsprechenden Resultat. Kontrollstrukturen werden in diesen Graphen mit Hilfe von Verzweigungen und Rückführungen im Datenfluss, den "Split", "Merge" und "Gate"-Operatoren dargestellt. Die allgemeinen Datenflussgraphen sind also gerichtet, aber nicht azyklisch. Wie in der Einführung (vgl. Kap. 2 "Die ADAM-Architektur") erwähnt, gibt es sogenannte feingranulare Datenflussrechner, die Graphen direkt als Maschinenprogramm verwenden, und nach dem oben beschriebenen Operationsprinzip funktionieren.

Die bisher vorgestellten Datenflussgraphen werden *statische Datenflussgraphen* genannt, da es keine Möglichkeit gibt, mehrere, unterscheidbare Aktivationen des gleichen Graphen zu haben. Wollen wir rekursive Programme mit Datenflussgraphen ausführen, so reicht es nicht, nur Datenwerte auf den Marken mitzuführen. Wir müssen den Marken auch eine Identifikation ("Tag", "Farbe") mitgeben, die die aktuelle Inkarnation des Graphen bezeichnet. Mit dieser Erweiterung erhalten wir *dynamische Datenflussgraphen* [ArvNik89].

Die *hierarchischen Datenflussgraphen,* die im MFL/FOOL-Compiler Verwendung finden und im folgenden beschrieben werden sollen, basieren auf dem gemeinsamen Zwischencode für den SISAL-Compiler [SkeGla84].

Wir definieren einen hierarchischen Datenflussgraphen in Anlehnung an die Definition von Graphen in der Graphentheorie G als Tripel,

$G = (N, E, U)$

wobei N die Menge der Knoten, E die Menge der Kanten und $U \in N$ den *Umgebungsknoten* von G bezeichnet. Der Umgebungsknoten stellt die Beziehung des Graphen nach aussen dar. Jede Kante, die von aussen in den Graphen hineinführt, kommt von einem Ausgang des Umgebungsknotens, und jede Kante, die den Graphen verlässt, führt auf einen Eingang des Umgebungsknotens. Der Umgebungsknoten wurde aus Gründen der Bequemlichkeit eingeführt. Dank ihm müssen für

---

[1]) Wir verwenden im Folgenden den deutschen Begriff Marke, obwohl in der englischen Fachliteratur im Zusammenhang mit Datenflussgraphen immer von "Token" gesprochen wird. Tatsächlich können auch mehrere Marken pro Kante in einem FIFO-Buffer zugelassen werden, ohne dass sich die Semantik eines Datenflussgraphen verändert.

Ein- und Ausgänge des Graphen keine Spezialfälle in den nachfolgenden Definitionen unterschieden werden. Kanten sind gerichtet. Jede Kante verbindet einen bestimmten *Ausgang* eines Knotens (n, z) mit einem bestimmten *Eingang* eines Knotens im gleichen Graph:

$$N \times Z^+ \times N \times Z^+ \supset E$$

Eingänge und Ausgänge eines Knotens müssen näher spezifiziert werden, da die Position der Operanden eines Operators natürlich eine Rolle spielt. Knoten dürfen mindestens einen Ausgang haben. Sie sind aber nicht auf nur einen Ausgang beschränkt, wie dies in Ausdrücken und klassischen Datenflussgraphen häufig gefordert wird. Ein- und Ausgänge sind bei jedem Knoten mit ganzen Zahlen ab 1 numeriert.

$$\forall e_1,e_2 \in E: e_1 \neq e_2 \wedge e_1=(n_1, z_1, n_1', z_1') \wedge e_2 = (n_2, z_2, n_2', z_2') \Rightarrow (n_1', z_1') \neq (n_2', z_2')$$

Zu einem bestimmten Eingang hin führt immer nur eine Kante[1]. Diese Einschränkung macht die Ausführung eines Graphen erst deterministisch. Ausgänge können auf beliebig viele Kanten führen[2]. Von einem Ausgang kann hingegen eine beliebige positive Anzahl Kanten wegführen. Von Knoten mit mehreren Ausgängen müssen nicht alle Ausgänge mit Eingängen anderer Knoten verbunden sein. Es muss aber mindestens ein Ausgang jedes Knotens zu einem anderen Knoten führen, da sonst dieser Knoten nichts zur Berechnung beitragen würde. Solche Fälle kann man häufig bei Subgraphen von zusammengesetzten Knoten (vgl. Abschnitt 4.3 "Zusammengesetzte Knoten") beobachten. Zudem gibt es *konstante Kanten*. Sie liefern einen konstanten Wert am entsprechenden Eingang. Aus theoretischen Gründen nehmen wir an, dass die konstanten Kanten immer an einem Ausgang des Umgebungsknotens hängen[3].

---

[1]) Anders ausgedrückt: Der "Fan in" ist immer 1.

[2]) Von einem normalen Knoten sollte immer mindestens eine Kante wegführen, da sonst der Knoten keinen Sinn macht. Einzig beim Umgebungsknoten können alle Ausgänge unbenutzt sein, wenn ein Subgraph eines zusammengesetzten Knotens nur konstante Kanten verwendet.

[3]) In den Graphiken und in der Datenstruktur zur Repräsentation der Knoten (vgl.) wird diese Verbindung jedoch nicht dargestellt.

Die Graphen sind *azyklisch*, wenn man den Umgebungsknoten U nicht in Betracht zieht. E definiert die *partielle Ordnung* $\leq^* := (E_{|N \times N})$ unter den Knoten. U ist bezüglich dieser Ordnung gleichzeitig der grösste und der kleinste Knoten. Alle Eingänge stammen von den Ausgängen des Umgebungsknotens und alle Ausgänge laufen schliesslich auf den Eingängen des Umgebungsknotens zusammen. U ist die Wurzel des Graphen.

$\forall n \in N \backslash U: U \leq^* n \leq^* U$

Es gibt drei Klassen von Knoten: Primitive Knoten, zusammengesetzte Knoten und Umgebungsknoten:

1. *Primitive Knoten* stellen die Operatoren dar, die von der Maschine direkt ausgeführt werden. Es gibt also Addition, Multiplikation, usw. Die Art des primitiven Knotens bestimmt die Anzahl von Ein- und Ausgängen des Knotens. So hat eine Addition zwei Eingänge und einen Ausgang.

2. Zu jedem Graphen definieren wir den *Umgebungsknoten*, dessen Ausgänge den Daten entsprechen, die von aussen in den Graphen hineinkommen und auf dessen Eingänge die Kanten geführt werden, die den Graphen verlassen. Der Umgebungsknoten definiert die Schnittstelle des Graphen nach aussen.

3. Mit Hilfe von *zusammengesetzten Knoten* werden die Kontrollflusskonstrukte von MFL dargestellt. Ueber zusammengesetzte Knoten werden die Datenflussgraphen zu *hierarchischen, azyklischen* Datenflussgraphen. Später in diesem Kapitel werden die verschiedenen Klassen von zusammengesetzten Knoten noch im Detail beschrieben.

Graphisch stellen wir die einzelnen Elemente der Datenflussgraphen in diesem Kapitel wie in der Figur 4.2 gezeigt dar:

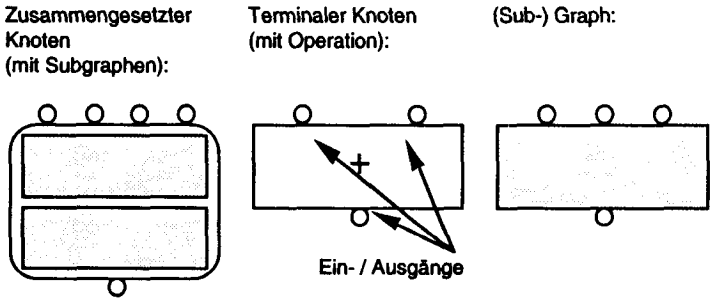


Fig. 4.2: Graphische Darstellung von Datenflussgraphen

Schauen wir uns in der folgenden Zeichnung (Figur 4.3) die Elemente eines Datenflussgraphen am einfachen Beispiel des MFL-Programms und des entsprechenden Graphen zur Berechnung der beiden Lösungen einer quadratischen Gleichung an. Man sieht den Funktionsgraphen mit seinen drei Eingängen a, b und c, sowie eine Menge von einfachen Knoten und Kanten, die die Knoten untereinander verbinden. Ebenso finden wir *konstante Kanten*, die immer den gleichen Wert liefern.

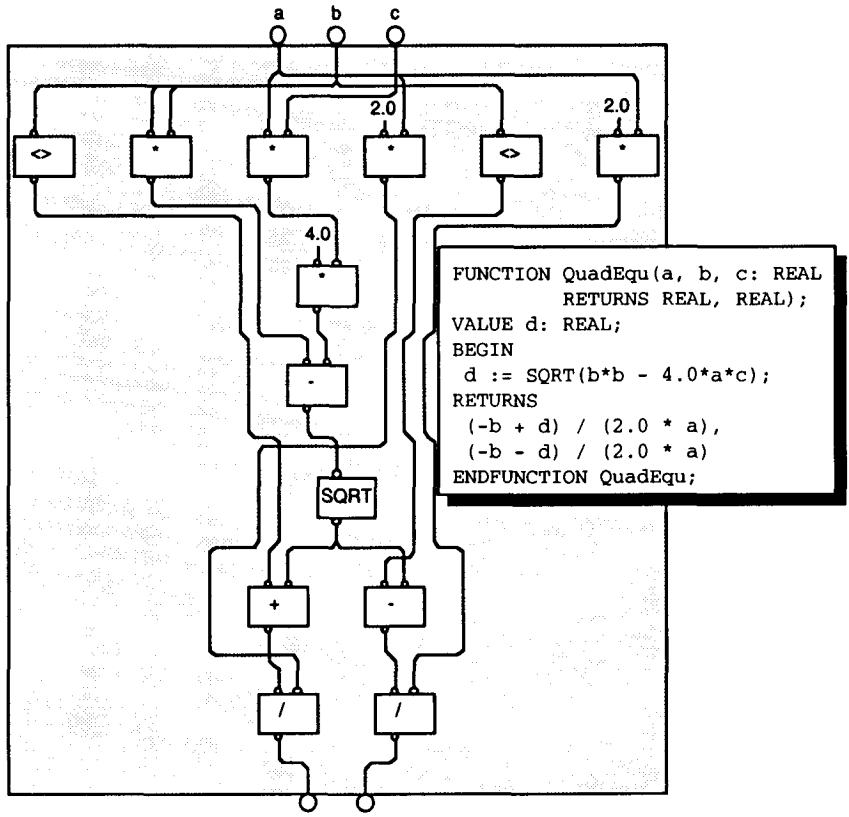


Fig.4.3: Beispiel: Die zwei Lösungen der quadratischen Gleichung

4.3. Zusammengesetzte Knoten

Da die verwendeten Graphen per definitionem keine Zyklen zulassen, brauchen wir andere Möglichkeiten, um Kontrollflusskonstrukte[1] darzustellen. Zusammengesetzte Knoten im Datenflussgraph entsprechen genau den komplexen Ausdrücken in MFL (vgl. Abschnitt 3.5 "Komplexe Ausdrücke"). Sie bestehen aus einem Knoten und einer Reihe von Subgraphen. Die Subgraphen entsprechen den einzelnen Teilen des entsprechenden Sprachkonstrukts. So definieren wir also einen Knoten rekursiv als eine Menge von Graphen:

$N = \{G\}$

Die Art des Knotens schränkt die Menge der Subgraphen ein. Primitive Knoten haben keine Subgraphen und zusammengesetzte Knoten haben eine Subgraphmenge mit fixer Kardinalität entsprechend der Art des zusammengesetzten Knotens. Zwischen den Ein- und Ausgängen der Subgraphen und des Knotens bestehen implizite Beziehungen, die durch die Semantik des Knotens festgelegt sind. Subgraphen haben mindestens gleich viele Eingänge wie der entsprechende Knoten. An den Eingängen der Subgraphen liegen dieselben Werte an, wie an den entsprechenden Eingängen des zusammengesetzten Knotens. Es gibt mindestens einen Subgraphen, dessen Ausgänge den Ausgängen des Knotens entsprechen.

4.3.1. Bedingte Ausdrücke

Betrachten wir als erstes Beispiel Subgraphen und implizite Datenabhängigkeiten im zusammengesetzten Knoten für einen einfachen bedingten Ausdruck gemäss Figur 4.4.

Der *IF-THEN-ELSE-Knoten* besteht aus folgenden drei Subgraphen: Dem *IF-Subgraph*, dessen Ausgang bestimmt, ob der *THEN-Subgraph* (TRUE) oder der *ELSE-Subgraph* (FALSE) ausgeführt werden soll. Die impliziten Datenabhängigkeiten dieses Knotens verbinden alle Eingänge des Knotens mit den entsprechenden Eingängen der Subgraphen (1 in Fig. 4.4). Die Ausgänge des THEN- und des ELSE-Subgraphen (3 in Fig. 4. 4) werden entsprechend dem Wert am Ausgang des IF-Subgraphen (2 in Fig. 4.4) mit den Ausgängen des Knotens verbunden. Daraus folgt, dass alle Subgraphen gleich viele Eingänge und die Subgraphen für die bei-

---

[1] Mit Kontrollflusskonstrukten sind Schleifen und Alternativen (IF-THEN-ELSE) gemeint.

den Alternativen gleich viele Ausgänge wie der Knoten haben müssen. Der IF-Subgraph hat immer nur einen Ausgang vom Typ BOOLEAN.

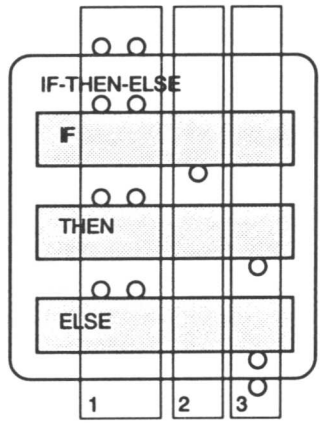


Fig. 4.4: Zusammengesetzter Knoten: IF-THEN-ELSE

Das in der Figur 4.5 dargestellte Beispiel einer einfachen MFL-Funktion, die zwei Zahlen sortiert, und deren Graph illustriert den Zusammenhang zwischen einem komplexen MFL-Ausdruck und dem entsprechenden zusammengesetzten Graphen.

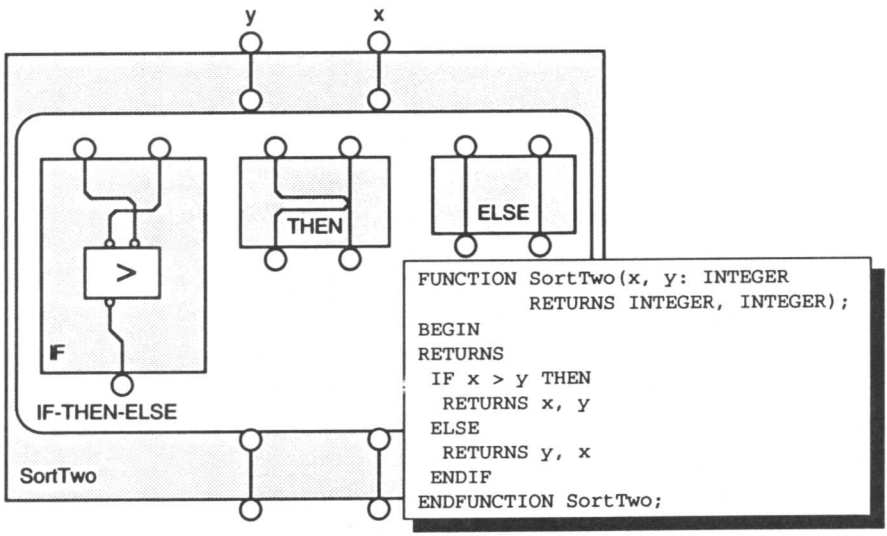


Fig. 4.5: Beispiel: Sortieren von zwei Zahlen

In MFL kann bei bedingten Ausdrücken mit Hilfe von ELSIF auch zwischen mehr als zwei Alternativen gewählt werden. Der zusammengesetzte Knoten lässt dementsprechend eine beliebige ungerade Anzahl von Subgraphen zu. Je zwei Graphen zusammen bilden dann Paare von Bedingung und entsprechender Aktion und der letzte Graph entspricht dem ELSE-Ausdruck.

4.3.2. Iterative Ausdrücke

Die iterativen Konstrukte von MFL werden ebenfalls durch entsprechende, zusammengesetzte Knoten dargestellt. Wir werden als Beispiel die Schleife mit Test vor dem Schleifenkörper (LOOPB)[1] näher betrachten:

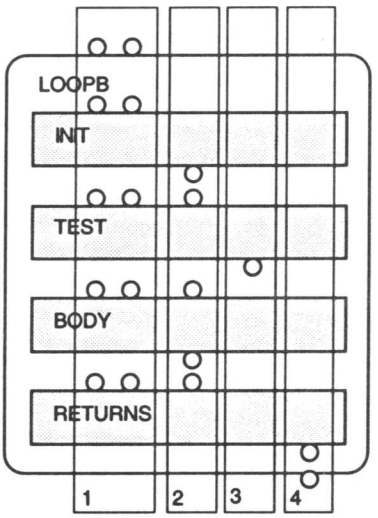


Fig. 4.6: Zusammengesetzter Knoten: LOOPB

Die Iterationsknoten haben immer vier Subgraphen: *Initialisierung* (INIT), *Test* (TEST), *Schleifenkörper* (BODY) und *Rückgabeausdruck* (RETURNS). Diese vier Subgraphen entsprechen direkt den entsprechenden Teilen eines komplexen Iterationsausdrucks in MFL. Auch bei den Iterationsknoten werden die Eingänge des Knotens direkt an alle Subgraphen weitergegeben (1 in Fig. 4.6). Die *Schleifenvariablen* bekommen im INIT-Graphen ihren ersten Wert und erscheinen an den Eingängen des TEST-Subgraphen (2 in Fig. 4.6). Abhängig davon, ob der Test an seinem Ausgang (3 in Fig. 4.6) TRUE oder FALSE ergibt, werden sie dann auf die

---

[1] LOOPB steht für "Loop with test Before". Diese Schleife entspricht dem WHILE-Konstrukt in Modula-2.

Eingänge des BODY-Subgraphen bzw. des RETURNS-Subgraphen geleitet. Der BODY-Subgraph produziert an seinen Ausgängen eine neue Generation von Werten, die dann wieder oben in den TEST-Subgraphen eingespiesen werden, womit der Zyklus geschlossen ist. Am Ende generiert der RETURNS-Subgraph die Resultate, die am Ausgang des Knotens erscheinen (4 in Fig. 4.6).

Konzeptionell sind die Schleifenvariablen ein lokaler, streng begrenzter Verstoss gegen das Einfachzuweisungsprinzip. Darum ist es auch angebracht, in diesem Zusammenhang von Variabeln und nicht von Werten zu sprechen. Diese Variablen lassen sich dementsprechend nicht direkt durch Graphen repräsentieren, weshalb sie durch die spezielle Semantik der zusammengesetzten Schleifenknoten dargestellt werden. Man kann die Schleifenvariablen als den vollständigen Zustand der Schleife interpretieren. Der BODY-Subgraph definiert dann den Zustandsübergang von einer alten zu einer neuen Generation von Schleifenvariablen. Ein Zustandswechsel findet nur an der Grenze zwischen zwei Iterationen statt.

Die zwei anderen Iterationsknoten in unseren Datenflussgraphen unterscheiden sich nur durch die Art der Schleifenkontrolle von LOOPB. Der LOOPA-Knoten testet die Bedingung erst nach dem Durchlaufen des Schleifenkörpers[1]. Der FOR-Knoten hat an Stelle eines TEST-Graphen einen INDEX-Graphen, in welchem die Anzahl Durchläufe a priori durch einen Laufbereichsausdruck bestimmt ist. In einen zusätzlichen Eingang des BODY-Subgraphen wird der aktuelle Schleifenindex eingespeist.

---

[1]) LOOPA steht für "Loop with test After". Diese Schleife entspricht dem REPEAT-Konstrukt in Modula-2.

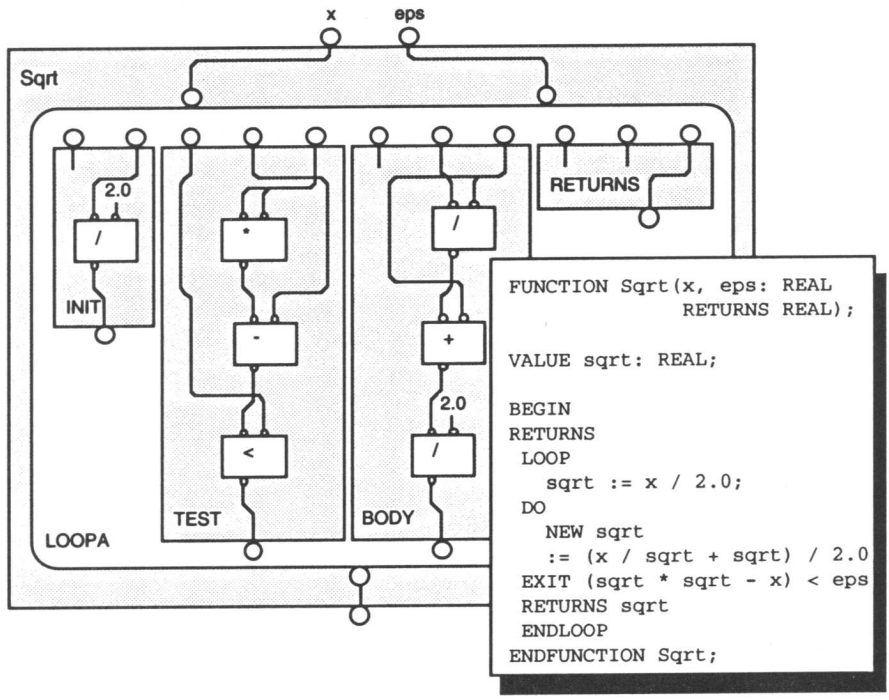


Fig. 4.7: Beispiel: Berechnung der Quadratwurzel durch Newton-Approximation

Das Beispiel in Figur 4.7 zeigt MFL-Code und -Graph für die Berechnung der Quadratwurzel nach der Newton-Methode. Im wesentlichen besteht das Programm aus einer Schleife mit Test am Schluss (LOOPA), deren Schleifenkörper so lange durchlaufen wird, bis der Fehler klein genug ist.

4.3.3. Parallele Ausdrücke

Neben den sequentiellen Schleifenkonstrukten, gibt es einen speziellen *FORALL-Knoten*, um das FORALL-Konstrukt von MFL darzustellen:

Der FORALL-Knoten hat die gleichen Subgraphen wie der FOR-Knoten. Bei der Bedeutung der Ein- und Ausgänge gibt es aber Unterschiede. Wie üblich, werden die Eingänge des zusammengesetzten Knotens direkt an alle Subgraphen weitergeleitet (1 in Fig. 4.8). Auch im FORALL-Knoten gibt es eine Art von Schleifenvariablen (2 in Fig. 4.8). Es handelt sich dabei jedoch nicht um die Variablen, die in jedem Schleifendurchlauf neue Werte bekommen, sondern um jene Arrays, welche die Resultate der einzelnen Iterationen aufnehmen. Sie werden im INIT-Graphen leer generiert. Die einzelnen Array-Elemente bekommen in den

einzelnen Inkarnationen des BODY-Graphen je einen Wert und die gefüllten Arrays fliessen dann in den RETURNS-Graphen[1]. Der erste Ausgang des INDEX-Graphen (3 in Fig. 4.8) liefert die Anzahl parallele Teile (sog. "loop slices"), in die die einzelnen Inkarnationen des Schleifenkörpers verpackt werden sollen. Dieser Wert ist nur für die Codegenerierung interessant und hat keinen entsprechenden Ein- oder Ausgang in einem anderen Subgraphen. Im Gegensatz zum FOR-Knoten, wo nur ein Schleifenindex vorgesehen ist, kann im INDEX-Graphen des FORALL-Knotens der Laufbereich mehrerer Schleifenindices definiert werden (4 in Fig. 4.8). Der Schleifenkörper wird dann genau einmal pro Index-Tupel aus dem kartesischen Produkt aller Laufbereiche ausgeführt. Das Indextupel liegt dann an den entsprechenden Eingängen (4 in Fig. 4.8) des BODY-Subgraphen an. Wiederum entsprechend zu den anderen Iterationskonstrukten werden die Ausgänge des RETURNS-Graphen an die Ausgänge des zusammengesetzten Knoten weitergeleitet (5 in Fig. 4.8).

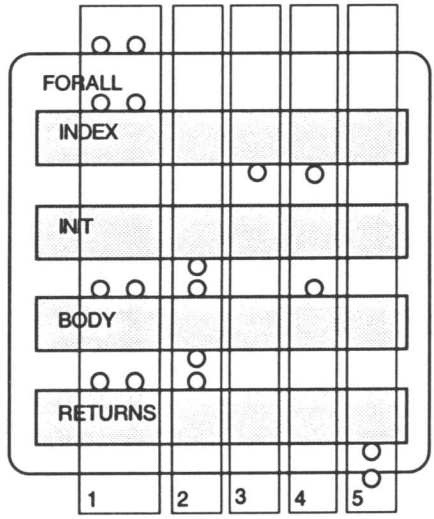


Fig. 4.8: Zusammengesetzter Knoten: FORALL

Die folgende Zeichnung (Fig. 4.9) zeigt am Beispiel der Addition zweier Matrizen einen FORALL-Knoten mit zwei Indices und einem im INIT-Subgraphen erzeugten Resultat-Array. Im RETURNS-Subgraphen wird der erzeugte Array direkt aus

---

1) Arrays, die im Schleifenkörper feldweise zugewiesen werden, haben auch in den sequentiellen Schleifen die gleiche Semantik wie hier. In FORALL-Knoten sind aber keine eigentlichen Schleifenvariablen möglich.

dem Knoten zurückgegeben. Dies muss nicht so sein. Es könnte dort ohne weiteres noch eine Berechnung auf dem Array durchgeführt werden. Beispielsweise könnten die Elemente des Arrays in einer sequentiellen Schleife summiert und nur die Summe zurückgegeben werden.

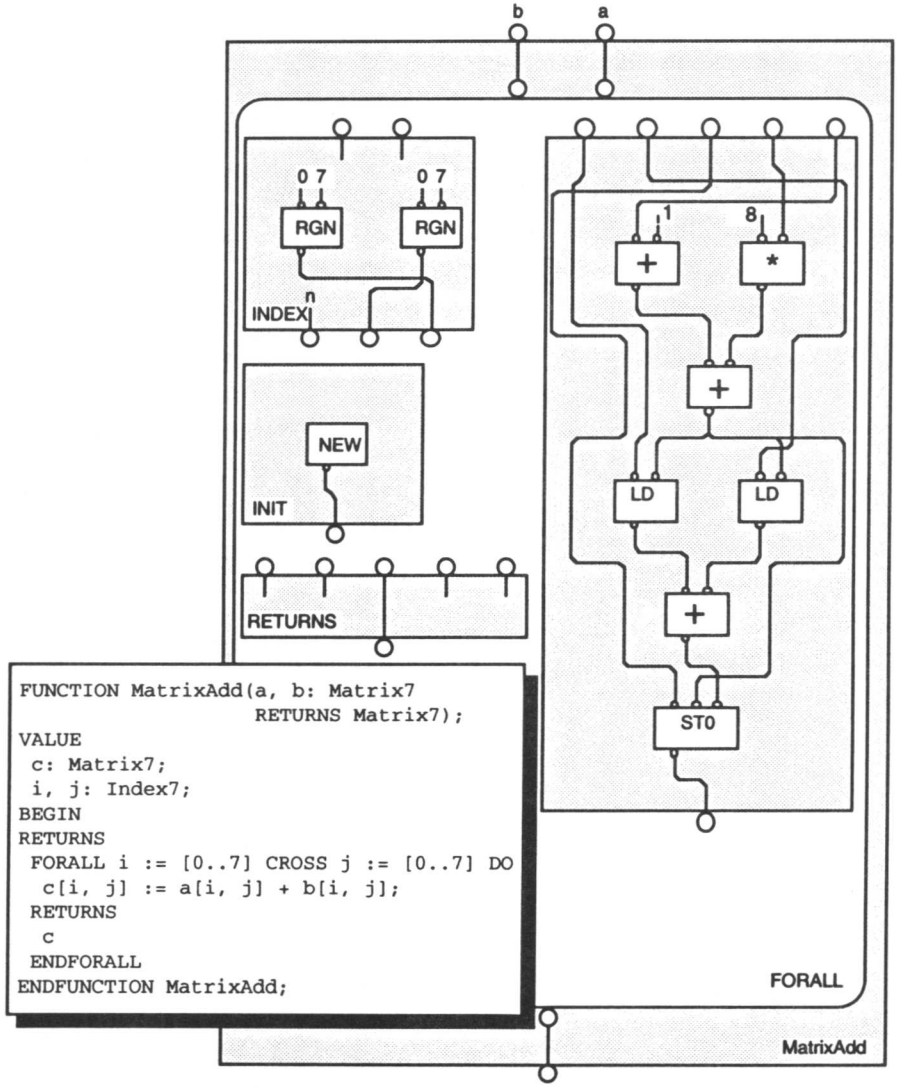


Fig. 4.9: Beispiel: Addition zweier Matrizen

Nebenbei zeigt dieses Beispiel auch, wie die Berechnung des eindimensionalen Objekt-Index aus den beiden logischen Indices "i" und "j" ebenfalls in die Graphen

eingefügt wird. Dadurch werden diese Index-Berechnungen auch durch die Optimierungstransformationen auf den Graphen erfasst. In diesem Fall werden die drei Arrays "a", "b" und "c" über die gleichen Indices zugegriffen, so dass diese als gemeinsamer Unterausdruck erkannt und zusammengefasst werden.

4.4. Funktionsgraphen und Rekursion

Unsere Datenflussgraphen sind *dynamisch,* was bedeutet, dass im Laufe der Programmausführung mehrere Inkarnationen desselben Funktionsgraphen existieren können. Es gibt einen speziellen *CALL-Knoten,* mit folgender Semantik: Der erste Eingang des CALL-Knotens bestimmt den Funktionsgraphen, der ausgeführt werden soll. Die folgenden Eingänge werden mit den Eingängen einer neuen Inkarnation des Funktionsgraphen verbunden. Der Funktionsgraph wird ausgeführt und die Werte an seinen Ausgängen werden an die Ausgänge des CALL-Knotens weitergeleitet. Die Anzahl der Ein- bzw. Ausgänge bei CALL-Knoten und aufgerufenem Funktionsgraph muss übereinstimmen. Innerhalb eines Funktionsgraphen kann ein CALL-Knoten, der sich auf den Graphen selbst bezieht, vorkommen: *Rekursion* ist also möglich. Das folgende Beispiel der rekursiven Berechnung der n-ten Fibonacci-Zahl zeigt die Anwendung des CALL-Knotens in einem rekursiven Programm:

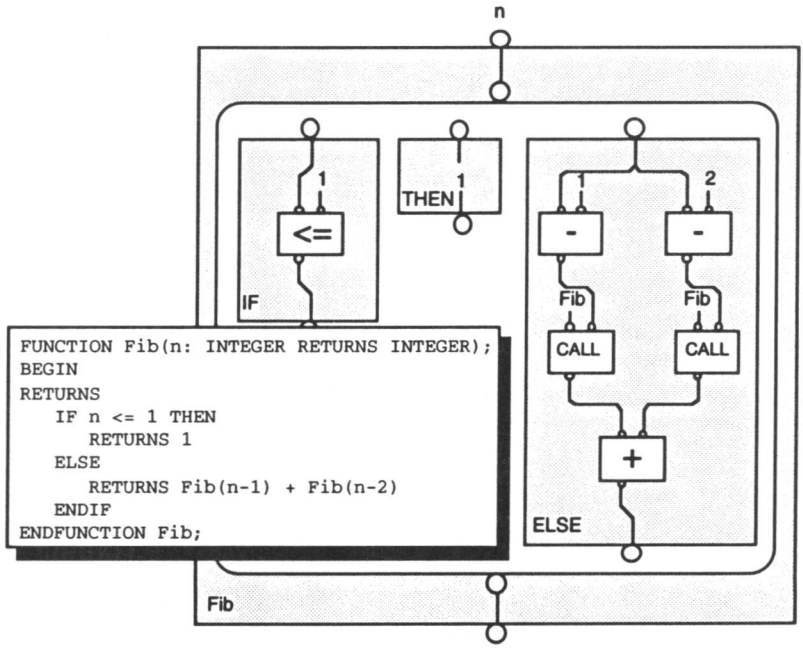


Fig. 4.10: Beispiel: Rekursive Berechnung der n-ten Fibonacci-Zahl

4.5. Knoten zur Handhabung von Datenstrukturen

Der Objektmanager der ADAM-Architektur präsentiert den Speicher als eine Menge von Objekten verschiedener Länge. Objekte beliebiger Länge können alloziert werden. Zu jedem Objekt wird die Anzahl Referenzen gezählt und das Objekt wird freigegeben, sobald die Anzahl auf Null fällt. Die einzelnen Felder der Objekte können schreibend oder lesend über eine Objektidentifikation, die bei der Allokation vergeben wird, und den Index des Feldes im Objekt zugegriffen werden. Neben Einzelfeldern können auch ganze Teile von Objekten, spezifiziert durch Quellenobjekt, -index, Länge, Zielobjekt und -index, kopiert werden. Diese Operationen haben ihre entsprechenden Knoten in den Datenflussgraphen. Fig. 4.11 gibt eine Uebersicht über die vorhandenen Knoten, deren Funktion weiter unten erklärt ist.

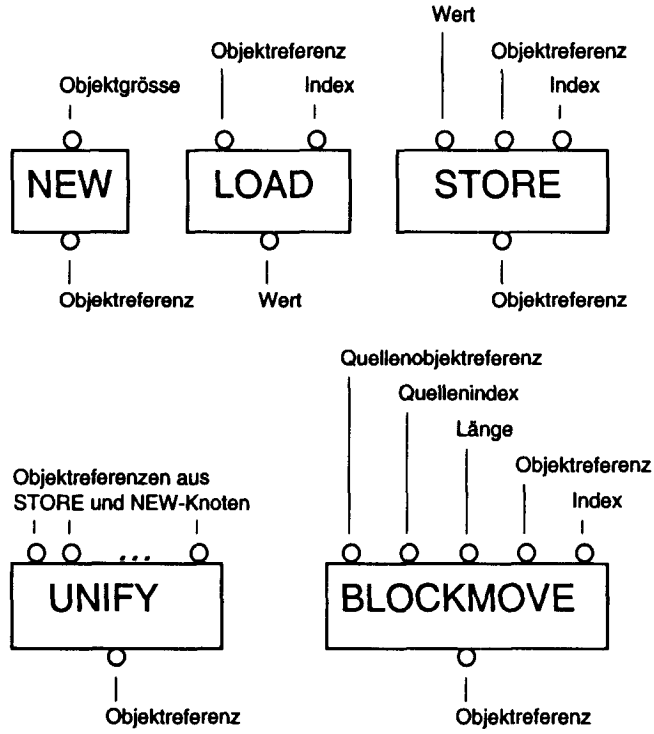


Fig. 4.11: Die Knoten zur Handhabung von Datenstrukturen

Es gibt drei verschiedene NEW-Knoten entsprechend der drei von der Maschine unterstützten Klassen von Objekten (vgl. Kapitel 2 "Die ADAM-Architektur"), die an ihrem Eingang eine Objektgrösse erwarten und deren Ausgang ein neu allozier-

tes Objekt der gewünschten Grösse liefert. Sie sind die einzigen nicht funktionale Knoten in den Datenflussgraphen[1], weil die mehrfache Allokation eines Objekts natürlich nicht immer die gleiche Objektreferenz liefert.

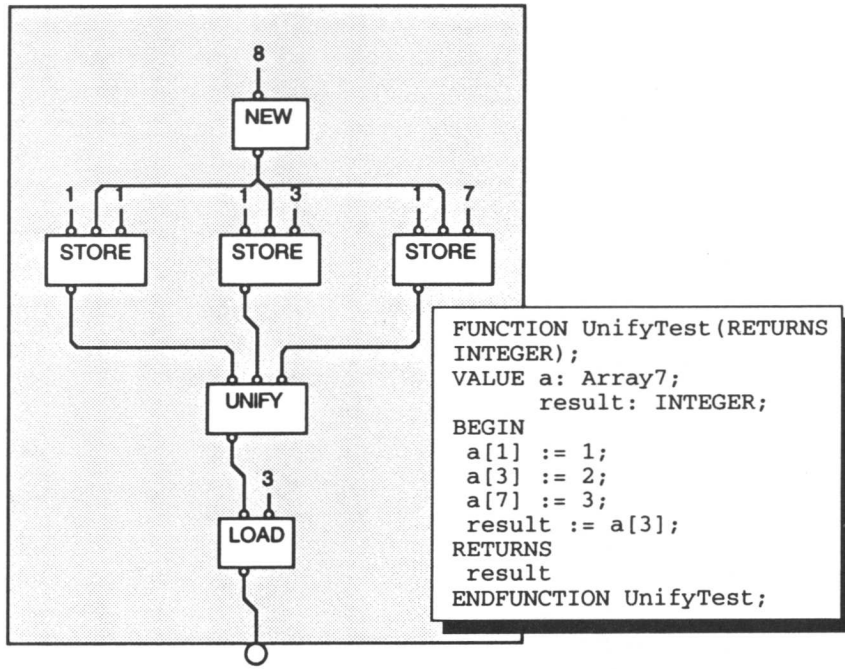


Fig. 4.12: Beispiel: Anwendung von UNIFY-, STORE- und LOAD-Knoten

Dann gibt es den LOAD-Knoten, der mit Hilfe von Objektidentifikation und Index am Ausgang den Wert an der bezeichneten Stelle im Objekt liefert. Entsprechend dazu ist der STORE-Knoten spezifiziert: An den Eingängen erwartet er einen Wert, eine Objektreferenz und einen Index, speichert dann den Wert an die richtige Stelle im Objekt und liefert an seinem Ausgang die gleiche Objektreferenz zurück. Die Resultatobjektreferenz dient zwei Zwecken: Wird eine ungültiges Resultat zurückgegeben, so konnte die Operation nicht ausgeführt werden, weil entweder der Index ausserhalb der Objektgrenzen lag oder bereits ein Wert an der entsprechenden Stelle im Objekt geschrieben wurde. Gültige Resultat-Objektreferenzen aus STORE-Knoten gemeinsam mit dem UNIFY-Knoten dienen

---

[1]) Der NEW-Knoten darf also beispielsweise nicht als gemeinsamer Unterausdruck wegoptimiert werden, wenn schon ein identischer NEW-Knoten existiert.

ausserdem zur Synchronisation von mehreren parallelen Schreiboperationen. Solange nicht gelesen wird, können Zuweisungen auf Objekte in beliebiger Reihenfolge gemacht werden. Vor dem ersten Lesen muss dann das Ende aller Zuweisungen abgewartet werden. Das folgende MFL-Beispiel zeigt die Anwendung des UNIFY-Knotens in Kombination mit STORE-Knoten.

Der BLOCKMOVE-Knoten entspricht im Wesentlichen dem STORE-Knoten. Nur wird anstelle eines zu speichernden Wertes ein Tripel bestehend aus Quellenobjektreferenz, -index und Länge angegeben.

Zusätzlich zu den normalen Ausgängen eines Graphen gibt es noch *Deallokationsausgänge*. Auf die Deallokationsausgänge werden die Referenzen auf jene Objekte geführt, die nicht mehr gebraucht werden. So wie der NEW-Knoten eine Quelle für neue Objekte darstellt, entspricht ein Deallokationsausgang einer Senke für nicht mehr gebrauchte Objekte. Auf die Frage, unter welchen Bedingungen ein Objekt dealloziert werden darf, wird in einem späteren Kapitel (sh. Kap. 6 "Effiziente Verwaltung von Datenstrukturen") noch eigens eingegangen. Die Deallokationsausgänge werden in den Datenflussgraphen normalerweise nicht dargestellt.

4.6. Datenstrukturen zur Repräsentation der Graphen

Die oben abstrakt beschriebenen Graphen werden vom MFL-Compiler mit Hilfe folgender MODULA-2 [Wirth85] Datenstrukturen gespeichert:

Die Knoten werden durch je einen Variantenrecord vom Typ NODE repräsentiert. Man sieht die drei Varianten zusammengesetzter Knoten (`compound`), einfacher Knoten (`simple`) und Graph (`graph`). Wie oben erwähnt, kann ein Graph mit Hilfe seines Umgebungsknotens dargestellt werden. Dank dieser Technik kann für den Graphen und die Knoten die gleiche Datenstruktur verwendet werden.

```
NodeRef = POINTER TO Node;
Node = RECORD
         topConns        : ConnectorRef;
         bottomConns     : ConnectorRef;
         CASE nodeClass: NodeClassType OF
           compound:
             cAction     : NodeType;
             graphs      : NodeRef
         | simple:
             sAction     : NodeType
         | graph:
             nextGraph   : NodeRef;
             toBeDisposed: ConnectorRef;
         END;
         marked          : LONGINT;
       END;
```

Fig. 4.13: MODULA-2-Datenstruktur für die Knoten

Beschreibung der einzelnen Felder:

- *topConns:* Liste der Eingänge bei Knoten bzw. Ausgänge bei Graphen (Datensenken).
- *bottomConns:* Liste der Ausgänge bei Knoten bzw. Eingänge bei Graphen (Datenquellen)
- *nodeClass:* Unterscheidung zwischen einfachen oder zusammengesetzten Knoten bzw. Graphen.
- *cAction:* Funktion des zusammengesetzten Knotens (Aufzählungstyp).
- *graphs:* Liste der Subgraphen des zusammengesetzten Knotens.
- *sAction:* Funktion eines einfachen Knotens (Aufzählungstyp).
- *nextGraph:* Referenz auf den nächsten Graphen in einer Untergraphliste.
- *toBeDisposed:* Deallokationseingänge des Umgebungsknotens für die Speicherverwaltung.
- *marked:* Markierungsfeld (wird von verschiedenen Algorithmen gebraucht, um den Graph zu traversieren)

Die Kanten des Datenflussgraphen bestehen aus je einem Eingang (topConn) und einem Ausgang (bottomConn)[1]. Die Bedingung, dass ein Eingang nur mit einem Ausgang verknüpft sein darf, spiegelt sich in der Datenstruktur an der Tatsache, dass ein Eingang nur auf einen Partner (toConn) zeigen kann. Dass man bei Eingängen auf den Partnerknoten (toNode) verweist, wäre eigentlich nicht nötig, da man dieselbe Information über den Zugriff (toConn^.myNode) erhalten könnte. Allerdings nimmt dieses Zusatzfeld im Variantenrecord keinen zusätzlichen Platz weg und beschleunigt die meisten Algorithmen, die den Graph traversieren. Anstelle eines Ausgangs kann sich ein Eingang auch auf eine Konstante (constant) beziehen. Die Konstante kann irgend einen 32-Bit-Wert enthalten. Aus praktischen Gründen kann dieser Wert sowohl als Fliesspunktzahl (realValue), als auch als Ganzzahl (value) zugegriffen werden.

```
ConnectorRef = POINTER TO Connector;
Connector    = RECORD
                 nextConn        : ConnectorRef;
                 portNr          : INTEGER;
                 CASE connClass: ConnectorClassType OF
                   topConn:
                     toNode      : NodeRef;
                     toConn      : ConnectorRef
                 | bottomConn:
                     myNode      : NodeRef;
                     nrOfRef     : INTEGER;
                 | constant:
                     CASE: BOOLEAN OF
                       TRUE : value      : LONGINT
                     | FALSE: realValue  : REAL;
                     END;
                 END;
                 marked           : Mark;
               END;
```

Fig. 4.14: MODULA-2-Datenstruktur für die Kanten

Beschreibung der einzelnen Felder:

- *nextConn:* Referenz auf nächstes Element in einer Verbindungsliste.

---

[1] Wie schon mehrfach erwähnt, sind bei Graphen topConn's Ausgänge und bottomConn's Eingänge, da sie durch ihren Umgebungsknoten dargestellt sind.

- *portNr:* Position der Verbindung innerhalb der Verbindungsliste (Numerierung beginnt mit 1)
- *connClass:* Art der Verbindung. Es wird zwischen Eingängen, Ausgängen und Konstanten unterschieden.
- *toNode:* Herkunftsknoten dieser ankommenden Verbindung.
- *toConn:* Entsprechender Ausgang an diesem Knoten.
- *myNode:* Knotenzugehörigkeit für Ausgänge.
- *nrOfRef:* Anzahl der Kanten, die sich auf diesen Ausgang beziehen.
- *value:* Wert der Konstante.
- *realValue:* Wert der Konstante für REALs.
- *marked:* Markierungsfeld (wird von verschiedenen Algorithmen gebraucht, um den Graph zu traversieren)

Ein rekursiver Algorithmus zur Traversierung aller Kanten ist einfach zu programmieren. Interessant ist die Tatsache, dass die Knoten nur in einer Richtung, nämlich von den Eingängen zu den Ausgängen, verbunden sind. In einer ersten Version wurden die Verbindungen in beide Richtungen hergestellt. Es zeigte sich aber recht bald, dass sich die statischen Abwärtsverbindungen mit etwas Sorgfalt leicht durch den dynamischen Aufrufstack der rekursiven Traversierungs- und Transformationsfunktionen ersetzen lassen.

Als Alternative zu dieser Art der Darstellung wären auch Adjazenzmatrizen oder eine textuelle Form wie bei IF1 [SkeGla84] in Frage gekommen. An einer textuellen Form wäre hauptsächlich die Möglichkeit einer manuellen Inspektion der Zwischenergebnisse der verschiedenen Uebersetzungsschritte interessant gewesen. Andererseits müsste der Text vor jedem Schritt neu lexikalisch und syntaktisch analysiert werden, was auch für eine einfache Sprache aufwendig gewesen wäre. Als Alternative dazu haben wir ein Werkzeug entwickelt, mit dem sich die Graph-Datenstrukturen bei Bedarf graphisch darstellen lassen [MitMur91]. Somit konnten wir Effizienz und Transparenz verbinden. In Kapitel 9 "Implementation" wird gezeigt, wie sich die Programmierumgebung inklusive das Graphdarstellungs-Werkzeug dem Benutzer darstellt.

4.7. Arbeiten mit ähnlichen Ansätzen

Die ersten Ideen, Programme als Datenflussgraphen darzustellen und eine Sprache zu entwickeln, die sich leicht in solche Graphen übersetzen lässt und diese Graphen direkt als Maschinencode verwendet, stammen von Dennis am MIT [Dennis75]. Er führte die klassischen Elemente von Datenflussgraphen wie Merge-, Gate und

Operatorknoten ein. Er prägte auch den Begriff von "anständigen" ("well behaved") Datenflussgraphen, für Graphen, die, falls sie terminieren[1], an jedem Ausgang genau einen Wert abliefern und den internen Zustand[2] des Graphen wieder herstellen. Er beschreibt auch eine einfache operationelle Semantik für Datenflussgraphen, auf deren Kanten Marken mit Werten fliessen. Im weiteren Sinne sind diese Datenflussgraphen eine Spezialisierung von Petri-Netzen. Einen schönen Ueberblick über das Thema "Klassische Datenflussgraphen" liefert [DavKel82].

Die in diesem Kapitel gezeigten Zwischengraphen lehnen sich stark an die in IF1 [SkeGla84], der Graph-Darstellung von SISAL verwendeten Konzepte an. Insbesondere die Idee, zusammengesetzte Knoten mit Subgraphen zur Darstellung von Kontrollflussgraphen zu verwenden, stammt von IF1. Auf der Basis von SISAL und IF1 entstanden eine Reihe von Arbeiten mit ähnlichen Zielsetzungen wie die vorliegende. Einige von ihnen werden noch in späteren Kapiteln zitiert.

Das in diesem Kapitel beschriebene Graph-Format unterscheidet sich dennoch in gewissen Punkten deutlich von IF1:

- Die Verwaltung von Datenstrukturen ist maschinennäher und auf die speziellen Bedürfnisse der ADAM-Architektur angepasst.
- Die zusammengesetzten Knoten funktionieren im einzelnen nicht gleich wie bei IF1.
- Die Menge der primitiven Knoten wurde auf die ADAM-Architektur angepasst.
- Im Gegensatz zu IF1 sind unsere Graphen nicht für eine textuelle Repräsentation, sondern für die effizientere, direkte Repräsentation im Speicher (vgl. Abschnitt 4.6 "Datenstrukturen zur Repräsentation der Graphen") ausgelegt. Die gleichen Datenstrukturen können allerdings auch für die Darstellung von IF1-Graphen verwendet werden.

---

1) Es gibt selbstverständlich auch Graphen, die nicht terminieren. Das Halteproblem ist auch in der Datenflusstheorie nicht gelöst.

2) Mit "internem Zustand" ist die Belegung der Kanten mit Marken gemeint.

5. Partitionierung von Datenflussgraphen in Codeblöcke

In diesem Kapitel wird das Problem der Partitionierung von Datenflussgraphen in Codeblöcke diskutiert. Ziel dieser Partitionierungsphase ist es, den Kommunikationsaufwand gegenüber der Rechenarbeit zu balancieren.

In einem ersten Teil wird das Partionierungsproblem allgemein beschrieben und verschiedene Gründe dafür angegeben, warum es keinen Sinn macht, für dieses Problem eine im strengen Sinne optimale Lösung zu suchen. Es wird dann kurz eine Heuristik für die Partitionierung eingeführt, die auf die vom Programmierer gewählte Struktur des Programms Rücksicht nimmt. Im nächsten Teil wird gezeigt, wie den einzelnen Teilen des Datenflussgraphen als Grundlage für die Partitionierung Gewichte zugeordnet werden. Dann werden die beiden Teilalgorithmen dieser Heuristik, Expansion zu kleiner Funktionen und Optimierung der Granularität von parallelen Codeblöcken bei FORALL-Schleifen detailliert diskutiert. Am Schluss des Kapitels finden sich Experimente mit dem ADAM-Simulator zur Verifikation der benutzten Konzepte und ein Vergleich mit anderen Forschungsarbeiten zum Thema. Alle in den Experimenten verwendeten MFL-Programme sind im Anhang B wiedergegeben.

5.1. Das Problem und mögliche Lösungen

5.1.1. Das Partitionierungsproblem

Grundsätzlich besteht das Problem der *Partitionierung* darin, alle Knoten des Datenflussgraphen den Codeblöcken zuzuteilen. In einem ersten Schritt wird dazu jedem Knoten des Graphen ein seinem Ausführungsaufwand entsprechendes Gewicht zugeordnet. Die Kanten bekommen ein Gewicht, welches ihrem Kommunikationsaufwand entspricht. Das Kantengewicht ist also für Kanten zwischen Knoten im selben Codeblock gleich Null oder sehr klein und für Kanten über die Codeblockgrenze hinaus gross. Gesucht ist nun eine Partitionierung, die die gesamte *Ausführungszeit* (= Kosten der Knoten und Kanten entlang des längsten Pfads im Graphen) minimiert. Dabei spielt das Verhältnis von Kommunikations- zu Rechenleistung des verwendeten Multiprozessors eine grosse Rolle. Gute Kommunikationsleistung entspricht vergleichsweise niedrigen Kantengewichten bei codeblocküberschreitenden Kanten und umgekehrt. Im Laufe der Partitionierung kann die *Granularität* der Codeblöcke variiert werden: Je grösser die Codeblöcke sind, desto weniger Kommunikation ist nötig, aber desto geringer ist auch ihre Anzahl und damit Parallelität.

Für eine formale Beschreibung des *Partitionierungsproblems* brauchen wir zunächst einige Definitionen. Für jeden Knoten des Graphen G = (N, E, U) müssen die maschinenabhängigen Ausführungskosten bekannt sein. Die *Kostenfunktion* C legt diese fest:

$C: N \rightarrow R^+$, sodass C(n) den Ausführungskosten des Knotens n entspricht

Die *Partitionierungsfunktion* Π für einen Graphen, weist jedem Knoten seine Partition aus den positiven ganzen Zahlen zu.

$\Pi: N \rightarrow Z^+$, sodass $\Pi(n)$ der Partition für den Knoten n entspricht

In unserer Terminologie (vgl. Kapitel 2 "Die ADAM-Architektur"), entspricht Π der Zugehörigkeit eines Knotens zu einem Codeblock. Knoten mit gleichem $\Pi(n)$ gehören zum gleichen Codeblock.

Den Kanten im Graph werden über die Funktion K *Kommunikationskosten* entsprechend dem Typ der über die Kante kommunizierten Daten zugewiesen. Im ADAM-Modell entsprechen diese Kosten dem Aufwand für nicht lokale Objekt-Zugriffe. Für die Kostenfunktion gelte:

$K: E \rightarrow R^+$, sodass K(e) den Kommunikationskosten für die Kante e entspricht

Zu jeder Partitionierungsfunktion Π auf einem Graphen G lassen sich ein *Partitionengraph* $G_\Pi = (N_\Pi, E_\Pi)$ und die beiden Kostenfunktionen $C_\Pi: N_\Pi \rightarrow R^+$ und $K_\Pi: E_\Pi \rightarrow R^+$ nach folgenden Regeln definieren:

1. Jeder durch die Partitionierungsfunktion erzeugten Partition entspricht ein Knoten in N_Π: $Z^+ \supset N_\Pi$ im Partitionengraph. Alle Knoten in der gleichen Partition werden sequentiell ausgeführt. Die Kosten für einen Knoten im Partitionengraph errechnen sich deshalb einfach als Summe aller Kosten der Knoten, die zur Partition gehören:

   $$C_\Pi: N_\Pi \rightarrow R^+, \text{ sodass } C_\Pi(i) = \Sigma_{(n \in N \wedge \Pi(n) = i)} C(n) \qquad (5.1)$$

2. Die Kanten zwischen den Partitionen entsprechen der gesamten Kommunikation zwischen Knoten in unterschiedlichen Partitionen. Zwischen zwei Partitionen existiert also eine Kante, falls zwischen zwei Knoten der beiden Partitionen im ursprünglichen Graph eine Kante besteht.

$$E_{\Pi} = \{(i,j) \mid (m, k, n, l) \in E \wedge \Pi(m) = i \wedge \Pi(n) = j\} \quad (5.2)$$

Die Kosten einer Kante im Partitionengraph entsprechen der Summe aller ursprünglichen Kantenkosten der darauf abgebildeten ursprünglichen Kanten:

$$K_{\Pi}: E_{\Pi} \rightarrow R^{+}, \text{ sodass } K_{\Pi}((i, j)) = \Sigma_{((m, k, n, l) \in E \wedge \Pi(m) = i \wedge \Pi(n) = j)} K((m, k, n, l)) \quad (5.3)$$

Die Ausführungszeit für eine Partitionierung Π lässt sich nun als Länge des *längsten Pfades* in G_{Π} ausdrücken, wobei sich die Länge eines Pfades als Summe aller Knoten- und Kantenkosten entlang des Pfades ergibt. Damit der längste Pfad eindeutig messbar ist, darf auch der Partitionengraph keine Zyklen enthalten. Um dieses Ziel zu erreichen, muss eine Partitionierung *konvex* sein [Sarkar89]. Konvexität bedeutet, dass, falls zwei Knoten zur gleichen Partition gehören, auch alle Knoten auf den Pfaden zwischen diesen beiden zur selben Partition gehören. Formal lässt sich die *Konvexitätsbedingung* für eine Partitionierung folgendermassen definieren:

$$m, n, o \in N \wedge \Pi(m) = \Pi(n) \wedge m \leq^{*} o \leq^{*} n \Rightarrow \Pi(o) = \Pi(n) \quad (5.4)$$

Mit Hilfe aller gezeigten Definitionen (5.1-4) lässt sich das *Partitionierungsproblem* folgendermassen definieren:

Finde zu einem gegebenen Graphen G mit architekturabhängigen Kostenfunktionen C und K jene Partitionierung Π, bei deren Partitionengraph der längste Pfad minimal ist.

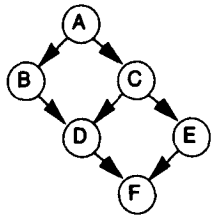
In der ganzen Definition des Partitionierungsproblems ist bisher die Anzahl der Prozessoren nicht aufgetaucht. Nimmt man den kritischen Pfad im Partitionengraph als Optimierungskriterium, ist dies auch nicht nötig. Man bekommt dann die optimale Partitionierung für eine unbeschränkte Anzahl Prozessoren.

Das Problem lässt sich auch für eine vorgegebene, begrenzte Anzahl Prozessoren formulieren, indem man die einzelnen Knoten des Partitionengraphen einer begrenzten Anzahl Prozessoren zuweist. Insbesondere bei kleinen Prozessorzahlen und grossen Programmgraphen dürfte die optimale Partitionierung für unendlich viele Prozessoren viel zu fein sein.

Die Figur 5.1 zeigt eine optimale Partitionierung für einen trivialen Beispielgraphen:

Ursprünglicher Graph G

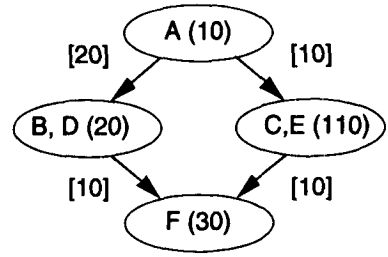
n	C(n)
A	10
B	50
C	100
D	10
E	10
F	30



K(e) = 10 für alle Kanten

Kritischer Pfad: 180

Partitionengraph G_{Π}



(Knotenkosten), [Kantenkosten]

Kritischer Pfad: 170

Fig. 5.1 Optimale Partionierung eines einfachen Graphen

Im folgenden Abschnitt werden wir sehen, dass diese Formulierung des Partitionierungsproblems eine Reihe von Randbedingungen vernachlässigt und sich somit nicht für eine praktische Lösung eignet.

5.1.2. Zum Sinn und zur Lösbarkeit des Partitionierungsproblems

Auf den ersten Blick ist das Partitionierungsproblem ein klassisches Optimierungsproblem aus der Graphentheorie. Alle Parameter sind gegeben, und das Optimierungskriterium ist festgelegt. Wie viele andere Optimierungsprobleme ist das Problem der optimalen Partitionierung *NP-vollständig* (Beweis bei [Sarkar89]), so dass sich die Anwendung eines entsprechenden Algorithmus auf Datenflussgraphen mit ihren Tausenden von Knoten aus Effizienzgründen verbietet. Man muss also hier eine *Heuristik* anwenden.

Ein zweites Problem liegt darin, dass die *Knotengewichte* nicht einfach zur Kompilationszeit berechenbar sind, da sich das Gewicht für jeden Knoten als Produkt der Kosten einer einmaligen Ausführung und der Anzahl Ausführungen zusammensetzt. Dieses Problem ist gravierend, wird doch in typischen Programmen der grösste Teil der Zeit in einigen wenigen Instruktionen, typischerweise in inneren Schleifen verbracht. Um ein Programm optimal partitionieren zu können, müsste man es zuerst mit den entsprechenden Eingabedaten laufen lassen, dabei ein Ausführungsprofil erstellen und dann erst die definitive Partitionierung vornehmen. Prinzipiell gilt dann eine solche Partionierung nur für einen bestimmten Satz von Eingabedaten, wobei man einräumen muss, dass in vielen praktischen Anwendungen die Eingabedaten keinen wesentlichen Einfluss auf die Laufzeitcharakteristik eines Programms haben.

Ausserdem ist für eine gut balancierte, parallele Architektur (wie sie durch die ADAM-Architektur verkörpert wird) bei der das Starten paralleler Aktivitäten nur einen kleinen Aufwand verursacht, das Minimum der Zielfunktion ziemlich *flach*, wie die folgende Rechnung zeigen wird. Nehmen wir an, eine Aufgabe, die insgesamt Kosten von t_{seq} verursacht, soll auf p Prozessoren durchgeführt werden. Die Rechnung habe die Struktur einer FORALL-Schleife und sei somit beliebig in n parallele Stücke gleicher Grösse aufteilbar[1]. Das Starten jedes einzelnen parallelen Teils (Codeblocks) sei mit einem konstanten Zusatzaufwand $t_{overhead}$ verbunden. Falls wir eine perfekte Lastverteilung unter den Prozessoren annehmen, errechnet sich der Aufwand für die Ausführung der ganzen Aufgabe auf p Prozessoren folgendermassen:

$$t_{total} = \lceil n/p \rceil * (t_{seq} / n + t_{overhead}) \qquad (5.5)$$

Figur 5.2 zeigt diese Funktion für p = 32, t_{seq} = 1000000, $t_{overhead}$ = 10, 100, 1000 in Abhängigkeit von n.

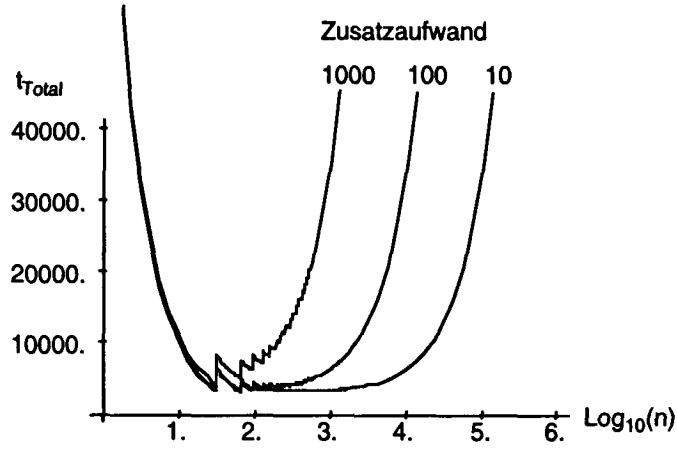


Fig. 5.2 Abhängigkeit des Partitionierungsoptimums von den Maschinenparametern

Trivialerweise ist t_{total} für jedes $t_{overhead}$ bei 32 optimal. Aus verschiedenen Gründen kann es nun möglich sein, dass n nicht gleich p gewählt werden kann. Mögliche Gründe dafür sind:

---

[1]) Diese Annahme ist zwar nicht vollständig korrekt, da eine Iteration in einer FORALL-Schleife nicht mehr weiter teilbar ist, stimmt aber näherungsweise für sehr grosse Schleifen mit vielen Iterationen.

- Mehr als ein Codeblock pro Prozessor wird gebraucht, um die Speicherlatenz zu verstecken (vgl. Kapitel 2 "Die ADAM-Architektur")
- Aus Diskretisierungsgründen lässt sich n nicht als Vielfaches von p wählen.
- Die Lastverteilung ist nicht perfekt.
- Der Rechenaufwand pro Codeblock ist nicht a priori klar (z. B. Mandelbrot-Rechnung)

Die Funktion in Figur 5.2 zeigt, dass das Optimum für die Partitionierung stark architekturabhängig ist. Für "schlechte" Parallelmaschinen, auf denen das Generieren der Parallelität viel kostet (hoher Zusatzaufwand), ist das Optimum für die beste Partitionierung viel schärfer als für "gute" Maschinen mit kleinem Zusatzaufwand. Je schlechter die Rechnerarchitektur ist, desto höher sind die Anforderungen an den Compiler, um brauchbaren Code zu liefern; eine Feststellung, die sicherlich nicht nur auf das Partitionierungsproblem zutrifft.

Zudem ist diese Funktion sehr unstetig, wenn n ungefähr gleich der Anzahl Prozessoren ist: In der Figur 5.2 sind *Diskretisierungseffekte* sichtbar Allein aus diesem Grund ist es sinnvoll, n deutlich grösser als die Anzahl Prozessoren zu wählen.

Weiter zu berücksichtigen ist, dass sich der Programmierer bei der Aufteilung seines Programms in Funktionen in der Regel etwas überlegt, und dass diese Ueberlegungen auch einen Einfluss auf die vom Compiler gewählte Partionierung haben sollten. Beim *"Debugging"* sollte die *Rückabbildbarkeit vom Code zum Programmtext* möglichst gewährleistet sein, was sicher schwierig ist, falls der Compiler aus Optimierungsgründen das Programm in völlig neue Teile partitioniert. Die vom Benutzer gewählte Programmstruktur sollte sich auch im Code widerspiegeln.

Zudem wurden bei der Definition des Partitionierungsproblems nur konvexe Partitionierungen zugelassen. Diese Beschränkung entspricht in der Realität einem sehr primitiven Laufzeitmodell, dessen Aktoren nicht unterbrochen werden können. Codeblöcke auf der ADAM-Architektur unterstützen ein flexibleres Ausführungsmodell.

Zusammenfassend gesagt, lässt sich das Partitionierungsproblem in der Theorie zwar sehr schön formulieren, seine praktische Lösung scheitert jedoch daran, dass einerseits die notwendigen Parameter gar nicht in der genügenden Genauigkeit festliegen und andererseits der Optimierungsprozess ausserordentlich aufwendig ist.

Diese Gründe führten zu dem im folgenden Abschnitt beschriebenen, einfachen Partitionierungsalgorithmus, der die vom Benutzer gewählte Partitionierung nur an besonders kritischen Stellen verbessert.

5.1.3. Die gewählte Lösung: Grundpartitionierung und Optimierung

Grundsätzlich generiert der MFL-Compiler für jeden Funktionsgraphen einen eigenen Codeblock. Ausserdem wird für den Schleifenkörper einer FORALL-Schleife ein eigener Codeblock generiert. CALL-Knoten entsprechen so direkt den CALL-Instruktionen der ADAM-Architektur, und der Kontrollcode für einen FORALL-Knoten wird um die PARCALL-Instruktion herum aufgebaut, wobei für eine FORALL-Schleife mit n Iterationen n Codeblöcke gestartet werden. Diese Art der Partitionierung ist für den Compiler mit wenig Aufwand zu bewerkstelligen, transparent für den Programmierer und führt in vielen Fällen zu sehr guten Resultaten.

An zwei Stellen kann der Compiler die Partionierung, falls nötig, einfach verbessern, ohne die vorgegebene Programmstruktur zu zerstören: Einerseits lassen sich Funktionsgraphen, deren Aufruf alleine schon mehr kostet als ein bestimmter, konstanter Bruchteil ihrer zu erwartenden Ausführungszeit, direkt in den aufrufenden Graphen *expandieren*, falls sie nicht Teil einer Rekursion sind. Andererseits muss bei der Uebersetzung einer FORALL-Schleife nicht unbedingt für jede Iteration ein eigener Codeblock aufgerufen werden. Jeder parallele Codeblock kann auch eine ganze Anzahl sequentieller Iterationen umfassen. Falls die gesamte Anzahl Iterationen oder die Anzahl vorhandener Prozessoren nicht schon zur Kompilationszeit bekannt ist, kann diese Art Partitionierung leicht auch erst zur Laufzeit vorgenommen werden. Es lohnt sich, diesen Aufwand bei FORALL-Schleifen zu treiben, da viele parallele Programme dort den grössten Teil der Arbeit verrichten.

In den Abschnitten 5.3 "Funktionsexpansion" und 5.4 "Forall-Schleifen" werden die Details zu den zwei Methoden erläutert.

5.2. Kostenzuweisung

Basis für die Partitionierung der Graphen ist eine *Kostenzuweisung* an die Graphen, damit überhaupt Kostenverhältnisse abgeschätzt werden können. Die im Abschnitt 5.1.1 "Das Partitionierungsproblem" definierte Kostenfunktion C, welche jedem Knoten eines Graphen seine Kosten zuweist, soll für unsere Datenflussgraphen konkret definiert werden. Entsprechend der Struktur der Graphen ist die Kostenzuweisung rekursiv aufgebaut.

Jedem *primitiven Knoten* werden entsprechend seinem *Knotentyp* (z. B. Addition, LOAD, etc.) und dem Aufwand zur Ausführung der entsprechenden Instruktion(-en) auf dem Zielprozessor *konstante Kosten* zugewiesen. Die Kostenfunktion C ist also für primitive Knoten als Tabelle von Kosten für jeden Knotentyp aufgebaut. Die Kosten eines Knotens entsprechen ungefähr seiner Ausführungszeit auf der ADAM-Maschine, wobei diese nicht für alle Instruktionen statisch vorhersagbar ist. Wie schon in Abschnitt 5.1 "Das Problem und mögliche Lösungen" erwähnt wurde, ist eine absolute Präzision der Kostenfunktion in unserem Algorithmus gar nicht notwendig. Wichtig ist die relative Vergleichbarkeit der Kosten verschiedener Programmteile.

Die Kosten für einen gesamten *Graphen* entsprechen der Summe der Kosten seiner Knoten:

$$C(G) = \sum_{n \in N} C(n) \text{ für } G = (N, E, U) \tag{5.6}$$

Zusammengesetzte Knoten bestehen aus einer Menge von Subgraphen (vgl. Abschnitt 4.3 "Zusammengesetzte Knoten"). Dementsprechend setzen sich die Kosten für einen zusammengesetzten Knoten als gewichtete Summe der Subgraph-Kosten plus einen gewissen Zusatzaufwand für Flusskontroll-Instruktionen zusammen. Im folgenden werden die Kostenfunktionen für die verschiedenen Klassen von zusammengesetzten Knoten angegeben, wobei aus Gründen der Uebersichtlichkeit die Flusskontroll-Zusatzkosten weggelassen wurden:

Im Falle des *IF-THEN-ELSE-Knotens* mit n Alternativen, von denen jede mit Wahrscheinlichkeit p_i auftritt[2], werden die Kosten als gewichtete Summe aller TEST-Graphen und aller THEN-Graphen berechnet. Bei den TEST-Graphen müssen zudem Kosten für alle bis zur endgültigen Auswahl der Alternative ausgewerteten TEST-Graphen aufsummiert werden. Mit der Restwahrscheinlichkeit gewichtet kommen noch die Kosten für den ELSE-Graphen zur Summe (vgl. Formel 5.7).

---

2) Es ist vorgesehen, dass der Benutzer die Wahrscheinlichkeiten für die verschiedenen Alternativen selbst setzen kann. Automatisch nimmt der Compiler an, dass alle Alternativen gleich wahrscheinlich sind.

$$C(IF-THEN-ELSE) = \sum_{i=1,n} p_i \sum_{j=1,i} C(TEST_j) + \sum_{i=1,n} p_i C(THEN_i) + (1 - \sum_{i=1,n} p_i) C(ELSE) \quad (5.7)$$

Bei der Ausführung einer Schleife mit Abbruchbedingung werden der INIT- und der RETURNS-Graph einmal am Anfang und am Ende der Schleife ausgeführt. Der TEST-und der BODY-Graph werden hingegen für jede Iteration wiederholt. Entsprechend werden die Kosten für einen *LOOP(A/B)-Knoten* wie folgt berechnet:

$$C(LOOP) = C(INIT) + n_{it}[C(BODY) + C(TEST)] + C(RETURNS) \quad (5.8)$$

Bei allen Kostenfunktionen für Schleifen tritt die Anzahl der Iterationen n_{it} als Faktor auf. Der Compiler nimmt in der Regel einen vorgewählten Wert für n_{it} an. Der Benutzer hat aber die Möglichkeit, für jede Schleife diesen Wert speziell zu setzen, um die Kostenabschätzung zu verbessern. Bei FOR- und FORALL-Knoten mit konstanten Laufbereichen für die Indices lässt sich n_{it} statisch vorhersagen, was vom Compiler auch getan wird.

Die Kosten für den *FOR-Knoten* berechnen sich entsprechend:

$$C(FOR) = C(INDEX) + C(INIT) + n_{it} C(BODY) + C(RETURNS) \quad (5.9)$$

Die Kosten für den *FORALL-Knoten* (vgl. Formel 5.10) berechnen sich im Prinzip analog zu den Kosten für den FOR-Knoten. Der Unterschied besteht einzig darin, dass bei diesem Knoten der Zusatzaufwand für die Flusskontrolle nicht mehr vernachlässigt werden kann, da das Aufstarten paralleler Codeblöcke inklusive Parameterübergabe eine relativ aufwendige Angelegenheit ist. Dabei verteilt sich dieser Zusatzaufwand auf zwei Teile: Einen *externen Teil* (c_{extern}), der für die ganze Schleife nur einmal, und einen *internen Teil* (c_{intern}), der für jede Iteration ausgeführt werden muss. Im folgenden Abschnitt 5.4 "FORALL-Schleifen" wird noch im Detail auf diese Kosten eingegangen.

$$C(FORALL) = C(INDEX) + C(INIT) + c_{extern} + n_{it}\left[C(BODY) + c_{intern}\right] + C(RETURNS) \quad (5.10)$$

Diese Kostenbewertung ist nicht absolut präzise, da die Kostenfunktion für die primitiven Funktionen vereinfacht als operationsabhängige Konstante angenommen wurde. Dies ist zwar korrekt für reine ALU-Operationen, stimmt aber nicht für

Objekt-Manager-Operationen, deren Kosten stark von der Grösse des bearbeiteten Objekts, dessen Objektklasse, der Netzwerktopologie, der momentanen Belastung des Netzwerks und der aktuellen Netzwerkdistanz zwischen dem anfordernden Prozessor und dem Prozessor, auf welchem das gewünschte Element tatsächlich liegt, abhängen.

Die meisten dieser Faktoren sind zur Uebersetzungszeit ebensowenig bekannt, wie die tatsächlichen Wahrscheinlichkeiten bei der Auswahl von Alternativen oder die Anzahl Iterationen bei Schleifen. Eine Möglichkeit, dieses Problem zu lösen, besteht darin, Ausführungsprofile eines Programms aufzuzeichnen [Sarkar89]. Man hat dann immerhin eine präzise Kostenanalyse für ein bestimmtes Programm mit bestimmten Daten auf einer bestimmten Maschine.

In unserer Lösung verzichten wir darauf, von der Kostenzuweisung solch präzise Angaben zu fordern, und verwenden aus den bereits gezeigten Gründen einen Partitionierungsalgorithmus, der auf die Unschärfe der Kostenfunktion Rücksicht nimmt.

5.3. Funktionsexpansion

5.3.1. Kosten für Codeblockaufrufe auf der ADAM-Architektur

Parallelität kostet auf jeder Maschine etwas. Bei der ADAM-Architektur zeigen sich diese Kosten beim Codeblock-Aufruf: Um einen neuen Codeblock zu starten (vgl. 2 "Die ADAM-Architektur"), muss zuerst ein Objekt für die Eingabe-Parameter (*In-Objekt*) alloziert und mit den entsprechenden Werten gefüllt werden. Innerhalb des neu gestarteten Codeblocks müssen diese Werte wieder ausgepackt werden. Umgekehrt müssen am Ende des aufgerufenen Codeblocks alle Resultate in ein Objekt für die Resultate (*Out-Objekt*) verpackt und im aufrufenden Codeblock ausgepackt werden. Der Grund für diese Art der Parameterübergabe liegt darin, dass das *Token*, die verschiebbare, dynamische Instanz eines Codeblocks, nur beschränkt Platz bietet. In der heutigen ADAM-Architektur [Maquel92] sind zwei Worte vorgesehen: eines für die Referenz auf das In-Objekt und eines für die globale Adresse des Frameregisters für das Resultat. Einfacher ist das ganze Verfahren für Codeblöcke, die nur einen einzelnen Eingabe- oder Ausgabeparameter haben. Man kann dann die Parameter direkt im Token übergeben, ohne sie zuerst in In- bzw. Out-Objekte zu verpacken.

Das folgende Beispiel in Tabelle 5.1 illustriert anhand eines MFL-Programmfragments und des generierten Codes im aufrufenden Codeblock den Aufrufmechanis-

mus für eine Funktion mit drei Eingangsparametern und zwei Resultaten. Zuerst wird ein In-Objekt alloziert (NEWOBJ)und mit den Parametern gefüllt. Mit der Instruktion WAIT2 wird auf den Abschluss aller asynchronen STO-Anweisungen gewartet und anschliessend die Funktion mit dem In-Objekt als Parameter aufgerufen. Das Resultat des Funktionsaufrufes ist ein Out-Objekt, aus welchem die einzelnen Resultate mit LD-Operationen ausgepackt werden. Zum Schluss wird das Out-Objekt wieder freigegeben (DISPOSE). Das Auspacken des In-Objekts und das Verpacken der Resultate ins Out-Objekt geschieht im aufgerufenen Codeblock genau gegengleich.

MFL-Ausschnitt	**ADAM-Code**
first, maxFirst := MergeSort(low, mid, stream) (wobei sich im nebenstehenden Code die Parameter "low", "mid" und "stream" am Anfang in den Register r1, r2 und r3 befinden und die Resultate nach dem Auspacken als Register r1 und r2 erscheinen)	r4 := NEWOBJ(4) r5 := STO(r4, 1, r1) r6 := STO(r4, 2, r2) r4 := STO(r4, 3, r3) WAIT2(r5, r6) r4 := CALL(MODREF, 4, r4) r1 := LD(r4, 1) r2 := LD(r4, 2) DISPOSE(r4)

Tabelle 5.1: Beispiel für Ein- und Auspacken der In- und Out-Objekte in der ADAM-Architektur

Die Kosten für einen Codeblockaufruf mit dem gezeigten Mechanismus lassen sich mit Hilfe der im Abschnitt 5.2 "Kostenzuweisung" definierten Kostenfunktion C ausdrücken. Es entstehen dabei *externe Kosten* (c_{extern}) beim aufrufenden Codeblock und *interne Kosten* (c_{intern}) beim aufgerufenen Codeblock entsprechend den zusätzlich auszuführenden Instruktionen auf den beiden Ebenen. Der folgende Satz von Formeln zeigt diese Kosten für einen Codeblockaufruf mit n_{in} Eingabe-Parametern und n_{out} Ausgabewerten:

Kosten für	$n_{in} \leq 1 \wedge n_{out}=1$	$n_{in} \leq 1 \wedge n_{out}>1$	$n_{in}>1 \wedge n_{out}=1$	$n_{in}>1 \wedge n_{out}>1$
c_{extern} =	C(CALL)	C(CALL) + $n_{out} \times$ C(LD) + C(DISPOSE)	C(NEWOBJ) + $n_{in} \times$ C(STO) + C(CALL)	C(NEWOBJ) + $n_{in} \times$ C(STO) + C(CALL) + $n_{out} \times$ C(LD) + C(DISPOSE)
c_{intern} =	C(STF) + C(ENDCB)	C(NEWOBJ) + $n_{out} \times$ C(STO)	$n_{in} \times$ C(LD) + C(DISPOSE)	$n_{in} \times$ C(LD) + C(DISPOSE) + C(NEWOBJ) + $n_{out} \times$ C(STO)

Tabelle 5.2: Externe und interne Kosten für einen Funktionsaufruf

5.3.2. Expansion von zu teuren Funktionsaufrufen

Der Partitionierungsprozess läuft nun so ab, dass, ausgehend von den Blättern im *Aufrufgraph der Funktionen*, also jenen Funktionen, welche keine weiteren aufrufen, jeder Funktionsgraph und seine Untergraphen traversiert werden. Mit Aufrufgraph ist ein Graph gemeint, in welchem jeder Funktion ein Knoten und jedem Funktionsaufruf eine gerichtete Kante von der aufrufenden zur aufgerufenen Funktion entspricht. Die Traversierung von den Blättern her hat zum Zweck, dass kein Funktionsgraph traversiert wird, bevor nicht alle Funktionen, die von ihm aus aufgerufen werden, traversiert worden sind.

Bei jedem angetroffenen CALL-Knoten werden die Kosten für den Aufruf ($c_{extern} + c_{intern}$) mit den Kosten des aufgerufenen Funktionsgraphen $C(G_{aufgerufen})$ verglichen. Ist ein gewisses Verhältnis g zwischen den *Aufrufkosten* (*Kommunikationskosten*) und den Graphkosten der aufgerufenen Funktion (*Arbeitskosten*) überschritten, wird der Graph der aufgerufenen Funktion direkt in den Graph der aufrufenden Funktion integriert. Man sagt, der Funktionsaufruf werde *expandiert*. Nach einer Expansion müssen die Graphkosten in der aufrufenden Funktion neu berechnet werden. Die Kommunikationskosten fallen dabei weg, dafür wird das Programm "sequentieller". Formal sieht das Kriterium für die Expansion eines Funktionsaufrufs so aus:

$$\left[g > \frac{c_{intern} + c_{extern}}{C(G_{aufgerufen})} \right] \Rightarrow \text{Expansion} \qquad (5.11)$$

Besondere Beachtung verdient die Tatsache, dass durch *rekursive Funktionsaufrufe* der Aufrufgraph eines Programms durchaus zyklisch werden kann. Beginnt man aber, rekursiv aufgerufene Funktionen zu expandieren, so wird der Graph potentiell unendlich gross. Der MFL-Compiler expandiert deshalb rekursive Funktionen

nicht[3]. Indirekt rekursive Funktionsaufrufe werden, falls nötig, bis zu direkt rekursiven Aufrufen expandiert. Die Grenze liegt dort, wo eine Funktion sich direkt selbst aufruft. Da in indirekten Rekursionen die Strategie versagt, von den Blättern im Aufrufgraphen her zu traversieren, muss von jenen Funktionsgraphen, welche zu einer Rekursion gehören und durch eine Expansion verändert werden, eine Kopie des Originals für weitere Expansionen angelegt werden.

Das Problem, *Gruppen von gegenseitig rekursiven Funktionen* im Aufrufgraphen zu identifizieren, entspricht in der Graphentheorie dem Problem, stark zusammenhängende Komponenten zu suchen. Einen guten Algorithmus mit linearer Komplexität zur Lösung dieses Problems findet man in [Tarjan72].

---

3) Mit besonderen Vorkehrungen wäre es zwar möglich, auch rekursive Funktionsaufrufe begrenzt zu expandieren. Möglicherweise wird eine spätere Version des MFL-Compilers über diese Fähigkeit verfügen.

Der Algorithmus für die Expansion der Funktion mit Graph G' = (N', E', U') an der Stelle des Aufrufknotens n_{call} im Graph der aufrufenden Funktion N = (N, E, U) sieht folgendermassen aus:

```
(* Kopiere alle Knoten und Kanten des zu expandierenden Graphen in den Zielgraphen *)
N := N ∪ N'
E := E ∪ E'

(* Verbinde die Knoten an den Eingängen des CALL-Knotens mit jenen an den Ausgängen
des Umgebungsknoten im zu expandierenden Graphen *)
FORALL (n1, z1, ncall, k) ∈ E DO
    e' := (U', k, n2, z2)
    N := N ∪ {(n1, z1, n2, z2)}
    N := N \ {e', (n1, z1, ncall, k)}
END

(* Verbinde die Knoten an den Ausgängen des CALL-Knotens mit jenen an den Eingängen
des Umgebungsknoten im zu expandierenden Graphen *)
FORALL (ncall, k, n2, z2) ∈ E DO
    e' := (n1, z1, U', k)
    N := N ∪ {(n1, z1, n2, z2)}
    N := N \ {e', (ncall, k, n2, z2)}
END

(* Entferne unnötige Knoten aus dem Graphen *)
N := N \ {U', ncall}
```

In Figur 5.3 wird der im Algorithmus dargestellte Vorgang an einem Beispiel illustriert, wobei die Ausgänge der Knoten A, B, C im aufrufenden Graph neu mit den entsprechenden Eingängen der Knoten X und Y im aufgerufenen Graph und die Ausgänge des Knotens Z im aufgerufenen Graph neu mit den Eingängen des Knotens D im aufrufenden Graph verbunden werden.

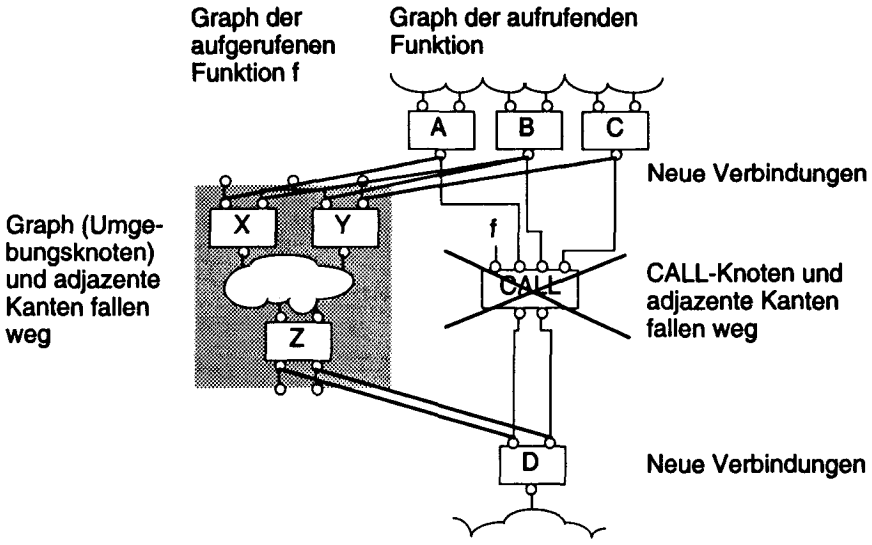


Fig. 5.3: Expansion eines Funktionsaufrufes

Man kann die Konstante g, welche bestimmt, ob ein Funktionsaufruf expandiert wird oder nicht, als minimale *Körnigkeit* oder *Granularität* der Parallelität verstehen. Nach der Partitionierung gibt es keine Funktionsaufrufe einer feineren Körnigkeit als g. Vernünftige Werte für die Konstante g wurden im Laufe dieser Arbeit experimentell ermittelt (vgl. Abschnitt 5.5 "Experimente").

5.4. Forall-Schleifen

Die wichtigste Quelle von Parallelität sind in vielen Algorithmen die FORALL-Schleifen. Wegen ihrer einfachen Struktur und ihrer Bedeutung für die Parallelität in einem Programm verdienen sie besondere Beachtung bei der Partitionierung.

5.4.1. Uebersetzung von Forall-Schleifen

Die einzelnen unabhängigen Iterationen einer FORALL-Schleife werden auf mehrere parallele Codeblöcke verteilt, wie in Figur 5.4 gezeigt, wobei die kleinen Quadrate den Iterationen (FORALL i := [1..11] DO) und die Rechtecke den Codeblöcken entsprechen.

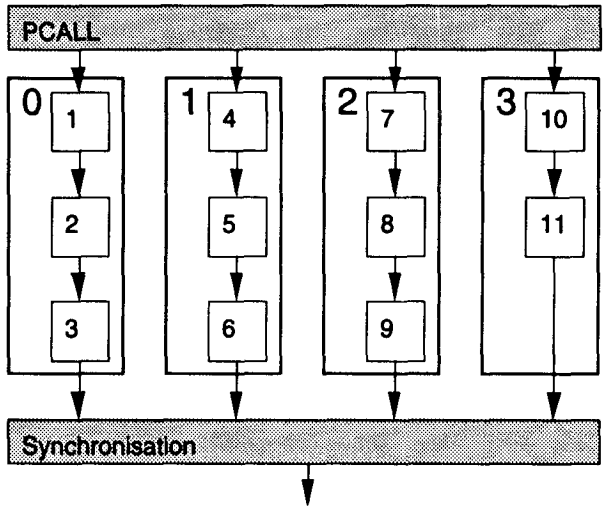


Fig. 5.4: Verteilung der Iterationen auf parallele Codeblöcke im FORALL-Code

Die *Verteilung der Iterationen* auf die parallelen Codeblöcke erfolgt in zusammenhängenden Indexbereichen. Der Indexbereich wird für jeden einzelnen der parallelen Codeblöcke anhand seiner Nummer zur Laufzeit berechnet, wobei die parallelen Codeblöcke eines PARCALLs, beginnend von Null aus, aufsteigende Nummern bekommen.

Falls sich der *Indexbereich* der gesamten FORALL-Schleife von l_{global} bis und mit h_{global} erstreckt und die Iterationen auf insgesamt n_{CB} Codeblöcke aufgeteilt werden, berechnet sich der lokale Indexbereich für den i-ten Codeblock (l_i, h_i) nach den folgenden Formeln:

$$r = (h_{global} - l_{global} + 1)$$

$$l_i = \left\lfloor \frac{i \cdot r}{n_{CB}} \right\rfloor + l_{global} \qquad \text{für } i = 0..n_{CB} \qquad (5.12)$$

$$h_i = \left\lfloor \frac{(i+1) \cdot r}{n_{CB}} \right\rfloor - 1 + l_{global}$$

Um zu zeigen, dass der ganze Indexbereich korrekt auf die Codeblöcke verteilt wird, müssen wir die folgenden Eigenschaften dieser Formeln zeigen:

1. Der erste Codeblock (i=0) fängt bei l_{global} an: $l_0 = l_{global}$
2. Der letzte Codeblock endet bei h_{global}: $h_{n_{CB}-1} = h_{global}$

3. Die Indexbereiche der Codeblöcke reihen sich nahtlos aneinander: $h_{i-1} + 1 = l_i$

Diese Eigenschaften lassen sich für h_{global}, $l_{global} \in Z$ und alle $n_{CB} > 0$ leicht beweisen, wobei der Fall $n_{CB} > r$ dazu führt, dass bei einzelnen Codeblöcken $l_i > h_i$ gilt. Diesen Fall muss man so interpretieren, dass in den betreffenden Codeblöcken gar keine Iterationen ausgeführt werden sollen, was durchaus vernünftig ist, wenn die Anzahl der Codeblöcke grösser ist als die Anzahl der Iterationen.

Bei der Codegenerierung werden die Fälle $l_{global} = 1$ und $l_{global} = 0$ gesondert betrachtet, da sich für diese häufigen Fälle die Formeln und damit auch der Code reduziert. Neben der Kommunikation zum Start der Codeblöcke und zur Uebertragung der Parameter auf die parallelen Codeblöcke ist die Berechnung der lokalen Indexbereiche nach den oben gezeigten Formeln der wesentliche *Zusatzaufwand*, der für jeden Codeblock bei der Kostenzuweisung (vgl. Abschnitt 5.2 "Kostenzuweisung") hinzukommt.

Die *Synchronisation* der parallelen Codeblöcke am Schluss einer FORALL-Schleife erfolgt über ein zählendes Semaphor. Als *zählendes Semaphor* werden direkt die Worte 1 und 2 im In-Objekt des PARCALLs mit den Befehlen SIGNAL und SWAIT der ADAM-Architektur [Maquel92] verwendet. Dies hat den Vorteil, dass pro einstufigen PARCALL nur ein Objekt alloziert werden muss.

In FORALL-Schleifen mit sehr vielen parallelen Codeblöcken auf vielen Prozessoren kann die Generierung der Codeblöcke bei PARCALL und die Synchronisation über ein gemeinsames Semaphor trotz Hardware-Unterstützung zu einem Flaschenhals werden. Aus diesem Grund werden alle FORALL-Schleifen mit mehr als einer minimalen Anzahl Codeblöcken automatisch zu zweistufigen PARCALLs übersetzt. Mit *zweistufigem PARCALL* ist eine Methode gemeint, bei der auf der oberen Stufe eine kleinere Anzahl paralleler Codeblöcke generiert wird, welche ihrerseits mit einem PARCALL der zweiten Stufe die definitive Anzahl Codeblöcke generieren. Die Synchronisation erfolgt analog zuerst lokal und dann global. Die minimale Anzahl von Codeblöcken, ab welcher diese Methode angewendet werden soll, kann vom Benutzer festgelegt werden (vgl. Abschnitt 8.3 "Benutzerschnittstelle"). Jede FORALL-Schleife enthält den Code für beide Methoden, so dass zur Laufzeit entschieden werden kann, ob ein ein- oder zweistufiger PARCALL ausgeführt werden soll.

5.4.2. Berechnung der Anzahl Codeblöcke

Ueber die *Anzahl paralleler Codeblöcke*, in die eine FORALL-Schleife aufgeteilt wird, lässt sich die Granularität der Berechnung nach zwei Kriterien steuern:

Das *erste Kriterium* garantiert, ähnlich wie bei der Expansion von Funktionen, dass das Verhältnis zwischen Zusatzaufwand zum Starten eines Codeblocks (c_{intern})[4] und der im Codeblock ausgeführten Arbeit (n_{it} * C(BODY) ein gewisses Mass nicht überschreitet, wobei n_{it} der minimalen Anzahl sequentieller Iterationen pro Codeblock entspricht, sodass diese Grenze eingehalten wird. Analog zu den Formeln in Tabelle 5.2, können die *internen Zusatzkosten* für die FORALL-Codeblöcke berechnet werden. Dabei ist n_{out} immer gleich 1, und n_{in} gleich der Anzahl Grapheingänge beim BODY-Graphen der FORALL-Schleife. Dazu kommen noch gewisse Kosten, um aus der Nummer des Codeblocks den zu bearbeitenden Unterbereich aus dem gesamten Indexbereich zu berechnen. Die Anzahl Iterationen pro Codeblock sollte daher folgender Gleichung genügen:

$$n_{it} = \frac{c_{intern}}{g \cdot C(BODY)} \tag{5.12}$$

Mit Hilfe des Laufbereichs der Variablen (r) lässt sich daraus die gesamte Anzahl der Codeblöcke ableiten:

$$n_{CB}^{P} = \left\lfloor \frac{r}{n_{it}} + 0.5 \right\rfloor \tag{5.13}$$

Bei grossen FORALL-Schleifen ist die so berechnete Anzahl jedoch häufig viel zu gross. Im Prinzip kann ja pro Prozessor immer nur ein paralleler Codeblock laufen. Zudem braucht es noch auf jedem Prozessor eine Anzahl zusätzlicher Codeblöcke, die aktiviert werden können, wenn bei Speicherzugriffen die Latenz versteckt werden muss (vgl. Kapitel 2 "Die ADAM-Architektur").

Als *zweites Kriterium* für die Anzahl Codeblöcke nehmen wir deshalb an, dass auf jedem Prozessor (p Prozessoren) mindestens k Codeblöcke laufen sollen, wobei die Grösse der Konstante k noch Gegenstand von Experimenten sein wird:

---

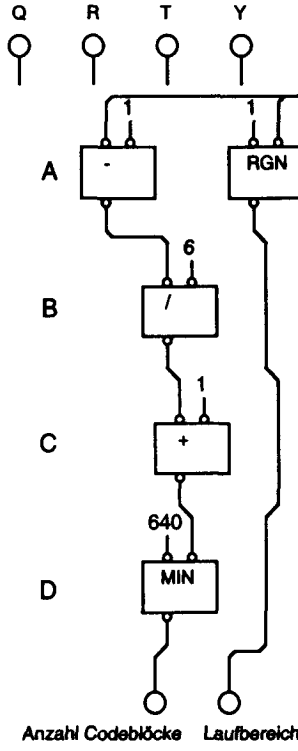
[4]) Die externen Zusatzkosten können in diesem Falle vernachlässigt werden, da sie nur einmal für den ganzen PARCALL auftreten.

$$n_{CB}^{M} = k \cdot p \tag{5.14}$$

Wir haben nun ein *programmabhängiges* und ein *maschinenabhängiges* Maximum für die Anzahl der Codeblöcke. Das kleinere der beiden Maxima bestimmt die tatsächliche Anzahl Codeblöcke für eine FORALL-Schleife:

$$n_{CB} = \min(n_{CB}^{P}, n_{CB}^{M}) \tag{5.15}$$

In den MFL-Datenflussgraphen ist der erste Ausgang des INDEX-Untergraphen im FORALL-Knoten für die Anzahl der Codeblöcke in der Schleife reserviert. An diesen Ausgang kann ein Graph angehängt werden, der diese Anzahl zur Laufzeit berechnet. Ausserdem ist diese Berechnung nach den gezeigten Formeln nicht aufwendig, so dass es sich lohnt, n_{CB} für Laufbereiche, die nicht zur Uebersetzungszeit feststehen, dynamisch zu berechnen. Die *Partitionierung* im MFL-Compiler ist also nicht rein statisch, sondern passt sich im wichtigen Fall der FORALL-Schleifen auch *dynamisch* an.




```
FUNCTION Loop1(i: INTEGER; Q, R, T: REAL;
               Y,Z: OneDimReal
               RETURNS OneDimReal);

VAR k: INTEGER;
    X: OneDimReal;
BEGIN
 RETURNS
  FORALL k := [1,,1] DO
   X[k] := Q + (Y[k] * ((R * Z[k+10])
             + (T * Z[k+11])));
  RETURNS X
 ENDFORALL
ENDFUNCTION Loop1;
```

Fig. 5.5: INDEX-Graph einer FORALL-Schleife mit dynamischem Laufbereich.

In Figur 5.5 ist der INDEX-Graph der zentralen FORALL-Schleife in einer modifizierten Version von Loop1 (vgl. Anhang B) mit variabler oberer Grenze des Index-Laufbereichs gezeigt. Zuerst wird die Anzahl Iterationen als Differenz zwischen oberer und unterer Grenze des Laufbereichs berechnet (A in Fig. 5.5). Dann berechnet man mit Hilfe von n_{it} die Anzahl Codeblöcke (B in Fig. 5.5) und addiert 1 (C in Fig. 5.5), so dass das Resultat immer grösser als 0 wird. In diesem Beispiel wurde mit einem maximalen Verhältnis von Kommunikation zu Rechenarbeit von 20% gearbeitet, was für diese Schleife zu einem n_{it} von 6 führte. Am Schluss (D in Fig. 5.5) wird der errechnete Wert mit der maschinenabhängigen oberen Grenze für die Anzahl Codeblöcke verglichen und das Minimum davon als Resultat genommen. Im Gegensatz zu den oben gezeigten Formeln werden zur Laufzeit alle Ausdrücke mit Ganzzahlarithmetik berechnet, um die Effizienz zu steigern.

5.5. Experimente

Im vorliegenden Abschnitt sollen die folgenden Codegenerierungs-Parameter (in dieser Reihenfolge) experimentell ermittelt werden:

1. Maximales Verhältnis von Kommunikation zu Rechenarbeit bei Funktionsaufrufen (g in der Formel 5.11)

2. Maximales Verhältnis von Kommunikation zu Rechenarbeit bei PARCALLS (g in der Formel 5.12)

3. Anzahl Codeblöcke pro Prozessor, die bei einem PARCALL ingesamt maximal gestartet werden sollen (k in der Formel 5.14)

4. Anzahl Codeblöcke bei einem PARCALL, ab welcher der PARCALL zweistufig ausgeführt werden soll (vgl. Abschnitt 5.4.1 "Uebersetzung von FORALL-Schleifen)

Die MFL-Quellenprogramme der in den Experimenten verwendeten Programme findet man in Anhang B.

5.5.1. Experimente mit Funktionsexpansion

Im folgenden Experiment wird am Beispiel der binären, adaptiven Integration (BinInt in Anhang B) das Laufzeitverhalten bei unterschiedlicher Wahl der Granularität beziehungsweise des maximal zugelassenen Verhältnisses zwischen Kommunikation und Arbeit (vgl. Abschnitt 5.2 "Funktionsexpansion") gezeigt. Das Programm "BinInt" wurde mit den Granularitäten g = 10%, 30%, 100% übersetzt.

Diese Granularitäten sind so gewählt, dass bei 10% alle Funktionsaufrufe von "F" und "Trap" und bei 30% nur die Aufrufe von "Trap" expandiert werden. Bei 100% wird für jede Funktion ein eigener Codeblock generiert. Je gröber die Granularität gewählt wird, desto weniger Codeblöcke werden erzeugt und umgekehrt. Die entsprechenden Zahlen für die verschiedenen Versionen sind in Tabelle 5.3 dargestellt:

Verhältnis Kommunikation/Arbeit	10%	30%	100%
Anzahl CALLs	444	2218	3105

Tabelle 5.3 Anzahl CALLs für verschieden Versionen von BinInt

Das *Parallelitätsprofil* in Figur 5.6 zeigt, wieviele Prozessoren der ADAM-Maschine zu jedem Zeitpunkt ausgelastet sind. Betrachtet man die Kurve für g = 100%, so fällt als Erstes auf, dass sich die Parallelität in diesem Fall viel weniger schnell entfaltet als bei relativ geringerem Kommunikationsaufwand. Der Grund dafür ist, dass durch die vielen, sehr kleinen Codeblöcke die Token-Manager der Prozessoren und das Token-Netzwerk derart stark belastet sind, dass sie die Entfaltung der parallelen Arbeit behindern. Aus dem gleichen Grund führt auch nicht die Version mit 100%, sondern jene mit 30% zur maximalen Parallelität.

Vergleicht man die Flächen unter den Kurven als Mass für die tatsächlich ausgeführte Arbeit, so ist offensichtlich, dass viele Codeblockaufrufe viel Arbeit bedeuten. Mit zunehmendem Kommunikationsaufwand steigt diese Fläche an. Diesbezüglich schneidet die 10%-Kurve am besten ab. Trotzdem terminiert das 30%-Programm knapp am schnellsten. Offenbar wurde durch 10%-Partitionierung das Programm zu stark sequentialisiert, sodass im Vergleich zu 30% auch nur die halbe Parallelität erreicht wird.

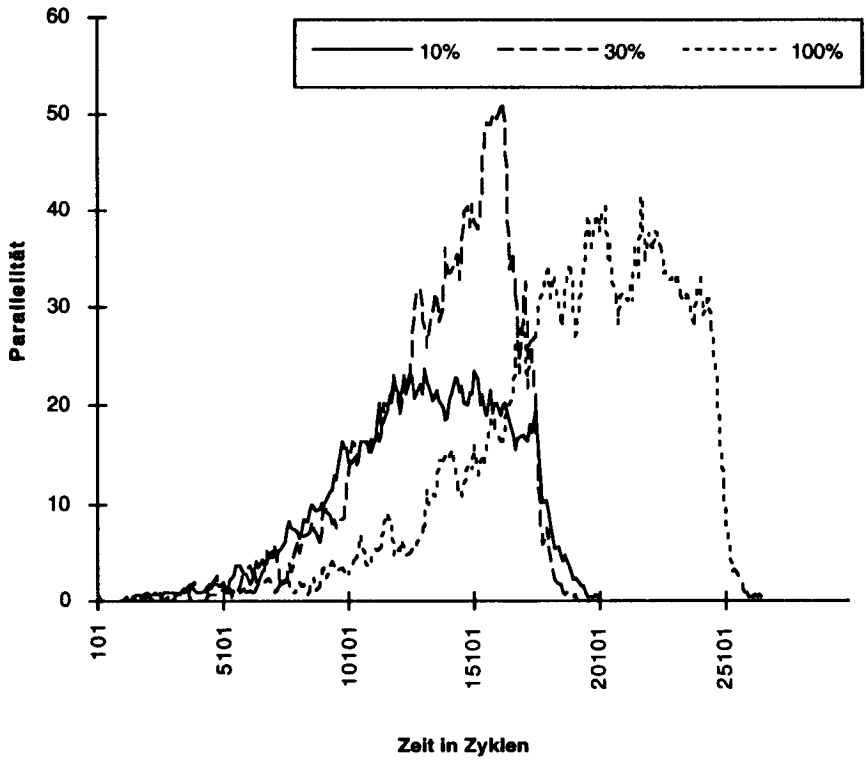


Fig. 5.6 Parallelitäts-Profile für "BinInt" mit unterschiedlicher Granularität

Einen interessanten Nebeneffekt der Expansion von Funktionen zeigt die Graphik in Figur 5.7. Der MFL-Compiler führt auf den Datenflussgraphen eine Reihe konventioneller Compiler-Optimierungen (vgl. Abschnitt 1.5 "Ueberblick über den Inhalt") aus. Durch die Expansion von Funktionen entstehen neue, grössere Datenflussgraphen, die neue Möglichkeiten der Optimierung bieten.

Im konkreten Falle von "BinInt" wird die Funktion "F" innerhalb der beiden Aufrufe von "Trap" in "Area" zweimal für den Wert "Mid" berechnet. Einmal erscheint "Mid" in "Trap" über den formalen Parameter "L" und einmal über "R". In den ursprünglichen Datenflussgraphen liess sich dieser gemeinsame Unterausdruck nicht erkennen, da für jede Funktion ein separater Graph generiert wurde. Im expandierten Graphen wird jedoch der gemeinsame Unterausdruck sichtbar. Analog dazu lassen sich bei Funktionsaufrufen mit konstanten Parametern nach der Expansion plötzlich neue konstante Unterausdrücke vorausberechnen. Darum werden die

konventionellen Optimierungstransformationen im MFL-Compiler einmal vor und einmal nach der Partitionierung auf die Graphen angewendet[5].

In der Figur 5.7 werden die Parallelitätsprofile für "BinInt" mit einem 30% Verhältnis zwischen Kommunikation und Arbeit verglichen. Die Elimination des gemeinsamen Unterausdruckes wirkt sich dadurch aus, dass die optimierte Version trotz geringfügig reduzierter Parallelität früher terminiert als die nicht optimierte.

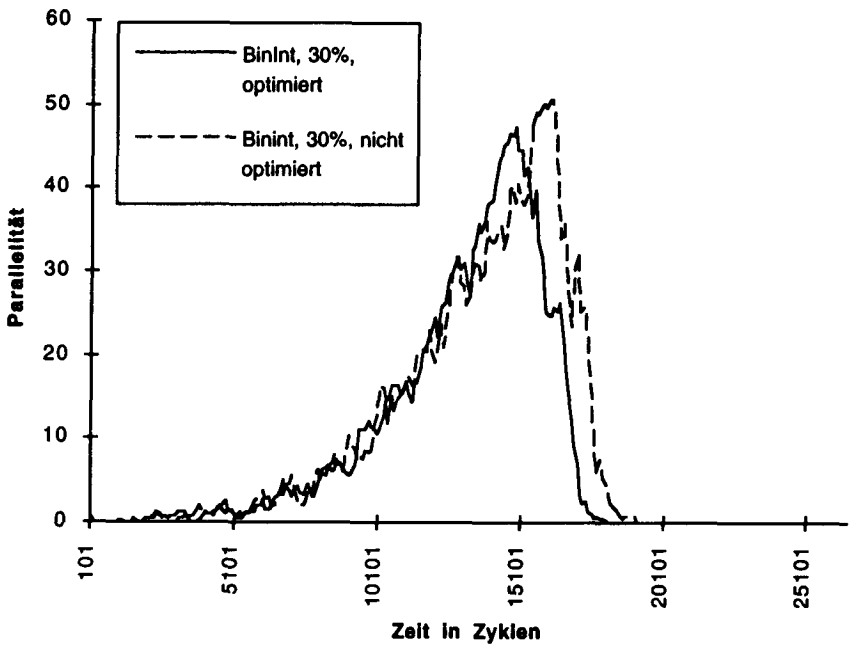


Fig. 5.7: Effekt der Graph-Optimierungen nach der Expansion

Die obigen Experimente zeigen einerseits, dass die Granularität der Funktionen ein wichtiger Parameter der Codegenerierung ist, obwohl der Unterschied zwischen 30% und 10% bei "BinInt" nicht wesentlich ist und sich auf Maschinen mit weniger Prozessoren, auf denen "BinInt" mit 30 % seine Parallelität nicht vollständig entfal-

---

5) Man kann sich fragen, ob nicht auf die Optimierungen vor der Partitionierung verzichtet werden könnte. Dies würde aber die Präzision der Kostenzuweisung (vgl. Abschnitt 5.2 "Kostenzuweisung") insbesondere dann beeinträchtigen, falls Schleifeninvarianten aus inneren Schleifen entfernt werden können.

ten kann, sogar ins Gegenteil umkehren wird. Das Entscheidende ist aber, dass zu kleine Codeblöcke vermieden werden, da man dafür einen doppelten Preis bezahlt: Einerseits entstehen dabei für den Start der Codeblöcke, die Parameterübergabe und für die Synchronisation zu grosse Zusatzkosten und andererseits werden jene Teile der Maschine, welche für die Lastverteilung zuständig sind, durch kleine Codeblöcke verstopft.

Es stellt sich dabei die Frage, ob so kleine Funktionen wie "Trap" und "F" in "BinInt" überhaupt sinnvoll sind und in praktischen Anwendungen vorkommen. Darauf gibt es zwei Antworten: Die Praxis der funktionalen Programmierung zeigt, dass man aus Gründen der Uebersichtlichkeit häufig kleine Funktionen verwendet, da sonst die Ausdrücke zu komplex werden. Auf der anderen Seite ist die ADAM-Architektur speziell dafür konstruiert, den Start und die Synchronisation von Codeblöcken so gut wie möglich in Hardware zu unterstützen. Auf heute erhältlichen Architekturen sind die Kosten für diese Operationen bedeutend höher. Dort entspricht eine obere Grenze für das Verhältnis von Kommunikation zu Rechenarbeit im Bereich von 10-30%, wie in der ADAM-Architektur sinnvoll, schon recht grossen Funktionen.

5.5.2. Experimente mit FORALL-Schleifen

In den Experimenten mit FORALL-Schleifen wurden vier verschiedene Beispielprogramme mit einfachen Schleifen verwendet. Die Livermore-Loops 1, 7 und 7B [McMaho86] bestehen aus jeweils zwei FORALL-Schleifen. In der ersten werden die verwendeten Arrays initialisiert, in der zweiten die eigentliche Berechnung ausgeführt. Der Schleifenkörper ist in "Loop1" einfacher als in "Loop7". Die Standard-Livermore-Loops laufen über Indexbereiche von ungefähr 1000. Für grössere Experimente mit vielen Prozessoren ist dies viel zu wenig. Deshalb wurde auch noch eine abgeänderte Version von Livermore-Loop 7 mit 19995 parallelen Iterationen verwendet. Das Programm "ParMerge" mischt zwei Arrays der Grösse 256 voller Zufallszahlen. Dabei wird zehnmal hintereinander eine sehr kleine FORALL-Schleife durchlaufen. Alle MFL-Quellenprogramme finden sich in Anhang B.

Im ersten Experiment, dessen Resultate in der Figur 5.8 zusammengefasst sind, wurden verschiedene Versionen der erwähnten Programme mit *unterschiedlicher Granularität* der Codeblöcke auf dem ADAM-Simulator mit 64 Prozessoren in Hypercube-Topologie ausgeführt. Es wurde nur die Zeit gemessen, welche die Programme in den parallelen Schleifen verbrachten. Alle Zeiten wurden auf das jeweils beste Resultat eines Programms normiert.

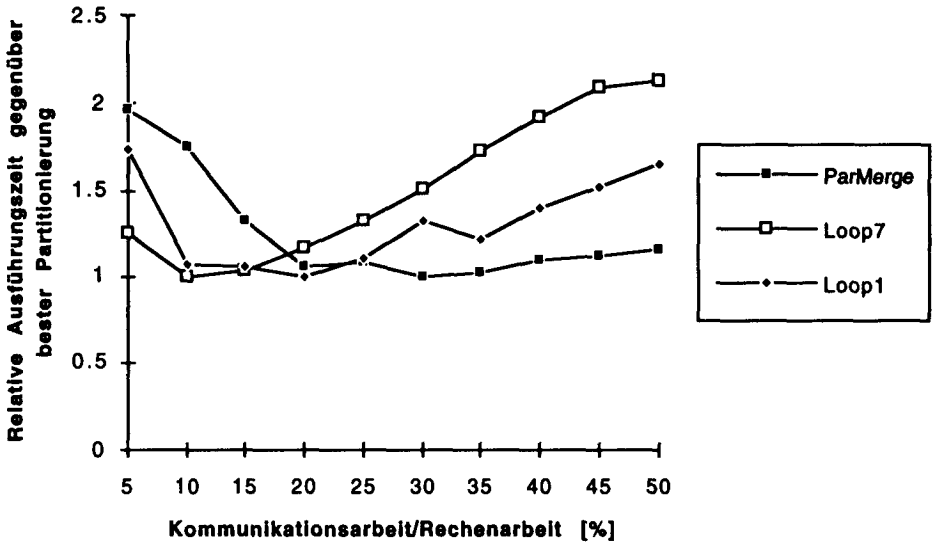


Fig. 5.8: Maximaler Zusatzaufwand bei FORALL-Schleifen (kleine Beispiele, 64 Prozessoren)

Die Resultate zeigen, dass die *optimale Wahl der Konstante g* (vgl. Abschnitt 5.4 "Uebersetzung von FORALL-Schleifen") mit den unterschiedlichen Programmen zwischen 10 und 30 % variiert, wobei Schleifen mit kleinerem Schleifenkörper offenbar eine feinere Granularität verlangen und umgekehrt. Dies liegt daran, dass in sehr kleinen Schleifen der Zusatzaufwand für die sequentielle Ablaufkontrolle, der bei der Kostenzuweisung nicht berücksichtigt wird, eine grössere Rolle spielt. Insgesamt lässt sich aber die in Zusammenhang mit der Figur 5.2 vorhergesagte Form der Kurven bei allen Beispielen erkennen.

Im nächsten Experiment wurde das zweite Kriterium zur Partitionierung von FORALL-Schleifen, die *Anzahl Codeblöcke pro Prozessor*, variiert. Die Resultate sind in Figur 5.9 in der gleichen Form dargestellt wie die Resultate des obigen Experiments (vgl. Fig. 5.8). Es wurde ebenfalls nur der parallele Anteil der Programme gemessen. Das Experiment wurde mit 16 Prozessoren[6] und Hypercube-Topologie ausgeführt.

---

[6]) Es stellt sich die Frage, weshalb mit den gleichen Programmen ein Experiment mit 64 (Fig. 5.8) und eines mit 16 Prozessoren (Fig. 5.9) ausgeführt wurde. Der Grund dafür ist, dass beim ersten Experiment das Minimum in

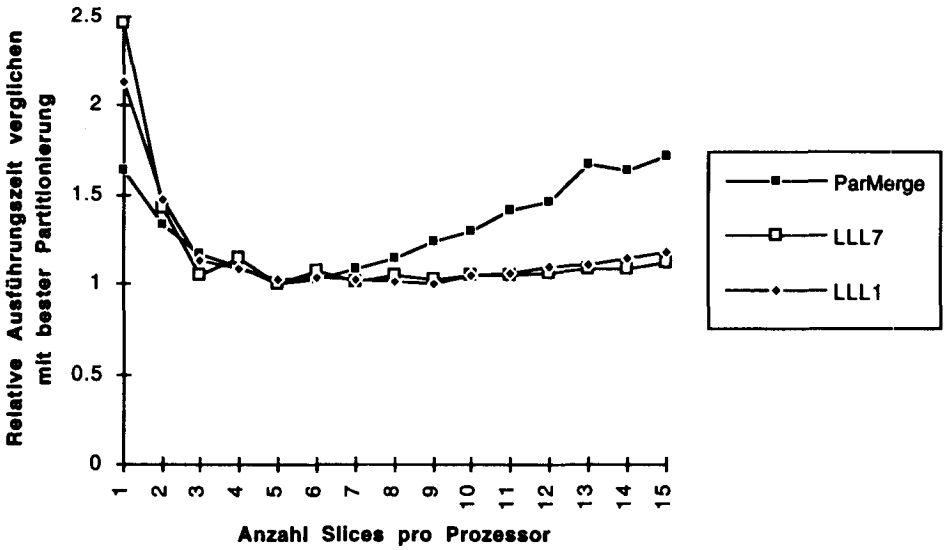


Fig. 5.9: Anzahl Codeblöcke pro Prozessor (kleine Beispiele, 16 Prozessoren)

In diesem Experiment fällt auf, dass die Resultate weniger stark von den Programmen abhängen. Dies entspricht eigentlich unseren Erwartungen, ist doch die Anzahl Codeblöcke pro Prozessor ein maschinen- und nicht ein programmabhängiges Kriterium. Die Graphik in Figur 5.9 zeigt auch, dass dieser Parameter ruhig etwas zu gross gewählt werden darf, da sich zu wenig Parallelität zum Verstecken der Speicherlatenz viel schlimmer auswirkt als zu grosser Zusatzaufwand. Dazu muss allerdings angemerkt werden, dass diese Beobachtung nur für gute Parallelrechner-Architekturen mit kleinem Zusatzaufwand zum Start und zur Synchronisation paralleler Codeblöcke gilt. Der markante Anstieg mit zunehmender Anzahl Codeblöcke beim Mischprogramm ("ParMerge") erklärt sich damit, dass diese Schleifen für 64 Prozessoren sehr klein sind und somit die Granularität schnell zu fein wird. Ein Vorteil feinerer Granularität ist zudem, dass sich die Quantisierungsfehler bei der Aufteilung der Schleife in parallele Abschnitte und Unregelmässigkeiten bei der Lastverteilung viel schwächer auswirken.

Für das dritte Experiment wurde der Livermore-Loop-7 so modifiziert, dass 20000 Iterationen ausgeführt werden, um auch das Verhalten grosser ADAM-Maschinen

---

der Formel 5.15 von der Granualrität (Formel 5.13) und beim zweiten Versuch das Minimum von der Anzahl Prozessoren (Formel 5.14) dominiert sein sollte.

(bis 256 Prozessoren) zu untersuchen. Für die Versuche wurde das Programm auf unterschiedlich vielen Prozessoren (p) und mit variierender Anzahl Codeblöcke pro Prozessor ausgeführt. Die Versuche bei 64 und 256 Prozessoren wurden ausserdem mit *ein-* und *zweistufigem* Aufruf der parallelen Codeblöcke ausgeführt (vgl. Abschnitt 5.4.1. "Uebersetzung von Forall-Schleifen").

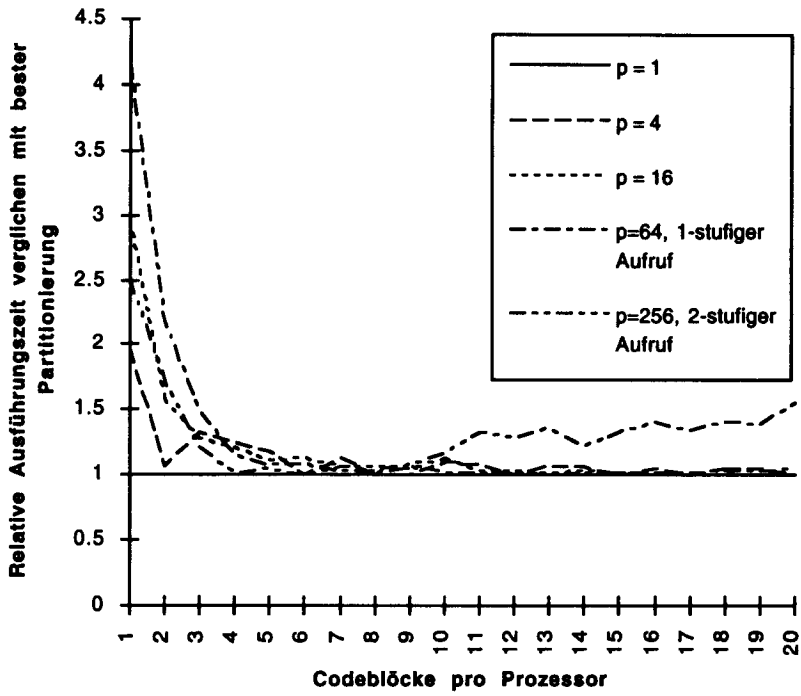


Fig. 5.10: Anzahl Codeblöcke pro Prozessor (grosses Beispiele 1-256 Prozessoren)

In der Figur 5.10 sind die Resultate für die verschiedenen Prozessoranzahlen dargestellt, wobei für 256 Prozessoren die Variante mit zweistufigem Aufruf aufgezeichnet wurde, weil für die bei 256 Prozessoren gebrauchte Anzahl von Codeblöcken diese Variante besser abschnitt. Der Graph für einen Prozessor ist praktisch eine Gerade, was zeigt, dass der Zusatzaufwand, der auftritt, wenn man eine so grosse FORALL-Schleife in 16 statt einen Codeblock aufteilt, keine Rolle spielt. Gegenüber den Resultaten in Figur 5.9 stellen wir fest, dass sich der optimale Bereich für die Anzahl Codeblöcke pro Prozessor leicht nach oben verschiebt bis ein Wert zwischen 6 und etwa 10 erreicht ist. Es fällt zudem auf, dass die Linien mit zuneh-

mender Anzahl Prozessoren, also auch zunehmend feinerer Granularität, immer stetiger verlaufen. Dies überrascht nicht, da bei insgesamt vielen Codeblöcken die *Quantisierungseffekte* viel weniger stark ins Gewicht fallen. Andererseits steigt die Kurve für 256 Prozessoren hinter dem Optimum am stärksten an, da man bei einer Aufteilung in derart viele Codeblöcke auch bei einer so grossen Anzahl Iterationen in jenen Bereich vorstösst, wo der Zusatzaufwand für die Parallelität eine Rolle zu spielen beginnt.

Zusammenfassend darf man sagen, dass die konstante, *obere Grenze für die Anzahl Codeblöcke pro Prozessor* für die ADAM-Architektur bei ungefähr 10 gewählt werden kann. Diese Anzahl genügt, um die gröbsten Quantisierungseffekte auszuglätten und die Speicherlatenz zu verstecken. In den Fällen, bei denen die Laufzeiten nach dem Optimum schnell wieder ansteigen ("ParMerge" in Fig. 5.8 und "p=256" in Figur 5.9), sorgt die obere Grenze für den Kommunikationsaufwand dafür, dass nicht zuviele Codeblöcke generiert werden.

Das vierte Experiment, ebenfalls mit dem ausgedehnten Livermore-Loop 7, gilt der Frage, wie sich der *Unterschied zwischen ein- und zweistufigem PARCALL* auswirkt. Auf der x-Achse in Figur 5.11 wurde wiederum die Anzahl Codeblöcke pro Prozessor variiert. Auf der y-Achse wurde hingegen die echte Laufzeit für das Programm, gemessen in Zyklen, aufgetragen. Der ausgedehnte Livermore-Loop 7 wurde auf 256 Prozessoren bearbeitet, wobei je eine Variante mit ein- und zweistufigem PARCALL generiert wurde. Da bei 256 Prozessoren die gesamte Bandbreite des Netzwerks in der üblichen Installation des Simulators zur Grenze wurde, wurden die Programme auch mit vierfacher Netzwerkbandbreite des Datennetzes ausgeführt.

Die Resultate in Figur 5.11 zeigen deutlich, dass ab einer bestimmten Anzahl Codeblöcke die Arbeit nicht mehr schnell genug auf die verschiedenen Prozessoren verteilt und am Schluss synchronisiert werden kann. Das einstufige Verfahren, welches weniger Zusatzaufwand verursacht, ist für 2 Codeblöcke pro Prozessor zwar noch schneller als der zweistufige Aufruf, jedoch reicht diese Zahl von Codeblöcken nicht aus, um die Speicherlatenz genügend zu tolerieren. Beim zweistufigen Verfahren zeigt sich eher der gewohnte Verlauf der Kurve: Steiler Abfall und Optimum bei 6 bis 10 Codeblöcken pro Prozessor und nachher ein leichtes Ansteigen. Bei grossen FORALL-Schleifen führen Lastverteilung und Synchronisation zu einem *Flaschenhals*, der durch ein zweistufiges Verfahren beseitigt werden kann.

Die gezeigten Resultate lassen auch einen Schluss darauf zu, in welcher Grössenordnung der letzte Parameter der Partitionierung für FORALL-Schleifen - die *minimale Anzahl paralleler Codeblöcke,* ab welcher der PARCALL zweistufig ausgeführt werden soll - sinnvollerweise gewählt wird. Bis drei Codeblöcke pro Prozessor ist der einstufige Aufruf effizienter als der zweistufige, später kehrt sich das Bild. Umgerechnet auf die gesamte Anzahl Codeblöcke ergibt sich eine Grenze von ungefähr 750. Diese Grenze ist abhängig von der Architektur. Gibt man dem Token-Netzwerk eine grössere Bandbreite, so erhöht sich diese Zahl, und umgekehrt. Denkbar wäre auch eine Architektur, die von sich aus die Token nach einem PARCALL baumartig verteilt. Dies würde die Aufgabe des Compilers zwar vereinfachen, da nur noch einstufige Aufrufe benötigt werden, jedoch die Hardware unnötig verkomplizieren, da die Token (vgl. Abschnitt 2.3. "Lebenszyklus eines Codeblocks") mit zusätzlicher Information versehen werden müssten.

Es zeigt sich bei diesem Versuch mit vielen Array-Zugriffen auch, dass auf 256 Prozessoren die üblicherweise mit 16 Zyklen pro Paket und Verbindung gewählte Netzwerkbandbreite ungegnügend ist. Die Speicherlatenz kann nicht mehr vollständig versteckt und somit die Maschine nicht mehr vollständig ausgenutzt werden. Vervierfacht man die Netzwerkbandbreite, so ändert dies im Falle der 1-stufigen PARCALLS nicht viel, da dort die Ausbreitung der Parallelität der Flaschenhals ist. Für zweistufige PARCALLs senkt sich die Rechenzeit jedoch beträchtlich.

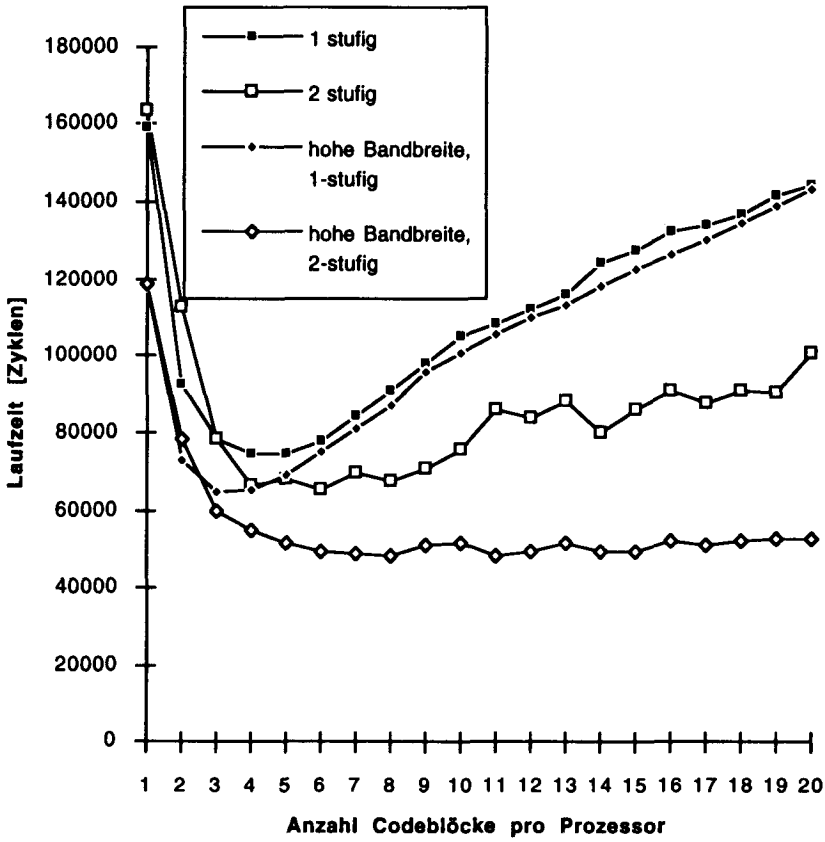


Fig. 5.11: Auswirkungen des zweistufigen PARCALLs (256 Prozessoren)

Betrachten wir noch die *Speedup-Kurven* für den ausgedehnten Livermore-Loop7 in Figur 5.12, so erhalten wir für zunehmende Prozessorzahlen bis 64 Prozessoren einen annähernd linear ansteigenden Speedup, wobei die Effizienz von fast 100 % bei vier Prozessoren auf 70% bei 64 Prozessoren absinkt. Erst für 256 Prozessoren trennen sich die drei Kurven deutlich. Die Netzwerkbandbreite und das ungeeignete Aufrufverfahren bei den einstufigen PARCALLs begrenzt den Speedup nach oben, so dass die Effizienz je nach dem auf 53 bis 33 Prozent absinkt.

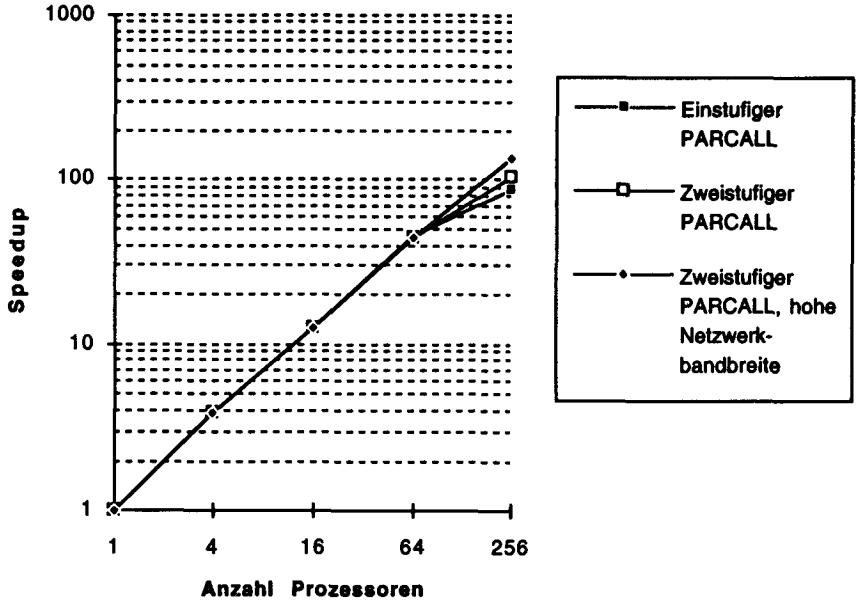


Fig. 5.12: Speedup für den ausgedehnten Loop7

Im letzten Experiment, dessen Resultate in der Figur 5.13 dargestellt sind, werden die *Parallelitätsprofile* der gleichen drei Programme wie im vorangehenden Experiment mit der zusätzlichen Variante "1-stufiger PARCALL, schnelles Netzwerk" verglichen.

Anhand der Parallelitätsprofile in Figur 5.13 lässt sich der Einbruch beim Speedup (vgl. Fig. 5.12) oberhalb von 64 Prozessoren erklären. Ungefähr 6000 Zyklen am Anfang, zwischen den beiden FORALL-Schleifen, sowie am Ende des Programms müssen ohnehin sequentiell ausgeführt werden, was den möglichen Speedup (vgl. Fig. 5.12) nach dem Gesetz von Amdahl [Amdahl67] limitiert. Berücksichtigt man zusätzlich, dass pro Prozessor mehrere, parallele Codeblöcke gebraucht werden, um die Latenz zu verstecken, und dass das ganze Programm insgesamt sehr kurz läuft (70000 Zyklen = 3.5µs bei 20 MHz Takt), so ist der erreichte Speedup von 100 recht beachtlich. Eine 256-Prozessor-Maschine setzt man eben sinnvollerweise erst bei grösseren Problemen ein.

In allen Parallelitätsprofilen lassen sich schön die zwei den beiden FORALL-Schleifen entsprechenden Maxima erkennen. Das Parallelitätsprofil des Initialisierungs-FORALL unterscheidet sich deutlich für den ein- und den zweistufigen

PARCALL. Beim einstufigen PARCALL überschreitet die Parallelität nie ein gewisses Mass um ca. 50. Das liegt daran, dass die Codeblöcke nicht schnell genug verteilt werden können und somit die Prozessoren schon wieder frei für neue Arbeit sind, bevor die Last vollständig verteilt ist. Der zweistufige PARCALL löst dieses Problem, obwohl auch hier die Begrenzung durch die Netzwerkbandbreite deutlich zu Tage tritt.

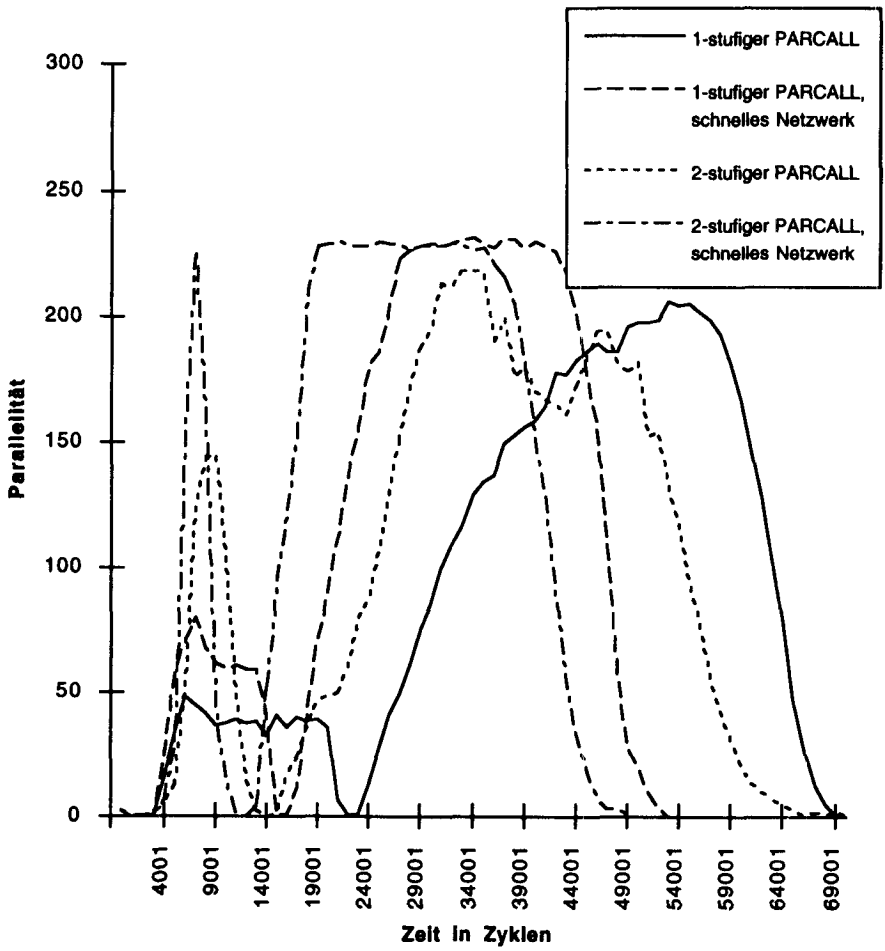


Fig. 5.13: Parallelitätsprofil für unterschiedliche Varianten von Loop7

Beim zweiten FORALL ist die Netzwerkbandbreite der dominierende Faktor. Für das langsame Netzwerk kann die volle Parallelität gar nicht erreicht werden. Der Einfluss des zweistufigen PARCALLs zeigt sich aber trotzdem deutlich dadurch,

dass die Parallelität bei gleicher Netzwerk-Charakteristik steiler ansteigt als beim einstufigen PARCALL. Zudem sieht man, dass ein langsameres Datennetzwerk auf der ADAM-Architektur auch eine langsamere Ausbreitung der Parallelität bedeutet, was sich dadurch begründen lässt, dass jeder Prozessor mehr parallele Codeblöcke benötigt, um die Latenz zu verdecken.

Die Experimente zeigen, dass die Wahl der Granularität für die Codeblöcke ein entscheidender Faktor bei der Codegenerierung für die ADAM-Architektur ist. Zudem geben sie Auskunft darüber, in welcher Grössenordnung die weiter vorne beschriebenen Parameter für die Codegenerierung zu wählen sind. Für die ADAM-Architektur selbst bewirken kleine Veränderungen dieser Parameter nur geringe Veränderungen im Laufzeitverhalten, da diese Architektur kompromisslos für diese Konzepte konstruiert wurde. Es ist aber denkbar, dass die ADAM-Architektur dereinst nicht in Hardware, sondern als Betriebssystemschicht (Stichwort: "*Lightweight Processes*") implementiert wird; Damit liegt der Zusatzaufwand für die Parallelität höher, was sich in einer empfindlicheren Reaktion auf die Wahl der Partitionierungsparameter äussern wird.

5.6. Arbeiten mit ähnlichen Ansätzen

Die umfassendste Arbeit zum Thema "Partitionierung" ist sicher [Sarkar89]. Er hat einen Partitionierer gebaut, der ausgehend von einem Kostenmodell IF1-Graphen [SkeGla84] partitioniert. Eng verwandt mit jener Arbeit ist auch die in diesem Kapitel gezeigte Formulierung des Partitionierungsproblems. Sarkar löst das Problem der mangelnden Ausgangsdaten für die Partitionierung, indem er vor der Partitionierung das Programm ablaufen lässt und dabei ein Ausführungsprofil erstellt. Das grundsätzliche Problem dieses Vorgehens liegt darin, dass die Partitionierung strenggenommen nur für die bei der Erstellung des Ausführungsproblems verwendeten Eingangsdaten korrekt ist. Die Partitionierung ist rein statisch und kann sich nicht an die Eingangsdaten anpassen. In der gleichen Arbeit wird auch gezeigt, dass das Partitionierungsproblem NP-vollständig ist. Dabei wird auch ein heuristischer Algorithmus vorgeschlagen, um "gute" Partitionierungen zu erreichen. Sarkar selbst nennt am Ende seiner Arbeit die Konvexität des Partitionengraphen eine unrealistische, aus theoretischen Gründen gewählte Einschränkung, bezeichnet aber deren Aufhebung als "Gegenstand zukünftiger Forschung", ohne konkreter zu werden. Eine frühere Version der Arbeit wurde schon in [SarHen86] präsentiert. Interessanterweise haben diese Autoren eine ähnliche Lösungsvariante verfolgt, wie sie in dieser Arbeit vorgeschlagen wird [SaSkMi88].

In [HudGol85] werden ALFL-Programme in sogenannte "Serial Combinators" partitioniert. Allerdings wurde dort die Einschränkung gemacht, dass auf keinen Fall Parallelität geopfert werden darf. Zusammen mit weiteren Einschränkungen der Serialisierbarkeit, die daher stammen, dass ALFL keine Einfachzuweisungs- sondern eine volle funktionale Sprache ist, führt dies zu einer weit kleineren Granularität als im MFL-Compiler.

Mit dem Paradigmenwechsel zum grobgranularen Datenfluss stellt sich das Partitionierungsproblem in neuester Zeit auch in der Datenflussforschungsgruppe des MIT. In [Traub91] wird ein Verfahren gezeigt, um Datenflussgraphen in sogenannte "threads" zu zerlegen. Im Gegensatz zur ADAM-Architektur gilt die Einschränkung, dass nur Instruktionen in einem "thread" sequentialisiert werden dürfen, falls diese nicht zu einem Unterbruch führen können. Bei jedem asynchronen Speicherzugriff muss dementsprechend ein neuer "Thread" gestartet werden. Partitionierung heisst in diesem Ansatz weniger "Ausgleich von Kommunikation und Rechenarbeit" sondern "Bilde die maximalen erlaubten Threads". Aehnlich wie in [HudGol85] werden diese kleiner sein als die ADAM-Codeblöcke.

In [InStXi86] wird das Partitionierungsproblem sehr schön in seiner allgemeinen Form beschrieben und es werden Lösungsansätze unter verschiedenen Rahmenbedingungen gezeigt. Allerdings wird das Problem für ungerichtete Graphen diskutiert, was eher dem Modell kommunizierender, sequentieller Prozesse als dem Datenflussmodell entspricht.

Weitere interessante Arbeiten zum Thema sind [GirPol88], [FisERN84] und [GauErc84].

6. Effiziente Verwaltung von Datenstrukturen

In diesem Kapitel wird zuerst gezeigt, dass die übliche Art [AhSeUl86, Kap. 7], strukturierte Variablen zu verwalten, sich nicht für funktionale Sprachen eignet, da sonst zuviel kopiert werden muss. Es wird ein alternatives Laufzeitmodell vorgeschlagen, welches nur Zeiger auf strukturierte Objekte anstelle der Objekte selbst verwaltet. Für dieses Modell müssen zwar die Referenzen auf ein Objekt dynamisch gezählt werden, doch können die expliziten Referenzzähleroperationen in vielen Fällen wegoptimiert werden. Der zweite Teil befasst sich damit, wie die richtige Objektklasse anhand des Zugriffsprofils der entsprechenden Variable gewählt wird. Im dritten Teil wird gezeigt, wie durch eine geschickte Kombination der Hardwaremöglichkeiten der ADAM-Architektur und entsprechender Software-Ergänzungen Referenzen gezählt werden, wenn strukturierte Werte nicht nur durch einfache Objekte, sondern durch ganze Objektbäume dargestellt werden.

Am Schluss des Kapitels findet man eine Reihe von Experimenten auf dem ADAM-Simulator, welche die gemachten Aussagen untermauern, und einige Hinweise auf ähnliche Arbeiten.

Voraussetzung zum Verständnis dieses Kapitels ist die Lektüre der einführenden Kapitel 1 bis 4 zur ADAM-Architektur und MFL.

6.1. Strukturierte Werte in MFL

6.1.1. Fixe Zuordnung von Speicherplätzen zu Variablen

Im Prinzip entsprechen die *Register-Frames* (vgl. Abschnitt 2.4 "Das Register-Frame") den *Aktivationsrahmen* ("Activation Frames") auf dem Aufrufstack in einer klassischen Programmiersprache. Da die Register-Frames in der ADAM-Architektur nur eine begrenzte Grösse haben, können strukturierte Datenobjekte (Tupel und Arrays) nicht direkt in den Frames gespeichert werden, wie dies normalerweise in klassischen Laufzeitmodellen mit strukturierten Variablen getan wird. Die ADAM-Architektur sieht für strukturierte Variablen sogenannte *Objekte* (vgl. Abschnitt 2 "Die ADAM-Architektur") vor. Dieses Modell ist durch eine *fixe Zuordnung von Speicherplatz zu einer Variablen* charakterisierbar.

Das bekannte Prinzip der Aktivationsrahmen [AhSeUl86, Kap. 7] lässt sich durch eine kleine Erweiterung problemlos auch mit Objekten verwenden, indem man beim Eintritt in eine Funktion für jede strukturierte, lokale Variable ein entspre-

chendes Objekt alloziert und dieses vor dem Ende der Funktion wieder deallociert. Bei der Uebergabe strukturierter Parameter zu aufgerufenen Funktionen lässt sich ohne Gefahr nur ein Zeiger auf das Objekt übergeben, da durch das Laufzeitmodell von MFL sichergestellt ist, dass die lokalen Variablen aufrufender Funktionen immer länger leben als jene der aufgerufenen Funktionen. Man kann für dieses Modell also auf ein *Garbage-Collection-Konzept* verzichten.

Das Problem einer solchen Lösung ist, dass bei jeder Zuweisung einer strukturierten Variablen ein grosser Aufwand entsteht, da deren ganzer Inhalt kopiert werden muss. In klassischen Programmiersprachen ist dieses Problem nicht gravierend. Man vermeidet derartige Zuweisungen, indem man Algorithmen als Transformationen auf einer einzelnen strukturierten Variablen entwirft[1]. Die *Einfachzuweisungsregel* verbietet solche Techniken in MFL. Ausserdem haben in MFL alle Kontrollflusskonstrukte den Charakter von Ausdrücken, deren Ergebnisse Variablen zugewiesen werden müssen. Der Zuweisungsoperator wird in funktionalen Programmen deshalb viel häufiger verwendet als in imperativen Programmen. Demzufolge wird das *Vermeiden unnötiger Kopierarbeit* ("copy avoidance") zu einem entscheidenden Problem bei der Uebersetzung funktionaler Sprachen zu effizienten Maschinenprogrammen. Aus diesem Grund ist das oben vorgeschlagene Laufzeitmodell für strukturierte Variablen untauglich.

6.1.2. Strukturierte Kanten in Datenflussgraphen

Im Datenflussgraph spielen die Zuweisungen keine spezielle Rolle mehr. Sie werden wie die Datenabhängigkeiten in Ausdrücken durch einfache Kanten ersetzt. Da MFL eine Sprache mit strengem Typensystem ist, ist immer bekannt, von welchem Typ eine Kante ist. Diese Information können wir benutzen, um zwischen *skalaren* und *strukturierten Kanten* zu unterscheiden. Auf strukturierten *Kanten* (vgl. Kapitel 4 "Datenflussgraphen als Zwischencode für MFL" werden in diesem Modell Referenzen auf Objekte zwischen den *Knoten* des *Datenflussgraphen* kommuniziert, womit die Notwendigkeit des Umkopierens bei einer Zuweisung wegfällt.

Auch diese Lösung ist selbstverständlich nicht ohne Nachteil. Da es keine einfache Kopplung zwischen Variablennamen und Objekten mehr gibt, lässt sich die

---

[1]) Klassische Beispiele dafür sind die üblichen Gauss-Algorithmen zur Auflösung eines linearen Gleichungssystems, die die LR-Zerlegung der ursprünglichen Matrix an die Speicherstellen der ursprünglichen Matrix zurückschreiben.

Lebensdauer eines Objekts nicht einfach durch den statischen Definitionsbereich der entsprechenden Variable bestimmen. Da einerseits Ausgänge von Knoten mit mehreren Eingängen verbunden sein können und andererseits Eingänge zu zusammengesetzten Knoten von mehreren Untergraphen übernommen werden, ist es notwendig, die *Anzahl Referenzen* auf ein Objekt zu kontrollieren.

Die ADAM-Architektur unterstützt diese Art der Objektverwaltung, indem sie das erste Wort jedes Objekts für einen solchen *Referenzzähler* reserviert und die entsprechenden Operationen zum *atomaren Inkrementieren* bzw. *Dekrementieren* des Zählers anbietet. Es ist sehr wichtig, dass in einem parallelen System diese Operationen durch die Hardware unterstützt werden, da sonst bei parallelen Zugriffen die konsistente Manipulation des Referenzzählers nur mühsam und ineffizient zu programmieren wäre. Fällt der Referenzzähler auf Null zurück, so kann über den betreffenden Speicherplatz neu verfügt werden.

Die Atomizität von Referenzzähler-Operationen kann bei grossen, verteilten Objekten, auf die von vielen Prozessoren zugegriffen wird, zu einem ernsthaften Problem werden, weil die Referenzzähler-Zugriffe sequentialisiert werden. Wie im nächsten Teil dieses Kapitels gezeigt wird, lässt sich die Anzahl dieser Operationen glücklicherweise durch einfache Optimierungen substantiell senken.

6.2. Statische Optimierung der Referenzzähler-Operationen

6.2.1. Referenzzähler-Operationen und Invarianten

Die einfachste *Lösung zur Manipulation der Referenzzähler* besteht darin, die tatsächliche Anzahl der Referenzen auf ein Objekt zählen. Für jeden Speicherplatz in der Maschine müsste dabei klar sein, ob er eine Referenz enthält. Wird ein solcher Speicherplatz überschrieben oder freigegeben, so muss der Referenzzähler des entsprechenden Objekts dekrementiert werden. Wird eine Referenz neu an eine Speicherstelle geschrieben, so muss der Referenzzähler erhöht werden. Würde man das Zählen der Referenzen direkt in die Hardware der Maschine integrieren, so wäre wahrscheinlich eine solche Lösung zu wählen.

Bei der ADAM-Architektur ist das Zählen der Referenzen nicht automatisiert. Es gibt vier Operationen, die den Referenzzähler eines Objekts beeinflussen (vgl. auch Abschnitt 2.5 "Der Objekt-Manager: Ein verteilter, gemeinsamer Speicher"):

1. *NEW* generiert ein neues Objekt der gewünschten Grösse und initialisiert den *Referenzzähler auf 1.*

2. *DEALLOCATE dekrementiert den Referenzzähler* und gibt den durch das Objekt belegten Speicherplatz frei, falls dieser auf 0 sinkt.

3. *REF inkrementiert den Referenzzähler.*

4. *DEREF dekrementiert den Referenzzähler*, wobei der Speicherplatz am Schluss nicht implizit freigegeben wird, sondern noch zusätzlich durch ein DEALLOCATE befreit werden muss.

Es ist nicht nötig, tatsächlich alle Referenzen zu zählen. Ueber die Lebenszeit eines Objekts muss lediglich die folgende *Invariante* erfüllt sein, wobei $n_{DEALLOC}$ und n_{REF} die Anzahl der DEALLOCATE- bzw. der REF-Instruktionen auf das Objekt bedeuten:

Während der Lebenszeit des Objekts: $n_{REF} > n_{DEALLOC} - 1$

Ueber 1 darf die Differenz nie steigen, da sonst Referenzzähler-Operationen auf inexistenten Objekten ausgeführt würden. Zusätzlich muss die Forderung erfüllt sein, dass am Ende des Programms der gesamte Speicher wieder freigegeben ist:

Am Ende des Programms: $n_{REF} = n_{DEALLOC} - 1$

Ausserdem sollen die absoluten Werte für $n_{DEALLOC}$ und n_{REF} möglichst klein gehalten werden, da wie oben erklärt, Referenzzähler-Operationen in einer parallelen Umgebung sehr teuer sein können. Wir werden nun schrittweise eine Methode aufbauen, um den *Datenflussgraph* (vgl. Kapitel 4 "Datenflussgraphen als Zwischencode für MFL") mit Informationen zur Manipulation der Referenzzähler zu ergänzen.

6.2.2. Ergänzung der Datenflussgraphen mit Referenzzähler-Operationen

An den MFL-Datenflussgraphen müssen zwei Erweiterungen vorgenommen werden, damit die Referenzzähler Operationen ebenfalls in den Graphen festgehalten werden können:

1. Es muss ein *neuer Knoten "REF"* vorgesehen werden, der an seinem Eingang ein Objekt erwartet und an seinem Ausgang dasselbe Objekt mit erhöhtem Referenzzähler zurückgibt.

2. Der *Umgebungsknoten* (bzw. der Graph) bekommt neben den Ausgängen des Graphen eine neue Klasse von Eingängen, die *Deallokationseingänge*, auf welche alle Objekte geführt werden, die freigegeben werden sollen. Nachdem alle Instruktionen eines Graphen ausgeführt worden sind, werden am Schluss eines Graphen noch DEALLOCATE-Instruktionen für jeden Deallokationseingang ausgeführt. In den graphischen Darstellungen werden Knotenausgänge, die mit Deallokationseingängen verbunden sind, als ausgefüllte schwarze Punkte anstelle Kreisen dargestellt.

Die Vergabe der Referenzzähler-Operationen im Graph erfolgt nun nach dem *lokalen Verursacherprinzip*: Jeder Graph deralloziert jene Objekte, die er lokal kreiert hat.

Am *Anfang der Lebenszeit* eines Objekts steht immer ein *NEW-Knoten* im Graph (vgl. Abschnitt 4.5 "Knoten zur Handhabung von Datenstrukturen"). Er liefert an seinem Ausgang ein Objekt, dessen Referenzzähler auf 1 steht. Jeder NEW-Knoten muss also mit einem Deallokationsausgang verbunden werden. Für alle Objekte, die über die Grenzen eines Graphen hinaus am Leben bleiben sollen, muss der *Referenzzähler erhöht* werden. Dies ist nur bei *strukturierten Kanten*, die zu den *Ausgängen des Graphen* führen, der Fall.

Auf der Ebene eines Graphen können neue Objekte nicht nur an den Ausgängen von NEW-Knoten entstehen, sondern auch an den Ausgängen von *zusammengesetzten Knoten* und *CALL-Knoten*. Somit müssen diese Ausgänge ebenfalls mit den Deallokationseingängen verbunden werden. Im Prinzip erscheinen auch Objekte, die über die Eingänge in einen Graphen eingespiesen werden, für diesen Graphen als neue Objekte, und man könnte argumentieren, dass diese Grapheingänge ebenfalls mit den Deallokationseingängen verbunden werden müssen. Dies ist aber nicht der Fall, da diese Objekte nicht innerhalb des Graphen oder eines seiner Untergraphen kreiert worden sind. Die Verantwortung für deren Deallokation liegt deshalb auf der Ebene des übergeordneten Graphen.

Eine spezielle Situation entsteht bei iterativen, zusammengesetzten Knoten (vgl. Abschnitt 4.3.2 "Iterative Ausdrücke"). Da strukturierte Iterationsvariablen im INIT-Graphen des zusammengesetzten Knotens und nicht in einem übergeordneten Graphen kreiert werden, müssen in den BODY- und RETURNS-Graphen auch die strukturierten Iterationsvariablen-Eingänge mit Deallokationsausgängen verbunden werden.

Wenn man die diskutierten Erkenntnisse zusammenfasst, entsteht der folgende Algorithmus, der auf jeden Graphen G = (N, E, U) angewendet werden muss:

```
FORALL n ∈ N DO
    (* Verbinde strukturierte Ausgänge von NEW, CALL und zusammengesetzten Knoten
    mit Deallokationseingängen *)
    IF n.operation ∈ {NEW, IF, LOOPA, LOOPB, FOR, FORALL, CALL} THEN
        FORALL e ∈ {e ∈ E | e = (n, x, m, z) ∧ Structured(e) } DO
            E := E ∪ {(n, x, U, d)}, wobei d Deallokationseingang ist
        END
    END

    (* Verbinde bei strukturierten Iterationsvariablen die Eingänge des Graphen mit
    Deallokationseingängen *)
    IF LoopBody(G) ∨ LoopReturns(G) THEN
        FORALL {e ∈ E | e = (U, x, m, z) ∧ Structured(e) ∧ IterationVar(x)} DO
            E := E ∪ {(U, x, U, d)}, wobei d Deallokationseingang ist
        END
    END

    (* Ergänze strukturierte Ausgangskanten mit REF-Knoten *)
    FORALL {e ∈ E | e = (m, x, U, z) ∧ Structured(e) } DO
        N := N ∪ {r}
        r.operation := REF
        E := E ∪ {(r, 1, U, z), (m, x, r, 1)}
    END
END
```

6.2.3. Optimierung der Referenzzähler-Operationen

Obwohl der oben gezeigte, graph-basierte Algorithmus im Vergleich zu einem wirklichen Zählen aller Referenzen schon mit deutlich weniger Referenzzähler-Operationen auskommt, gibt es noch eine einfache Möglichkeit, den Aufwand weiter zu reduzieren. Ohne die Invariante für die Referenzzähler zu verletzen, kann man problemlos Paare (REF und DEALLOCATE) von Referenzzähler-Operationen weglassen, falls man sicher ist, dass immer die Reihenfolge zuerst REF dann DEALLOCATE eingehalten ist. Einhalten einer Reihenfolge bedeutet in Datenflussgraphen, dass im Graph ein Pfad zwischen den beiden Knoten besteht. Etwas formaler gesagt sieht die Bedingung, damit ein REF-Knoten und eine Kante zu einem Deallokationseingang weggelassen werden können, folgendermassen aus:

$$(m \leq^* n) \wedge (n.operation = REF) \wedge ((m, x, U, d) \in E) \qquad (6.1)$$

Leider ist diese Bedingung etwas zu schwach, da sie auf dem Pfad von m zu n jede Art von Knoten zulässt. Tatsächlich dürfen auf diesem Pfad nur Knoten vorkommen, die ein Objekt transparent weitergeben. Zu dieser Gruppe von Knoten gehören nur STORE, UNIFY und BLOCKMOVE (vgl. Abschnitt 4.5 "Knoten zur Handhabung von Datenstrukturen"), welche von einem bestimmten Eingang zum Ausgang die Objektreferenz unverändert weitergeben. Ohne auf die Details einzugehen, definieren wir ein Prädikat ObjPath(m, z, n, y), welches erfüllt ist, falls vom Ausgang z eines Knotens m ein *transparenter Pfad* zum Eingang y des Knotens n führt. Damit sähe der *Optimierungsalgorithmus* für einen Graphen G = (N, E, U) so aus:

```
FORALL n ∈ N DO
    IF ObjPath(m, x, n, 1) ∧ (n.operation = REF) ∧ ((m, x, U, d) ∈ E) THEN
        (* Entferne den REF-Knoten, die Kanten, die sich auf ihn beziehen,
        und die Kante zum Deallokationsausgang aus dem Graph *)
        E := E \ {(o, y, n, 1), (n, 1, p, z), (m, x, U, d)}
        N := N \ {n}
        (* Verbinde die zum REF-Knoten adjazenten Knoten direkt. *)
        E := E ∪ {(o, y, p, z)}
    END
END
```

6.2.4. Anwendbarkeit der Methode

Bei der Codegenerierung werden die einzelnen Graphen zu Instruktionssequenzen umgesetzt (vgl. Kapitel 7 "Sequentialisierung und Codegenerierung"). Dabei wird ganz am Schluss dieser Sequenz für jeden Deallokationsausgang des Graphen eine DEALLOCATE-Instruktion generiert.

Die Anwendbarkeit der gezeigten Methode ist an die wichtige Bedingung geknüpft, dass die Ausführung der Graphen einer *strikten (Call-by-value) Semantik* gehorcht [Stoy77]. Vereinfacht gesagt muss die Bedingung gelten, dass die Berechnung eines Graphen vollständig abgeschlossen sein muss, bevor seine Resultate gebraucht werden dürfen. Dies ist ein Gegensatz zur *klassischen, nicht strikten Datenflussemantik* für feingranulare Datenflussmaschinen und allgemeinere funktionale Sprachen (vgl. Abschnitt 3.6 "Grenzen von MFL und weiterführende Ideen"). Diese Semantik entspricht aber genau dem durch die ADAM-Architektur und die Sprache MFL gegebenen Berechnungsmodell.

Anhand des folgenden Beispiels soll dieses Konzept noch abschliessend illustriert werden. Angenommen MFL hätte keine strikte Semantik und die Funktion "VerarbeiteArray" würde das erste Resultat viel schneller liefern als das zweite, so

könnte der Array "c" nach der vorgestellten Methode dealloziert werden, obwohl er in "VerarbeiteArray" noch gebraucht wird, um das zweite Resultat zu berechnen. Der Grund dafür ist, dass die Funktion "XXX" sofort terminiert und somit nach den gezeigten Regeln "c" dealloziert, sobald das erste Resultat aus "VerarbeiteArray" eingetroffen ist.

```
FUNCTION XXX(RETURNS REAL);
VAR   a, b: REAL;
      c: AnArray;
BEGIN
      c := ProduziereArray();
      a, b := VerarbeiteArray(c);
RETURNS
      a
ENDFUNCTION;
```

6.3. Auswahl der Objektklasse

6.3.1. Zugriffscharakteristik der verschiedenen Objektklassen

Die ADAM-Architektur sieht drei Klassen von Objekten mit unterschiedlicher *Zugriffscharakteristik* vor (vgl. Abschnitt 2.5. "Der Objekt-Manager: Ein physisch verteilter, logisch gemeinsamer Speicher"). Damit für verschiedene Anwendungsfälle die geeigneten Objekte verwendet werden können, müssen wir die Zugriffscharakteristik etwas genauer untersuchen.

In der Tabelle 6.1 sind die Zugriffslatenz und die Zugriffsbandbreite für Objekte der verschiedenen Objektklassen angenommen. Die Zeiten T_{mem}, T_{hop} und T_{bus} bezeichnen die Zeiten für einen Speicherzugriff, das Weiterreichen einer Anforderung und des Resultats über das Netz sowie die Zeit für das Schreiben eines Worts auf alle Prozessoren über den Broadcast-Bus. Die beiden Werte d_l und d_d bedeuten, über wieviele Prozessoren eine Anforderung durchschnittlich weitergereicht werden. Dabei wird zwischen den Werten für Zugriffe auf lokale Objekte und Zugriffe auf verteilte Objekte unterschieden, weil ein Zugriff auf ein lokales Objekt auf Grund der Lastverteilungsstrategie der ADAM-Architektur in der Regel unabhängig von der Anzahl Prozessoren auf einen relativ eng benachbarten Prozessor geschieht. Dagegen ist für d_d der halbe Netzwerk-Durchmesser eine gute Annahme, da verteilte Objekte unabhängig von der Laustverteilung statisch über die ganze Maschine verteilt werden. Mit c ist die Konnektivität der Prozessoren (Anzahl Verbindungen zu anderen Prozessoren) und mit p die Anzahl Prozessoren gemeint. Bei der Zugriffsbandbreite unterscheiden wir zusätzlich, ob insgesamt die Speicherbandbreite oder die Netzwerkbandbreite der begrenzende

Faktor ist. Für diese Rechnungen wurde zur Vereinfachung angenommen, dass es im Netz keine lokalen Ueberlastungen ("hot spots") gibt.

	Zugriffslatenz		Zugriffsbandbreite	
Objektklasse	lokal	über Netzwerk	T_{mem} dominiert	T_{hop} dominiert
lokal	T_{mem}	$T_{mem}+d_l\,T_{hop}$	$\frac{1}{T_{mem}}$	$\frac{c}{T_{hop}}$
verteilt	T_{mem}	$T_{mem}+d_d\,T_{hop}$	$\frac{p}{T_{mem}}$	$\frac{c\,p}{T_{hop}}$
repliziert (Lesen)	T_{mem}		$\frac{p}{T_{mem}}$	
repliziert (Schreiben)	T_{bus}		$\frac{1}{T_{bus}}$	

Tabelle 6.1: Zugriffslatenz und -bandbreite nach Objektklasse [MurFär92]

Diese Zugriffscharakteristiken unterscheiden sich fundamental zwischen den verschiedenen Objektklassen. Die Bedingung dafür, dass sich ein Objekt als Datenstruktur für massiv parallele Verarbeitung eignet, ist eine *skalierbare* Zugriffsbandbreite, was gleichbedeutend damit ist, dass die Anzahl der Prozessoren (p) im Zähler vorkommt. Dies ist bei verteilten Objekten beim Schreiben und beim Lesen und für replizierte Objekte beim Lesen der Fall.

Zusätzlich muss man berücksichtigen, dass die Operationen NEW, REF, DEREF, DEALLOCATE für replizierte und verteilte Objekte für die ganze Maschine *atomar* und somit *nicht parallel* ausgeführt werden müssen, weil bei Verwaltungsoperationen auf diesen Objekten immer alle Prozessoren betroffen sind.

6.3.2. Auswahl der geeigneten Objektklasse

Auf Grund der gezeigten Zugriffscharakteristiken lässt sich nun ein Katalog von Regeln aufstellen, nach denen die Klasse eines Objekts bestimmt werden kann:

1. Grundsätzlich sind alle Objekte *lokal* zu allozieren.

2. Grosse Objekte, auf die von vielen Prozessoren parallel zugegriffen wird, sollten *verteilt* angelegt werden. Insbesondere gilt diese Regel für Eingabe- und Resultatarrays von FORALL-Schleifen.

3. Kleinere Objekte, auf die parallel zugegriffen wird, können unter Umständen *repliziert* angelegt werden. Der Compiler verwendet automatisch replizierte Objekte für das In-Objekt von äusseren PARCALLs, weil dort in der Regel die Bedingung erfüllt ist, dass das Objekt nur einmal geschrieben, aber häufig gelesen wird.

4. Ausserdem werden replizierte Objekte für spezielle Zwecke wie für Code oder Konstanten verwendet.

Mit *grossen Objekten* sind Objekte gemeint, die mindestens soviele Elemente haben wie die Maschine Prozessoren. Kleinere Objekte können gar nicht vollständig verteilt werden. Zudem würden grosse Objekte als replizierte Objekte viel zu viel Speicherplatz beanspruchen.

In der heutigen, experimentellen Version des MFL-Compilers kann der Programmierer für jede Variable die Objektklasse selbst wählen (vgl. Abschnit 8.3. "Benutzerschnittstelle"). Es ist aber für eine spätere Version ohne weiteres denkbar, die Anwendung der oben gezeigten Regeln zu automatisieren.

6.4. Objekte mit Unterobjekten

6.4.1. Abbildung von Typbäumen auf Objektbäume

MFL lässt beliebig verschachtelte Tupel und Arrays zu (vgl. Abschnitt 3.3 "Deklarationen"). Man kann solche *strukturierten Typen* als *Typbäume* darstellen, deren Blätter atomaren Typen und deren innere Knoten komplexen Typen entsprechen. Eine Variable komplexen Typs wird durch eine entsprechende Objektstruktur im Speicher dargestellt. Es gibt verschiedene Möglichkeiten, einen Typbaum auf einen entsprechenden Objektbaum abzubilden: Eine Möglichkeit ist, den ganzen Objektbaum zu linearisieren und auf ein einziges Objekt abzubilden. Andererseits kann der ganze Typbaum direkt auf einen Objektbaum abgebildet werden, indem für jeden inneren Knoten im Falle eines Tupels ein oder im Falle eines Arrays mehrere Objekte alloziert werden. Ausserdem ist jede Zwischenform möglich, bei der ein Teilbaum des Typbaums auf ein einziges Objekt abgebildet wird. Die Figur 6.1 zeigt anhand des Beispiels einer Matrix von komplexen Zahlen, wie die MFL-Typdeklarationen und der zugehörige Typbaum aussehen.

```
TYPE  Complex =   TUPLE
                        re, im: REAL;
                  ENDTUPLE;
      Vektor =    ARRAY [1..3] OF
                        Complex
                  ENDARRAY;
      Matrix =    ARRAY [1..3] OF
                        Vektor
                  ENDARRAY;
```

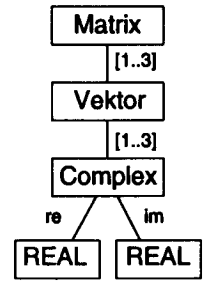


Fig. 6.1: Typ-Baum für eine Matrix komplexer Zahlen

In klassischen Programmiersprachen [Wirth85] steuert der Programmierer diese Abbildung, indem er nach Bedarf Zeigertypen verwendet und sich selbst um die Allokation bzw. Deallokation der entsprechenden Objekte kümmert. In MFL gibt es auf Sprachebene keine Zeigertypen; der Compiler setzt sie aber auf Maschinenebene prinzipiell für alle Objekte strukturierten Typs ein (vgl. Abschnitt 6.1 "Strukturierte Werte in MFL"). Der Programmierer hat aber bei allen komplexen Typen die Möglichkeit anzugeben, ob er die *Unterfelder expandieren* will oder nicht. Expansion der Unterfelder für einen Typ bedeutet, dass der entsprechende Knoten im Typbaum und alle seine direkten Nachfolger auf einen einzigen Knoten im Objektbaum abgebildet werden.

In der Figur 6.2 werden alle möglichen Abbildungen des in Figur 6.1 gezeigten Typ-Baums in Objekt-Bäume gezeigt, wobei die Zahlen in den Knoten die Grösse des entsprechenden Objekts bezeichnen. Für die Blätter im Objektbaum ist zusätzlich ihre innere Struktur angegeben.

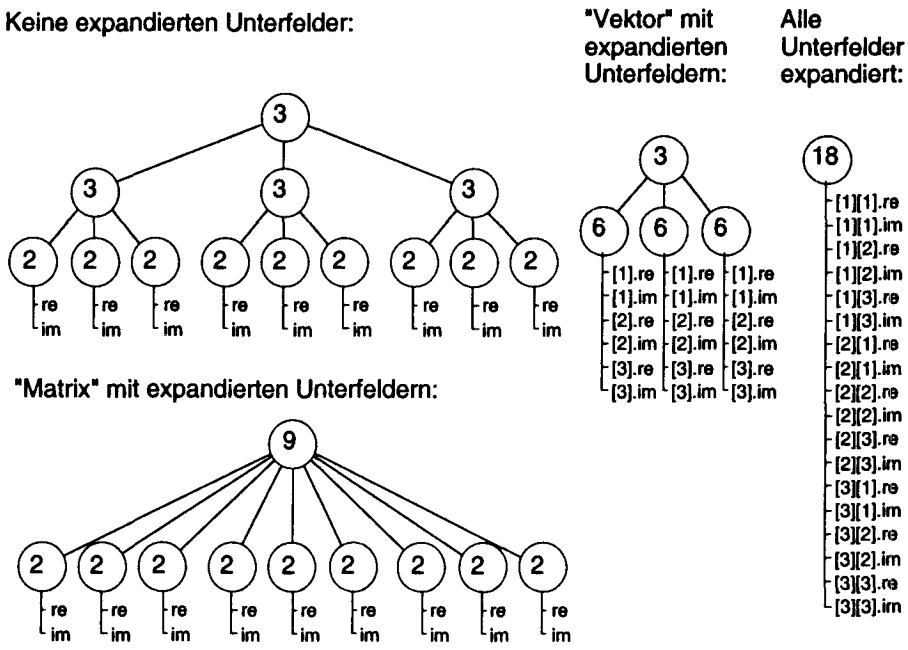


Fig. 6.2: Alle möglichen Abbildungen von "Matrix" auf einen Objektbaum

Es stellt sich nun die Frage, weshalb diese Abbildung überhaupt flexibel gemacht wird und nicht eine Standardmöglichkeit (alles oder nichts expandieren) angewendet wird. Der Grund dafür ist, dass beide Möglichkeiten abhängig vom Zugriffsmuster auf das Objekt grosse Vor- und Nachteile gegenüber der alternativen Lösung haben. Die Objektverwaltung (Allozieren, Deallozieren) kostet für Objektbäume bedeutend mehr als für Einzelobjekte, was besonders ins Gewicht fällt, wenn man bedenkt, dass viele Objektverwaltungsinstruktionen maschinenweit atomar ausgeführt werden müssen. Auf der anderen Seite muss für expandierte Typen bei der Zuweisung eines strukturierten Unterfelds kopiert werden, was ebenfalls hohe Kosten verursacht. Somit hängt der Entscheid, ob ein strukturierter Typ mit expandierten Unterfeldern angelegt werden soll, von der Anwendung ab. Die Experimente in Abschnitt 6.5 "Experimente mit dem ADAM-Simulator" belegen diese Aussage.

Damit zur Laufzeit ganze Objektbäume alloziert und dealloziert werden können, muss der Compiler Typ-Deskriptoren generieren, anhand welcher ein Laufzeitsystem mit Hilfe der primitiven Maschinenoperationen NEW, REF, DEREF und DEALLOCATE ganze Objektbäume verwalten kann. Die Abschnitte 6.3.2 "Typ-Deskriptoren" und 6.3.3 "Verwaltung von Objekten mit Unterobjekten" zeigen,

wie diese Typ-Deskriptoren bzw. das entsprechende Laufzeitsystem im MFL-Compiler realisiert sind.

6.4.2. Typ-Deskriptoren

Alle Typ-Deskriptoren eines MFL-Programms werden in einem konstanten Array von ganzen Zahlen nach der folgenden Syntax angelegt:

TypeDesc	::=	ObjectSize (*Array* | *Tuple*)
Array	::=	AStart RepCount Intervall (*Array* | *Tuple* | Size | DescIndex)
Tuple	::=	TStart FieldCount { Index (*Array* | *Tuple* | Size | DescIndex) }

Die Terminalsymbole "ObjectSize", "AStart", "TStart", "Size", "DescIndex", "Index", "FieldCount" und "RepCount" sind ganze Zahlen, wobei die folgenden Bedingungen gelten:

AStart	= 1
TStart	= 0
DescIndex	< 0, wobei -DescIndex immer ein gültiger Index im Deskriptor-Array sein muss.
Size	> 1
ObjectSize, RepCount, FieldCount	> 0
Index	≥ 0

Die vier Werte für "AStart", "TDelimit", "Size" und "DescIndex" sind ohne überschneidenden Bereich gewählt, sodass die gezeigte Syntax immer eindeutig geparst werden kann. Ein ganzer Typ-Deskriptor ("TypeDesc") enthält immer als erstes Wort die Grösse des Objekts, welches er beschreibt. Darauf folgt die Beschreibung der Struktur des Typs. Ein strukturierter Typ kann entweder ein Array oder ein Tupel sein. Das erste Symbol der Beschreibung ("AStart" oder "TStart") erlaubt die Unterscheidung der beiden Fälle. Im Falle eines Arrays folgen dann die Anzahl der Felder des Arrays ("RepCount"), die Grösse eines einzelnen Feldes ("Intervall") und schliesslich der Feldtyp. Im Falle eines Tupels folgt die Anzahl Felder ("FieldCount") und die entsprechende Anzahl Paare von Feldindex[1] ("Index") und Feldtyp. Der Feldtyp kann entweder ein Tupel oder ein Array im gleichen Objekt, ein Deskriptor-Index ("DescIndex") eines weiteren Unterobjekts mit Unterobjekten oder die Grösse ("Size") eines nicht weiter strukturierten Unterobjekts sein.

---

[1]) Der Index gibt nicht den absoluten Index des Feldes im gesamten Objekt, sondern nur den Index des Feldes relativ zum Ursprung des Tupels im Objekt an. Der Grund für diese Festlegung ist, dass so ohne weiteres ein Tupel als Feldtyp eines Arrays verwendet werden kann.

In der Figur 6.3 sind die Typ-Deskriptoren für die verschiedenen Objektbäume aus Figur 6.2 dargestellt.

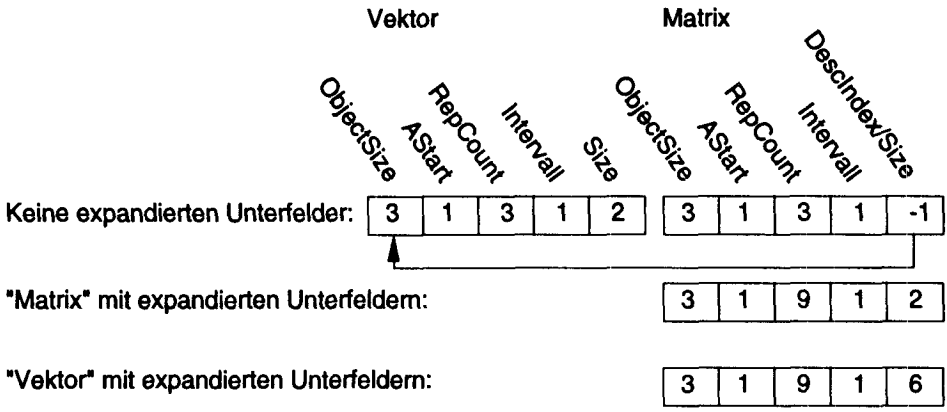


Fig. 6.3: Beispiele für Typ-Deskriptoren

Eine Besonderheit dieser Typ-Deskriptoren ist, dass für einfache strukturierte Typen ohne Unterobjekte kein Deskriptor angelegt werden muss, da diese in den Deskriptoren direkt als Grösse ("Size") vorkommen. Die Praxis hat gezeigt, dass der grösste Teil aller strukturierten Typen keine Unterobjekte hat, sodass mit dieser Methode viel Platz gespart werden kann. Dies ist auch der Grund dafür, dass für die Variante "Alle Unterfelder expandiert" in Figur 6.2 kein Typ-Deskriptor benötigt wird.

6.4.3. Verwaltung von Objekten mit Unterobjekten

Für Objekte mit Unterobjekten stehen die Verwaltungsoperationen NEW und DEALLOCATE wegen ihrer Komplexität nicht direkt auf der Ebene der ADAM-Architektur zur Verfügung. Sie müssen deshalb durch ein Laufzeitsystem in Software zur Verfügung gestellt werden.

Die NEW-Funktion für ein Objekt mit Unterobjekten macht im Prinzip nichts anderes, als den entsprechenden Typ-Deskriptor rekursiv zu parsen und mit Hilfe der primitiven NEW-Instruktionen einen Objektbaum aufzubauen.

Entsprechend parst die DEALLOCATE-Funktion den Typ-Deskriptor und baut so den Objektbaum wieder ab. Dies tut sie allerdings nur dann, wenn der Referenzzähler für das Objekt wirklich auf 0 zurückgeht. Sonst wird nur der Referenzzähler des obersten Objekts dekrementiert.

Diese Laufzeitsystem-Funktionen wurden in Assembler für die ADAM-Architektur implementiert.

6.4.4. Der BLOCKMOVE-Knoten

Bei Zuweisungen komplexer Unterfelder in Objekten mit expandierten Unterfeldern reicht die einfache Store-Operation nicht aus, da ganze Teile von Objekten kopiert werden müssen. Zu diesem Zweck wurde der BLOCKMOVE-Knoten (vgl. Abschnitt 4.5 "Knoten zur Handhabung von Datenstrukturen") eingeführt.

Auch für den BLOCKMOVE-Knoten gibt es keine direkt entsprechende Instruktion im ADAM-Instruktionssatz. Das Laufzeitsystem sieht deshalb eine Funktion "BlockMove" (vgl. Anhang C) vor.

Kopieren von Daten ist eine hochparallele Operation und es gilt, dieser Eigenschaft durch eine entsprechend parallele Implementation von "BlockMove" im Laufzeitsystem Rechnung zu tragen. Als Erstes gilt es dabei, zur Laufzeit zwischen den verschiedenen Objektklassen (vgl. Abschnitt 2.5. "Der Objekt-Manager: Ein physisch verteilter, logisch gemeinsamer Speicher" und 6.3 "Auswahl der Objektklasse") von Quellen- und Zielobjekt zu unterscheiden. Interessant ist ein paralleles Kopieren nur dann, falls die notwendige Zugriffsbandbreite auf das Objekt überhaupt zur Verfügung steht. Dies trifft nur zu, falls das Quellenobjekt entweder verteilt oder repliziert und das Zielobjekt verteilt ist. Ausserdem muss der Code so geschrieben sein, dass die Speicherlatenz möglichst gut mit möglichst wenigen Kontextwechseln versteckt werden kann.

6.5. Experimente mit dem ADAM-Simulator

Im folgenden Abschnitt werden für die Programme "Blockmove", "Gauss", "Matmul" und "Matmul2" die Abbildung der Typbäume auf Objektbäume und die Objektklassen variiert, um den Einfluss dieser Entscheidungen auf die Programmlaufzeit zu zeigen.

6.5.1. Experiment mit "BlockMove"

In diesem Experiment wird der Inhalt eines Objekts vollständig in ein neues Objekt kopiert, wobei die Grösse und die Klasse des Objekts variiert werden. Es werden zwei Implementationsvarianten für "Blockmove" verglichen. Die "optimierte" Variante entspricht der in Anhang C "Assembler-Code für die Laufzeitfunktionen" gezeigten Implementation. Die "einfache" Variante ist ein primitive,

sequentielle Schleife mit Schleifenzähler für den Index und je einer LD- und STO-Instruktion im Schleifenkörper.

Quellenobjekt		Zielobjekt					
		lokal		verteilt		repliziert	
Klasse	Grösse	optimiert	einfach	optimiert	einfach	optimiert	einfach
lokal	1	490	360	997	810	986	808
	10	1046	893	1570	1610	1366	1228
	100	4770	5746	5421	10043	5151	6101
	1000	41245	54800	48226	99266	43528	56190
	10000	405335	545574	470934	986583	425197	555497
verteilt	1	992	872	*1626*	*873*	1021	871
	10	1486	2171	*1835*	*1881*	1743	2458
	100	4617	14666	*2615*	*14986*	4893	15043
	1000	39111	143588	*6811*	*150242*	40057	144865
	10000	380066	1429644	*31552*	*1499596*	384860	1440101
repliziert	1	939	809	*1573*	*810*	968	808
	10	1450	1297	*2481*	*1945*	1449	1305
	100	5220	6196	*3636*	*10493*	5380	6294
	1000	41683	55260	*7984*	*99592*	43634	56528
	10000	405791	546030	*33934*	*987365*	425760	556021

Tab. 6.2: Laufzeiten für Blockmove auf 16 Prozessoren (in Zyklen)

Am augenfälligsten ist der Unterschied zwischen den beiden Varianten dort, wo das Kopieren parallel ablaufen kann. Dies ist der Fall für Objekte mit skalierbarer Zugriffsbandbreite (vgl. Abschnitt 6.3.1. "Zugriffscharakteristik der verschiedenen Objektklassen"), also für verteilte und replizierte Objekte beim Lesen sowie für verteilte Objekte beim Schreiben. Die entsprechenden Kombinationen sind in der Tabelle 6.2 kursiv hervorgehoben.

Aber auch in den anderen Fällen ist die optimierte Variante zumindest für grössere Objekte deutlich schneller. Dies liegt daren, dass in dieser Implementation die Objekt-Manager-Anforderungen immer in Paketen zu 16 ausgegeben und wieder synchronisiert werden. Es kann somit im Maximum für jeden 16. Speicherzugriff zu einem Kontextwechsel kommen und die Speicherlatenz kann durch weitere Speicherzugriffe besser versteckt werden. Verschärfend wirkt sich aus, dass neben dem Kopieren kein weiteres Programm auf der Maschine läuft und damit keine Speicherlatenz durch ein fremdes Programm versteckt werden kann.

Für Objekte der Grössen 1 und 10 ist teilweise die einfache Variante schneller. Der Grund dafür ist, dass das Kopieren im Experiment nur je einmal ausgeführt wird. Der Code für die optimierte Variante ist bedeutend länger als der Code für die einfache Variante, was zur Folge hat, dass es für die optimierte Variante länger dauert, bis der Code vollständig im Sequencer-Cache (vgl. Abschnitt 2.7. "Der Simulator: Die verwendete Implementation der ADAM-Architektur") vorhanden ist. Wiederholt man den Aufruf von Blockmove genügend oft, so sinkt die Kopierzeit für ein Objekte der Grösse 1 unabhängig von der Variante auf ca. 400 Zyklen ab, was dem Aufwand für den Aufruf des Blockmove-Codeblocks entspricht.

Absolut gesehen, sind die Kosten für das Kopieren mit ca. 40 Zyklen pro Wort auch für die optimierte Variante recht hoch. Der Laufzeitaufwand für das komplexe Speichersystem macht sich deutlich bemerkbar.

6.5.2. Experimente mit der Matrix-Multiplikation

Im nächsten Experiment (Resultate in Tabelle 6.3) wird die Matrix-Multiplikation auf 64 und 256 Prozessoren ausgeführt, wobei die Objektklasse und Abbildung der Matrix auf den Objektbaum variiert wird. Für nicht expandierte Matrizen sind sowohl die gesamte Matrix als auch die einzelnen Zeilen Objekte der angegebenen Klasse. Die jeweils schnellste Version ist kursiv hervorgehoben.

Das wichtigste Ergebnis dieses Experiments bestätigt die Hypothese, dass lokale Objekte keine skalierbare Zugriffsbandbreite haben, auf eindrückliche Weise. Alle Versuche mit lokaler Quellenmatrix führen auf 64 Prozessoren zu Laufzeiten, die um mindestens einen Faktor 15 über den anderen Varianten liegen. Für 256 Prozessoren wurden diese Versuche gar nicht durchgeführt (***), die Ergebnisse wären aber noch deutlicher.

Weniger deutlich sind die Unterschiede für verschiedene Objektklassen bei der Zielmatrix. Bei einer Matrix-Multiplikation der Dimension n wird jedes Element der Quellenmatrix 2n-mal gelesen, die Zielmatrix jedoch nur einmal geschreiben. Die Schreibbandbreite der Zielmatrix spielt deshalb keine grosse Rolle. Normalerweise überwiegt der Vorteil der einfacheren Adressrechnung für lokale und replizierte Objekte.

Widersprüchlich sind die Antworten auf die Frage, ob die Matrix als einzelnes Objekt oder als Objekt mit Unterobjekten angelegt werden soll: Für 256 Prozessoren ist eine expandierte Variante am schnellsten, für 64 Prozessoren eine mit Unterobjekten.

Bei der Anzahl der Kontextwechsel zeigt sich allerdings der Unterschied zwischen den beiden Varianten deutlich. Die Variante mit Unterobjekten verursacht annähernd doppelt soviele Kontextwechsel wie die expandierte Variante, was durchaus den Erwartungen entspricht, sind doch die Lesezugriffe auf die Arrays zwei- bzw. einstufig und können dementsprechend zwei oder einen Kontextwechsel zur Folge haben. Eine Ausnahme bilden die replizierten Quellenmatrizen, die bei den Ladeoperationen keine Kontextwechsel verursachen können, da die Zugriffe auf jedem Prozessor lokal sind.

Die schlechte Leistung für verteilte Quellenmatrizen mit Unterobjekten auf 256 Prozessoren lässt sich dadurch begründen, dass die verteilten Objekte kleiner als die Anzahl Prozessoren sind und sich somit nicht auf die ganze Maschine verteilen lassen. Die Objekte sind zu klein nach der Definition in Abschnitt 6.3.2."Auswahl der geeigneten Objektklasse".

Warum sind für 64 Prozessoren die Varianten mit Unterobjekten in der Regel besser als die expandierten Varianten, obwohl die Matrix nirgends im Programm zeilenweise gefüllt wird, was bei der expandierten Variante Kopieroperationen zur Folge hätte (vgl. Abschnitt 6.4.4. "Der BLOCKMOVE-Knoten")? Der Grund dafür ist, dass die Indexbereiche beim Typ Matrix optimal für eine Variante mit Unterobjekten zugeschnitten sind, da sie bei 1 beginnen. Für die expandierte Version verursacht der Indexursprung bei 1 für die Zeilen bei jedem Zugriff zusätzliche Additionen. Eine leicht veränderte Version von Matmul, deren Typ "Zeile" den Ursprung bei 0 hat benötigt für die Variante "verteilt, verteilt, expandiert" auf 64 Prozessoren nur 478649 Zyklen und ist somit nur unwesentlich langsamer als die beste Variante mit Unterobjekten.

Quellen-matrizen	Ziel-matrix	Typ Matrix	64 Prozessoren			256 Prozessoren		
			Zyklen absolut	Zyklen relativ [%]	Kontext wechsel	Zyklen absolut	Zyklen relativ [%]	Kontext wechsel
lokal	lokal	nicht exp.	10475155	2349	574868	***	***	***
lokal	verteilt	nicht exp.	10477996	2350	576537	***	***	***
lokal	repliziert	nicht exp.	10470556	2348	571917	***	***	***
verteilt	lokal	nicht exp.	483167	108	558827	377673	238	578052
verteilt	verteilt	nicht exp.	478182	107	558142	348467	220	588150
verteilt	repliziert	nicht exp.	485499	109	559800	376668	237	588662
repliziert	lokal	nicht exp.	*445864*	*100*	13192	178562	112	13420
repliziert	verteilt	nicht exp.	450674	101	13184	179268	113	13434
repliziert	repliziert	nicht exp.	454575	102	13211	177736	112	13439
lokal	lokal	expandiert	6777222	1520	316609	***	***	***
lokal	verteilt	expandiert	6722431	1508	316841	***	***	***
lokal	repliziert	expandiert	6703615	1504	312507	***	***	***
verteilt	lokal	expandiert	517169	116	335016	158840	100	336281
verteilt	verteilt	expandiert	520089	117	335607	161490	102	336358
verteilt	repliziert	expandiert	518599	116	330246	*158739*	*100*	334055
repliziert	lokal	expandiert	541511	121	13816	197788	125	14426
repliziert	verteilt	expandiert	549935	123	13807	195279	123	14447
repliziert	repliziert	expandiert	549132	123	10225	200193	126	11895

Tab. 6.3: Resultate für Matmul auf 64 und 256 Prozessoren

Im obigen Experiment waren die Zeilen und die Matrix immer von der gleichen Objektklasse. Es stellt sich die Frage, ob es nicht sinnvoll wäre, für Zeilen und Matrix unterschiedliche Objekte zu verwenden. Zu diesem Zweck dient das Programm "Matmul2" (vgl. Anhang B.11), welches die Matrix zeilenweise generiert und multipliziert. Um den Vergleich zwischen den verschiedenen Varianten gerecht zu machen, wurden in diesem Beispiel die Schleifeninvarianten-Optimierung nicht ausgeführt, da sonst bei der Initialisierung der Matrix nur eine Zeile mit Einsen generiert und dann 64 mal abgespeichert würde.

In Tabelle 6.4 sind die Ergebnisse dieses Experiments dargestellt. Sie weichen nicht wesentlich von den Resultaten in der Tabelle 6.3 ab.

Zeilen	Matrix	Typ Matrix	Zyklen absolut	Zyklen relativ [%]	Kontext-wechsel
lokal	verteilt	nicht exp.	679617	150	627022
lokal	repliziert	nicht exp.	552032	122	304463
verteilt	verteilt	nicht exp.	552492	122	626123
verteilt	repliziert	nicht exp.	*453583*	*100*	307465
lokal	verteilt	expandiert	654658	144	310665
verteilt	verteilt	expandiert	648719	143	309687

Tab. 6.4: Resultate für Matmul2 auf 64 Prozessoren

Wir haben eine Reihe von weiteren Experimenten mit der Matrix-Multiplikation und variierenden Maschinen-Parametern (Netzwerk-Topologien, Latenzen, verschiedene Objektklassen) durchgeführt. Die Resultate sind eine interessante Ergänzung zu den hier gezeigten und können in [MurFär92] gefunden werden.

Eine entscheidendes Kriterium für die Effizienz eines parallelen Codes ist der erzielbare Speedup gegenüber einer guten sequentiellen Version. Für die sequentiellen Versionen mit lokalen Objekten von "Matmul" und "Matmul2" wurden Zeiten von 22779810 bzw. 25337152 Zyklen gemessen, was einem Speedup von 51 (MatMul), 56 (MatMul2) bzw. 144 (MatMul auf 256 Prozessoren) entspricht[1]. Dies ergibt eine Effizienz zwischen 56% und 88%, was für ein Programm, welches während weniger als 10^6 Zyklen oder 50 ms bei 20MHz Taktfrequenz läuft, ein gutes Ergebnis ist.

6.5.3. Experimente mit Gauss

In diesem Experiment (Resultate in Tabelle 6.5) wurden die Laufzeiten und die Anzahl der Kontextwechsel für das Programm "Gauss" (vgl. Anhang B.8.) auf 64

[1] Zusätzlich muss man betonen, dass es sich im Gegensatz zu vielen anderen Experimenten bei der sequentiellen Version nicht einfach denselben Code sondern um eine für die sequentielle Ausführung optimierte Version handelt, so dass einiger Aufwand für die Parallelität wegfällt.

Prozessoren gemessen, wobei alle sinnvollen[1] Kombinationen von Objektklassen und Expansion von Unterobjekten berücksichtigt wurden. Tabelle 6.5 zeigt die Ergebnisse des Experiments in der Uebersicht. Die beste Variante mit verteilten Objekten für den Vektor "z" in der Funktion "Pivotzeile" und die Matrix "newA", lokalen Objekten für die Vektoren "z" in der Funktion "Pivotspalte" sowie "zzz" und Unterobjekten für die Zeilen der Matrix ist kursiv hervorgehoben. Die Zahlen in der Spalte "relativ [%]" geben die Laufzeit für die entsprechende Variante in Prozent der schnellsten Variante an.

[1]) Mit "sinnvoll" sind all jene Kombinationen gemeint, die nicht a priori zu schlechten Ergebnissen führen. So wurden beispielsweise die Zeiten für jene Varianten, die für die Matrix ein lokales Objekt vorsehen, nicht gemessen, da diese offensichtlich zu einem "hot spot" führen würden.

Pivot-zeile.z	Pivot-spalte.z	newA	zzz	Typ Matrix	absolut	relativ [%]	Kontext wechsel
lokal	lokal	verteilt	lokal	nicht expandiert	3924366	179	784808
lokal	lokal	verteilt	verteilt	nicht expandiert	4069367	185	767348
lokal	verteilt	verteilt	lokal	nicht expandiert	3871455	176	782670
lokal	verteilt	verteilt	verteilt	nicht expandiert	4059631	185	768288
verteilt	*lokal*	*verteilt*	*lokal*	*nicht expandiert*	*2195454*	*100*	*738788*
verteilt	lokal	verteilt	verteilt	nicht expandiert	3930395	179	724749
verteilt	verteilt	verteilt	lokal	nicht expandiert	2207858	101	738370
verteilt	verteilt	verteilt	verteilt	nicht expandiert	3924704	179	726018
lokal	lokal	repliziert	lokal	nicht expandiert	4200095	191	465624
lokal	lokal	repliziert	verteilt	nicht expandiert	4245973	193	498342
lokal	verteilt	repliziert	lokal	nicht expandiert	4222746	192	465488
lokal	verteilt	repliziert	verteilt	nicht expandiert	4260303	194	497756
verteilt	lokal	repliziert	lokal	nicht expandiert	2428337	111	538637
verteilt	lokal	repliziert	verteilt	nicht expandiert	3969106	181	562062
verteilt	verteilt	repliziert	lokal	nicht expandiert	2421459	110	528947
verteilt	verteilt	repliziert	verteilt	nicht expandiert	3954389	180	562242
lokal	lokal	verteilt	lokal	expandiert	4464164	203	522236
lokal	lokal	verteilt	verteilt	expandiert	3943988	180	592119
lokal	verteilt	verteilt	lokal	expandiert	4444000	202	523744
lokal	verteilt	verteilt	verteilt	expandiert	3926902	179	590892
verteilt	lokal	verteilt	lokal	expandiert	4079223	186	591523
verteilt	lokal	verteilt	verteilt	expandiert	3827468	174	644316
verteilt	verteilt	verteilt	lokal	expandiert	4092944	186	561626
verteilt	verteilt	verteilt	verteilt	expandiert	3824610	174	642940

Tab. 6.5: Resultate für Gauss auf 64 Prozessoren

In der Figur 6.4 sind die Varianten nach aufsteigender Laufzeit sortiert, wobei die Balken die relative Laufzeit einer Variante angeben. Die Abkürzungen entlang der x-Achse beziehen sich auf die Anfangsbuchstaben der ersten 5 Spalten in der Tabelle 6.5, um die Varianten zu identifizieren.

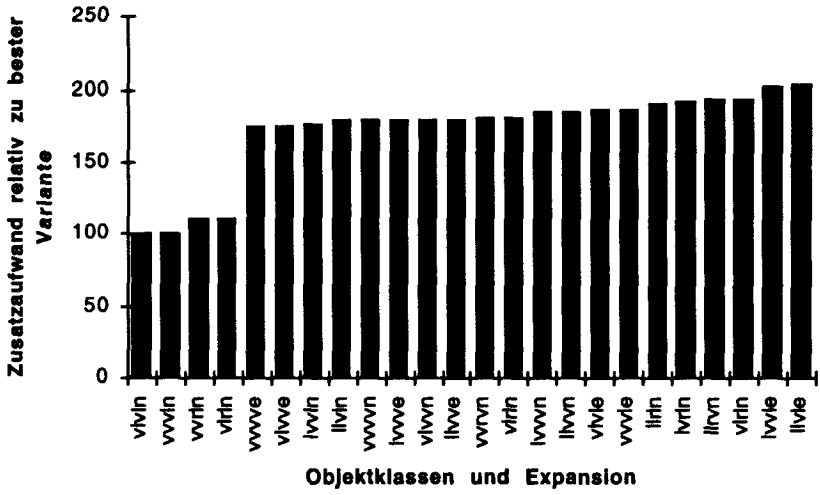


Fig. 6.4: Verschiedene Versionen von Gauss im Vergleich

Es ist nicht offensichtlich, welche Parameter beim Programm "Gauss" die dominante Rolle spielen. Obwohl keine der "sinnvollen" Varianten stark aus dem Rahmen fällt (die langsamste Variante ist etwas weniger als halb so schnell wie die beste), gibt es hinter den vier schnellsten Varianten eine deutliche Zäsur. Betrachtet man die vier schnellen Varianten etwas näher, so fällt auf, dass bei allen "Pivotzeile.z" verteilt und "zzz" lokal angelegt sind, wobei die Varianten mit verteilter Matrix etwas besser abschneiden. "zzz" muss lokal sein, da sich sonst die Allokationen gegenseitig behindern[1], weil "zzz" innerhalb der äusseren FORALL-Schleife in der Funktion "Austausch" parallel generiert wird. "Pivotzeile.z" muss hingegen in ein verteiltes Objekt gespeichert werden, da das gleiche Objekte als "pivotZ" in der Funktion "Austausch" in der innersten FORALL-Schleife häufig zugegriffen wird. Da die im innersten IF-Ausdruck nur selten der ELSE-Fall eintritt spielt es keine Rolle, wie "PivotSpalte.z" alloziert ist.

Es ist keine Ueberraschung, dass keine Variante mit expandierter Matrix unter den schnellsten zu finden ist, da in "Gauss" die Matrix zeilenweise geschrieben wird.

---

[1]) Pro memoria: Allokationen von verteilten und replizierten Objekten sind atomare Aktionen.

Auch für "Gauss" stellt sich die Frage, wie effizient dieses Programm parallelisiert wurde. Eine sequentielle Version mit lokalen Objekten läuft auf einem Prozessor in 38459369 Zyklen, so dass der Speedup für die beste Version in Tabelle 6.5 17.5 beträgt, was einer Effizienz von 27% entspricht. Dieses Ergebnis ist bedeutend schlechter als für die Matrix-Multiplikation. Die zwei Gründe dafür sind:

1. Die Funktion "Austausch" hat einen schwach parallelen Teil am Anfang (Amdahl).

2. Die ganze Parallelität muss 64-mal auf- und abgebaut werden, wobei dies bedingt durch die Methode der Lastverteilung auf der ADAM-Architektur nicht beliebig schnell geschehen kann. Das Problem ist also eher zu klein für eine Maschine mit 64 Prozessoren.

6.6. Arbeiten mit ähnlichen Ansätzen

Aus der Literatur [Cohen81] sind unter dem Begriff "Garbage Collection" zwei grundsätzlich verschiedene Methoden bekannt, um die Lebensdauer von Heap-Objekten zu kontrollieren: Bei der *"Mark und Scan"-Methode* wird in einem ersten Durchgang jedes von einer gemeinsamen Wurzel aus erreichbare Objekt markiert und im zweiten Durchgang werden die nicht markierten Objekte freigegeben. Bei der von uns verwendeten *Referenzzähler-Methode* ("Reference Counting") wird für jedes Objekt ein Zähler für alle Referenzen auf dieses Objekt nachgeführt. Beide Methoden haben ihre Vor- und Nachteile: Die Referenzzähler-Methode hat Probleme mit zyklischen Referenzen und bei der "Mark und Scan"-Methode muss der Ablauf des Programms von Zeit zu Zeit unterbrochen werden, um ungebrauchte Speicherplätze einzusammeln. Da in den Typ-Bäumen von MFL keine Zyklen zugelassen sind, wurde die Referenzzähler-Methode gewählt; ihr wesentlichster Nachteil fällt damit nicht ins Gewicht.

Die Methode, strukturierte Werte in funktionalen Programmen als Heap-Objekte mit Referenzzählern zu implementieren, und somit Kopierkosten bei Zuweisungen zu sparen wird von verschiedenen bekannten Implementationen verwendet [ArNiPi86, Ackerm84, Cann89]. Es gibt eine Reihe von Arbeiten, die sich in ähnlicher Weise mit der Optimierung von Referenzzähleroperationen auf komplexen Objekten in funktionalen Sprachen beschäftigen. Im SISAL-Projekt wurde ein früher Prototyp [SkeSim88] und eine verbesserte Version [Cann89] eines Optimierers implementiert. Der wesentliche Unterschied zwischen der ersten und der zweiten Version besteht darin, dass die erste Version die volle Parallelität bis auf

die Ebene der Knoten erhält, wogegen die zweite Version zusätzliche, künstliche Kanten im Datenflussgraph einführen kann, um auf Kosten von Parallelität Referenzzähler-Operationen zu sparen. In [Hudak86] wird ein semantisches Modell für die Referenzzähler-Operationen gegeben. Die in diesem Kapitel vorgestellte Methode unterscheidet sich insofern von den besprochenen Ansätzen, als von Anfang an die Referenzzähler nicht als eigentliche Zähler aller Referenzen auf ein Objekt, sondern als ein jedem Objekt zugeordneter Zahlenwert betrachtet wird, für welchen eine gewisse Invariante nicht verletzt werden darf.

Von grosser Bedeutung für die effiziente Implementation funktionaler Sprachen ist auch die Frage, ob, falls ein Objekt kreiiert wird, das bis auf ein verändertes Unterfeld genau einem bereits bestehenden Objekt entspricht, dieses kopiert werden muss, um die funktionale Semantik zu erhalten. In [Ackerm84] taucht zum ersten Mal die Idee auf, Objekte mit nur einer Referenz (Referenzzähler = 1) nicht zu kopieren, sondern direkt zu verändern. Diese "Update in Place"-Analyse wurde im Laufe der Jahre zusätzlich verfeinert [Cann89, HudBlo85] und ist heute die wichtigste Voraussetzung dafür, dass funktionale Programme für numerische Applikationen mit vergleichbarer Effizienz ausgeführt werden können wie konventionelle Programme [CanFeo90]. In MFL wird das Problem umgangen, indem Arrays innerhalb einer Schleife nicht als vollständig funktionale Objekte betrachtet werden, sondern als Datenstrukturen, die elementweise generiert werden.

Die verteilten Objekte sind nichts grundsätzlich Neues, sondern tauchen in der Form von mehreren Speicherbanken [HwaBri84] schon in den ersten Vektorprozessoren auf, um die Speicherbandbreite zu erhöhen. Der wesentliche Unterschied zu diesen Lösungen besteht darin, dass in der ADAM-Architektur nicht der ganze Speicher ein verteiltes Objekt ist, sondern dass die verschiedenen Objekt-Klassen vom Programmierer oder vom Compiler entsprechend den Bandbreite-Bedürfnissen gezielt eingesetzt werden können. In diesem Zusammenhang taucht sofort die Frage auf, ob nicht Indexintervalle, die Vielfache der Anzahl Prozessoren sind, zu Bankkonflikten führen. Dieses Problem ist nicht so gravierend wie auf konventionellen Maschinen, da die Speicherlatenz versteckt werden kann. Doch auch das Verstecken der Speicherlatenz verhindert nicht, dass die Zugriffsbandbreite auf ein verteiltes Objekt zusammenbricht, falls immer zu Elementen auf dem gleichen Prozessor zugegriffen wird ("hot spot").

Leer - Vide - Empty

7. Sequentialisierung und Codegenerierung

Datenflussgraphen bestimmen nur eine partielle Ordnung für die Ausführung der einzelnen Operationen. Damit das Programm auf der ADAM-Architektur ausgeführt werden kann, muss für die einzelnen Codeblöcke eine Sequenz der Operationen festgelegt werden. Dieses Kapitel behandelt die Frage, wie diese Sequenz im Rahmen der gegebenen, partiellen Ordnung festgelegt werden soll, so dass die Ressourcen der Maschine möglichst gut ausgenutzt werden können.

In einem ersten Teil wird beschrieben, welche Arten von Parallelität von der ADAM-Architektur ausgenutzt werden können. Dann stellen wir uns die Frage, wie die gerichteten azyklischen Datenflussgraphen so in sequentiellen Code übersetzt werden, dass die Möglichkeiten zur Parallelausführung auf der ADAM-Maschine gut genutzt werden. Im nächsten Teil besprechen wir die Methoden der Registerzuordnung innerhalb von Graphen und über die Grenzen von einzelnen Untergraphen hinaus. Zudem wird beschrieben, wie Code für den Kontrollfluss in zusammengesetzten Knoten generiert wird. Zuletzt werden die verwendeten Ansätze experimentell verifiziert und in den Kontext mit anderen Forschungsergebnissen gestellt.

7.1. Parallelität auf der ADAM-Architektur

Computer können ganz verschiedene Ebenen von Parallelität ausnutzen. Es gibt Parallelität zwischen den einzelnen Bits eines Worts, Pipeline-Parallelität, Parallelität zwischen einzelnen funktionalen Einheiten eines Prozessors und Parallelität zwischen verschiedenen Prozessoren. Im ADAM-Prozessor ist einerseits die Parallelität zwischen den funktionalen Einheiten auf einem Prozessor (Exekutor, Objekt-Manager und Netzwerk) und andererseits die Parallelität zwischen den verschiedenen Prozessoren entscheidend.

Betrachten wir das Modell der ADAM-Architektur näher, so gibt es zwei Arten von zweiphasigen Operationen, die Parallelität generieren können: Es gibt auf der einen Seite die CALL- bzw. PARCALL-Instruktion, deren erste Phase einem FORK bzw. mehrfachen FORK und deren zweite Phase einem impliziten JOIN [PetSil85] entspricht (vgl. Kapitel 2 "Die ADAM-Architektur"). Anderseits gibt es asynchrone Objekt-Manager-Zugriffe auf nicht-lokale Objekte, um die Latenz von Speicher und Netzwerk zu verbergen. Diese zweiphasigen Instruktionen sind die einzige Quelle von Parallelität auf der ADAM-Maschine. Alle anderen Instruktionen werden in gewohnter Weise sequentiell ausgeführt.

Die Maschine führt normale, sequentielle Instruktionen (1) und zweiphasige Instruktionen (2) unterschiedlich aus:

(1) Die Folgeinstruktion einer *sequentiellen Instruktion* wird erst dann ausgeführt, wenn die vorangehende Instruktion vollständig beendet ist und ihre Resultate in das entsprechende Register zurückgeschreiben wurden.

(2) Eine *zweiphasige Instruktion* gibt nur einen Auftrag an den Objekt-Manager bei einem Zugriff auf den verteilten Speicher oder an den Token-Manager, falls ein neuer Codeblock gestartet werden soll. Auf das Resultat dieses Auftrags wird erst bei jener Instruktion gewartet, die das entsprechende Resultat-Register als Operand benutzen will (Datenfluss-Synchronisation).

Grob gesagt, geht es bei der Sequentialisierung darum, ausgehend von den partiell geordneten Instruktionen, im Datenflussgraph eine Instruktionssequenz zu finden, die möglichst viele *Zwischeninstruktionen* zwischen den beiden Phasen der zweiphasigen Instruktionen enthält. Mit diesen Zwischeninstruktionen erreicht man zwei Ziele:

(1) Falls erwartete Daten nicht vorhanden sind, wird auf der ADAM-Maschine ein Kontextwechsel ausgelöst und mit der Bearbeitung eines anderen Codeblocks weitergefahren (vgl. Abschnitt 2.3 "Lebenszyklus eines Codeblocks" und 2.4 "Das Registerframe: Synchronisation über Daten"). Diese Kontextwechsel sind zwar auf einer geeigneten Maschine billig, aber doch nicht kostenlos. Können nun mehrere Aufträge verschickt werden, bevor zum ersten Mal synchronisiert wird, so führt diese erste Synchronisation mit grosser Wahrscheinlichkeit zu einem Kontextwechsel. Sind aber genügend parallele Codeblöcke auf dem Prozessor vorhanden, so sind mit hoher Wahrscheinlichkeit alle Aufträge für einen Codeblock abgeschlossen, bis ihm der Prozessor wieder zugeteilt wird. Unter Umständen gelingt es sogar, soviele Zwischeninstruktionen zwischen die beiden Phasen einer Instruktion zu legen, dass das Resultat bereits vorhanden ist, wenn die konsumierende Instruktion ausgeführt und somit gar kein Kontextwechsel nötig wird.

(2) Die Zwischeninstruktionen können selbst wieder zweiphasige Instruktionen sein, was zu einer schnelleren Ausbreitung von Parallelität führt, da mehrere Aufträge verschickt werden können, bevor synchronisiert wird.

7.2. Von der partiellen Ordnung im Datenflussgraph zur totalen Ordnung in der Instruktionssequenz

Für die Codegenerierung aus gerichteten, azyklischen Graphen (Directed Acyclic Graphs, DAGs) existieren verschiedene Methoden. Die Literatur [AhSeUl86] kennt Methoden, die optimalen Code unter anderem unter den folgenden Einschränkungen generieren:

- begrenzte Anzahl von Registern
- Graph ist ein Baum
- Register müssen temporär in den Hauptspeicher ausgelagert werden (register spilling)

All diesen Methoden ist aber gemeinsam, dass sie für konventionelle von-Neumann-Prozessoren entworfen wurden. Für solche Prozessoren gilt die Vorgabe, dass jede Instruktion abgeschlossen ist, bevor die nächste in der Sequenz angefangen wird[1].

Für die ADAM-Maschine gibt es ein völlig neues Ziel: Die Latenz von zweiphasigen Instruktionen muss durch Einsetzen von Zwischeninstruktionen versteckt werden. Es ist ganz klar, dass dieses neue Ziel im Konflikt mit bestehenden Zielen steht. Beispielsweise wird eine derart optimierte Codegenerierungsstrategie mehr Register brauchen als eine klassische. Die Anzahl gleichzeitig erwarteter Resultate von zweiphasigen Instruktionen bestimmt nämlich gerade den Grad der erzeugten Parallelität.

7.2.1. Graph und Sequenz

Um das Problem formal zu erfassen, müssen wir, ausgehend von der Definition der Datenflussgraphen im Kapitel 4 "Datenflussgraphen als Zwischencode für MFL", einige zusätzliche Begriffe einführen:

---

[1]) In modernen Mikroprozessoren mit mehreren funktionalen Einheiten gilt diese Einschränkung zwar nicht mehr unbedingt. Sie stellen in kurzen Abschnitten die Reihenfolge der Instruktionen im Rahmen der Datenabhängigkeiten um.

Für die Betrachtungen in diesem Kapitel spielt die Zuordnung von Kanten zu bestimmten Ein- bzw. Ausgängen bei den Knoten keine Rolle. Zur Vereinfachung reduzieren wir die Kantenrelation E auf die reinen Knotenbeziehungen. Aus der ursprünglichen Kantenrelation erhalten wir diese neue Relation E' durch eine Projektion.

$N \times N \supset E'$, sodass $E' = \{(m,n) \mid (m, i, n, j) \in E\}$

Wir definieren in einem Graph G = (N, E, U) einen Pfad P vom Knoten m zum Knoten n als Teilmenge der Kantenrelation E:

$P(m,n) = \{(m,x_1), (x_1,x_2), \ldots ,(x_i,n)\}$, wobei gelten muss $E' \supseteq P(m,n)$

Selbstverständlich können mehrere Pfade von einem Knoten zu einem anderen führen. Die Distanzfunktion D zwischen zwei Knoten ist definiert als der längste Pfad vom Startknoten zum Endknoten:

$D: N \times N \rightarrow Z^+$, sodass $D(m,n) = \max(\mid P(m,n) \mid)$

Die durch den Datenflussgraph definierte, partielle Ordnung $<^*$ muss für die Ausführung auf eine *Sequenz* von Instruktionen abgebildet werden. Diese Abbildung stellen wir als eineindeutige Funktion *S* dar, die jedem Knoten seinen Platz in der Sequenz zuweist:

$S: N \rightarrow Z^+$,
sodass $(n <^* m \Rightarrow S(n) < S(m)) \wedge (S(n) = S(m) \Rightarrow m = n) \wedge 0 < S(n) \leq |N|$

Der erste Teil der Bedingung garantiert, dass die partielle Ordnung des Datenflussgraphen in der Sequenz nicht verletzt wird. Der zweite Teil der Bedingung sorgt dafür, dass die Funktion eineindeutig wird und jedem Knoten eine eigene Sequenznummer zugeteilt wird. Im dritten Teil wird der Bereich der Sequenznummern aus technischen Gründen lückenlos definiert. Im Prinzip könnte man diese Sequenznummern direkt als Adressen in einem Programmspeicher auffassen.

Es ist klar, dass es im allgemeinen nicht nur eine einzige derartige Funktion gibt. Die Frage ist nun, welche Funktion S bzw. welche Sequentialisierungsstrategie eignet sich am besten für die ADAM-Architektur?

7.2.2. Erstes Optimierungskriterium: Maximale Anzahl von Zwischeninstruktionen

In normalen Compilern wird in der Regel die Strategie der *Tiefentraversierung* verwendet: Man beginnt bei den Eingängen des Umgebungsknotens und durchläuft immer über den tiefsten, noch nicht benutzten Eingang eines Knotens rekursiv den Graph, bis man entweder auf eine Konstante, einen bereits traversierten Knoten oder einen Grapheingang stösst. Sobald alle Eingänge eines Knotens markiert sind, also alle seine Operanden berechnet sind, fügt man ihn hinten in der Sequenz ein. Die Methode der Tiefentraversierung und andere heuristische Codegenerierungsmethoden für gerichtete, azyklische Graphen (DAGs) wurden schon eingehend analysiert [AhoJoh76, AhJoUl77, AhSeUl86]. Es ist auch bekannt, dass das Problem der optimalen Codegenerierung in Bezug auf solche Graphen auf Maschinen mit beschränkter Registerzahl NP-vollständig ist [BruSet76, AhJoUl77]. Um einen Vergleich mit besseren Methoden zu haben, wurde im MFL-Compiler die Tiefentraversierung als Option eingebaut.

Wie schon im einführenden Abschnitt 7.1 "Parallelität auf der ADAM-Architektur" erwähnt, sollen zwischen die zwei Phasen einer zweiphasigen Instruktion möglichst viele *Zwischeninstruktionen* plaziert werden. Wir definieren die Menge der *zweiphasigen Instruktionen (N_2)* als Untermenge aller Knoten im Graph:

$N \supseteq N_2$

Für die weiteren Betrachtungen brauchen wir neben den zweiphasigen Instruktionen auch noch die Synchronisationsinstruktionen. Mit *Synchronisationsinstruktionen* N_S sind jene Instruktionen gemeint, die ein Resultat einer zweiphasigen Instruktion erwarten:

$N \supseteq N_S$, sodass $n \in N_2 \wedge m \in N_S \wedge (n,m) \in E'$

Zu jeder Kante lässt sich für eine bestimmte Sequentialisierungsfunktion S die Anzahl Zwischeninstruktionen (*interleaving instructions*) als Differenz der Positionen in der Sequenz angeben:

$I: E \rightarrow Z^+$, sodass $I((n, m)) = S(m) - S(n)$

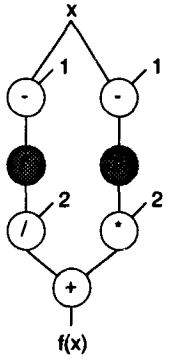
Es soll nun die Summe aller Zwischeninstruktionen zwischen einer zweiphasigen Instruktion und der jeweils ersten, zugehörigen Synchronisationsinstruktion maximiert werden:

$\Sigma_{(n \in N_2 \wedge (n,x) \in E)} \min(I((n,x))) = \max$

Dass die Tiefentraversierung im Vergleich zu einer Sequentialisierungsfunktion, die die Anzahl der Zwischeninstruktionen maximiert, extrem schlecht abschneiden kann, demonstriert das Beispiel in Figur 7.1 eindrücklich:

Rekursive Funktion: f(x) = **IF** x = 0 **THEN** 1 **ELSE** f(x-1) * 2 + f(x-1) / 2

Graph für Rekursion:



Code für Rekursion:

Tiefentraversierung:

```
r1 := x - 1
r1 := f(r1)
r1 := r1 / 2
r2 := x - 1
r2 := f(r2)
r2 := r2 * 2
f(x) := r1 + r2
```

Maximale Anzahl Zwischeninstruktionen:

```
r1 := x - 1
r1 := f(r1)
r2 := x - 1
r2 := f(r2)
r1 := r1 / 2
r2 := r2 * 2
f(x) := r1 + r2
```

Fig. 7.1: Unterschiedliche Sequentialisierungsfunktionen

Die Beispielfunktion wurde mit x = 14 in den beiden gezeigten Versionen auf dem ADAM-Simulator mit 256 Prozessoren simuliert. Während der nach der Tiefentraversierung sequentialisierte Code 340 * 10^4 Zyklen lief, lag das Ergebnis beim optimierten Code schon nach 1.7 * 10^4 Zyklen vor. Das experimentelle Resultat, eine um den Faktor 200 unterschiedliche Laufzeit, lässt den Schluss zu, dass die eine Version rein sequentiell ablief, während die andere die Maschine nahezu optimal auslastete. Analysiert man den unterschiedlichen Ablauf der Berechnung, so lässt sich dieser Unterschied leicht erklären. In der Version mit Tiefentraversierung wird jeder Ast des riesigen Aufrufbaums bis zu seinem Blatt durchgerechnet, bevor eine andere Berechnung gestartet wird. In der zweiten Version breitet sich die Parallelität explosionsartig aus: Auf jeder Stufe des Aufrufbaums verdoppelt sich die Anzahl der aktiven Codeblöcke. Im Sinne der Berechnungstheorie entsprechen die beiden Varianten den Berechnungsstrategien *leftmost-innermost* und *parallel-outermost* [EngLäu88].

7.2.3. Zweites Optimierungskriterium: Minimale Anzahl von Kontextwechseln

Mit dem zweiten Optimierungskriterium wird angestrebt, die Anzahl der Teilsequenzen, die ohne Unterbruch durchlaufen werden können, zu minimieren. Im folgenden werden wir diese Teilsequenzen *Blöcke* nennen. Betrachtet man eine Instruktionssequenz, so kann ein *Unterbruch* mit Kontextwechsel auf dem Prozessor immer nur vor einer Synchronisationsinstruktion auftreten.

Es wird im folgenden angenommen, dass ein Codeblock auf dem Prozessor erst dann wieder in die Liste der bereiten Prozesse eingefügt wird, wenn alle ausstehenden, durch zweiphasige Instruktionen ausgelösten Anforderungen erledigt sind. Dies ist zwar in der momentanen Architektur nicht der Fall; das Modell wird aber die Realität dennoch gut wiedergeben, da aufgeweckte Codeblöcke hinten an die Warteschlange der bereiten Codeblöcke angehängt werden und somit in der Regel genügend Zeit zur Verfügung steht, um alle ausstehenden Anforderungen abzuschliessen, bis der betreffende Codeblock wieder zum Laufen kommt.

Unter dieser Annahme lässt sich ein *Block* als maximale, lückenlose Teilsequenz definieren, sodass sich nie eine Synchronisationsinstruktion und ihre zugehörige zweiphasige Instruktion im gleichen Block befinden. Der erste Block in einer Sequenz geht also immer bis vor die erste Synchronisationsinstruktion. Die weiteren Blöcke umfassen jeweils alle Instruktionen bis hin zur ersten Synchronisationsinstruktion, die sich auf eine Instruktion im gleichen Block bezieht.

Formal können wir die Blockgrenzen dadurch definieren, dass wir für jeden Block, abhängig von der Sequenz, die Position der ersten Instruktion des Blocks in der Sequenz $B_i(S)$ angeben. Jeder Block umfasst dann alle Instruktionen an den Positionen von $B_i(S)$ bis und mit $B_{i+1}(S)$-1. Für den letzten Block, für den man kein $B_{i+1}(S)$ mehr angeben kann, ist die letzte Instruktion einfach die letzte Instruktion der Sequenz.

B_i: $(N \rightarrow Z^+) \rightarrow N$,

sodass	$B_0(S) = 1$ (Verankerung, erste Instruktion der Sequenz)
und	$B_i(S) = \min(S(m))$ so dass $n \in N_2 \wedge (n,m) \in E \wedge S(n) \geq B_{i-1}(S)$ (Induktion)

Zusammenfassend kann man sagen, dass die Anzahl der Blöcke die maximale Anzahl von Kontextwechseln während der Ausführung einer Sequenz bestimmt.

Interessant ist die Frage, ob man direkt aus der Struktur des Graphen herleiten kann, in wieviele Blöcke eine Sequenz zu diesem Graphen mindestens zerlegt werden muss. Tatsächlich ist es einfach, eine untere Grenze für die Anzahl Blöcke einer Sequenz aus der Struktur des Graphen herzuleiten.

Zuerst definieren wir die maximale Anzahl zweiphasiger Instruktionen (D_2) zwischen zwei Knoten, wobei der Startknoten mitzählt, der Endknoten aber nicht.

D_2: $N \times N \rightarrow Z^+$, sodass $D_2(m,n) = \max_{p=P(m,n)}(\mid \{(n,x) \mid n \in N_2 \wedge (n,x) \in p\} \mid)$

Man muss nun die maximale Anzahl zweiphasiger Instruktionen auf einem Pfad vom Eingang bis zum Ausgang bestimmen. Mit anderen Worten gesagt, wird die Länge des *kritschen Pfades* C_2 in G bezüglich zweiphasiger Instruktionen gemessen:

$C_2 = D_2(T,T)$

Für einen bestimmten Graphen G gilt nun, dass es keine Sequenz mit weniger als C_2 Unterbrüchen, beziehungsweise weniger als $C_2 + 1$ Blöcken gibt. Beweisen lässt sich diese Aussage, indem man eine Sequenz für den Graph, der nur aus den Knoten auf diesem maximalen Pfad besteht, bildet. Für jede zweiphasige Instruktion auf diesem Pfad muss sicher einmal synchronisiert werden, bevor die nächste ausgeführt werden kann.

7.2.4. Sequenz, die beide Kriterien optimiert?

Die nächste Frage ist, ob es eine Sequentialisierungsfunktion gibt, die für alle Graphen beide Optimierungskriterien (maximale Anzahl Zwischeninstruktionen wie auch minimale Anzahl Blöcke) erfüllt. Auf den ersten Blick sieht es so aus, als ob die beiden Optimierungskriterien eng miteinander verwandt sind. Viele Zwischeninstruktionen bedeuten lange Blöcke und umgekehrt. Bei einer näheren Betrachtung zeigt sich aber, dass es Graphen gibt, für die keine Sequenz existiert, die beide Kriterien erfüllt. Das Beispiel in Figur 7.2 zeigt die Widersprüchlichkeit der Anforderungen in speziellen Fällen.

Um die Anzahl Zwischeninstruktionen im Beispiel aus der Figur 7.2 zu maximieren, müssen die Instruktionen F und G unbedingt als Zwischeninstruktionen zu den zweiphasigen Instruktionen B und C verwendet werden. Damit dies möglich ist, muss aber zuerst bei F auf das Resultat von D synchronisiert werden, bevor die zweiphasige Instruktion E ausgeführt wird. Dies führt zu einem zusätzlichen

Unterbruch vor F. Sollte dieser vermieden werden, müssten D und E vor F und G ausgeführt werden, womit sie nicht mehr als Zwischeninstruktionen für B und C zur Verfügung stehen. Die Anforderungen für die beiden Optimierungskriterien widersprechen sich also in diesem Beispiel.

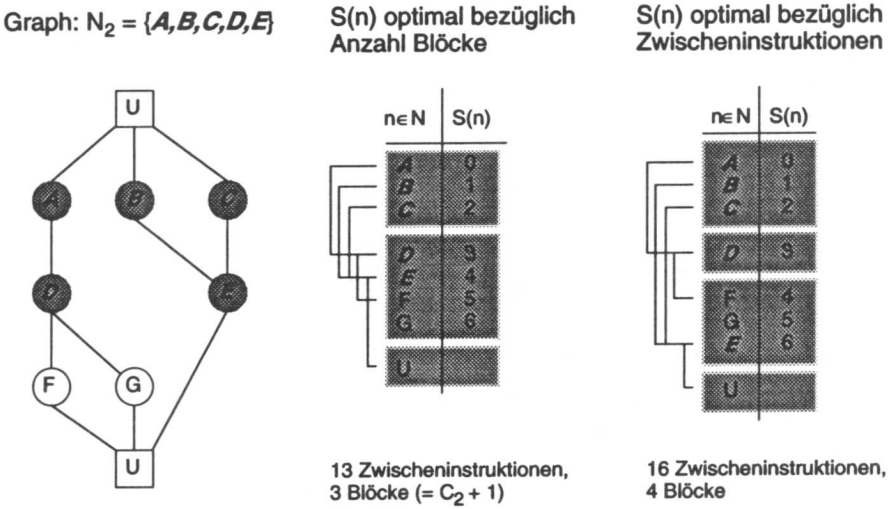


Fig. 7.2: Schwierig zu optimierende Sequenz

Aus diesen Ueberlegungen folgt, dass keine Sequentialisierungsfunktion existiert, die für alle möglichen Graphen beide Kriterien optimiert!

7.2.5. Kriterien für die bestmögliche Sequentialisierungsfunktion

Aus den vorangegangenen Ueberlegungen ist klar, dass eine Sequentialisierungsfunktion gesucht wird, die zwei ähnliche, sich aber in gewissen Fällen trotzdem widersprechende Optimierungskriterien erfüllt. Wir können also nicht mehr erwarten als eine Funktion, die im Sinne eines der beiden Kriterien optimal und für das andere zumindest eine gute Heuristik ist.

Es stellen sich nun zwei Fragen: Welches Kriterium ist wichtiger und für welches Kriterium ist eine optimale Lösung mit geringerer berechnungstheoretischer Komplexität zu erreichen?

Die Reduktion der Anzahl Blöcke bewirkt weniger Kontextwechsel bei der Programmausführung. Es wird also angestrebt, unproduktive Arbeit zu vermeiden. Eine maximale Anzahl von Zwischeninstruktionen ist erwünscht, um eine möglichst schnelle Ausbreitung von Parallelität auf der Maschine zu erreichen. Aus

verschiedenen Arbeiten [Maquel90, Ruggie87] geht hervor, dass im allgemeinen eher das Problem von zuviel anstatt zuwenig Parallelität besteht. Das Problem der Zwischeninstruktionen ist dementsprechend nicht so wichtig, da genügend Parallelität von anderen Codeblöcken auf dem gleichen Prozessor zur Verfügung steht, um die Latenz zu verstecken. Hingegen ist die Reduktion von Kontextwechseln von grosser Bedeutung. Dies gilt besonders, wenn man annimmt, dass die ADAM-Architektur nicht als spezielle Hardware, sondern als Software auf eine Gruppe von existierenden Prozessoren mit ihren höheren Umschaltezeiten realisiert wird.

Auch die zweite Frage lässt sich zu Gunsten der Reduktion von Kontextwechseln beantworten. Wie später gezeigt wird, gibt es effiziente Algorithmen, um für einen beliebigen Graphen eine Sequenz mit der theoretisch minimalen Anzahl von Blöcken zu generieren. Das Problem mit der minimalen Anzahl von Zwischeninstruktionen ist hingegen, analog zu anderen Optimierungsproblemen aus der Codegenerierung [BruSet76, AhJoUl77] oder der Betriebssystemtheorie [CofDen73, Coffma76], nur mit Hilfe einer Suche durch den Baum der verschiedenen Möglichkeiten und somit exponentiellem Aufwand zu lösen.

Eine *gute Sequentialisierungsfunktion* ist also optimal bezüglich dem Kriterium der Kontextwechsel und gleichzeitig eine gute Heuristik für das Kriterium der Zwischeninstruktionen. Im folgenden wird schrittweise eine Sequentialisierungsfunktion konstruiert, die beiden Kriterien genügt.

Zuerst werden alle zweiphasigen Instruktionen, entsprechend ihrem maximalen Abstand (gemessen in zweiphasigen Instruktionen (D_2) zum Ausgang des Graphen (den Eingängen des Umgebungsknotens)) in *Klassen* eingeteilt. Jeder Knoten gehört natürlich zu genau einer Klasse. Die Figur 7.3 zeigt die Klasseneinteilung für einen kleinen Beispielgraphen.

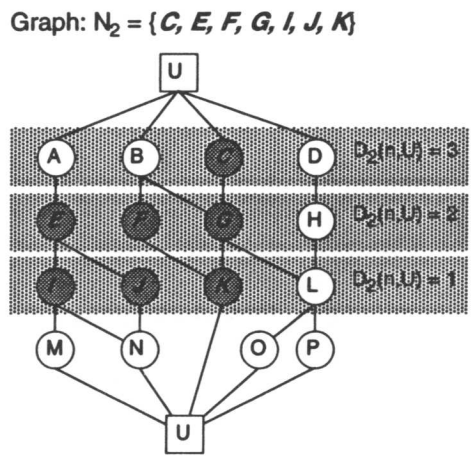


Fig. 7.3: Klassen von zweiphasigen Instruktionen

Alle zweiphasigen Instruktionen der gleichen Klasse müssen in der Sequenz vor der ersten Synchronisationsinstruktion auf einen Knoten dieser Klasse zu stehen kommen. Im Beispiel aus Figur 7.3 bedeutet dies konkret, dass I und J in der Sequenz auf keinen Fall vor G zu stehen kommen dürfen, obwohl dies rein aufgrund der Abhängigkeiten im Graph möglich wäre. G gehört nämlich zur Klasse 2, wogegen I und J auf E in der Klasse 2 synchronisieren. Formal lässt sich die Regel folgendermassen formulieren:

$$N_2 \supseteq \{m, n\} \wedge o \in N \wedge m \leq^* o \wedge D_2(m, U) = D_2(n, U) \Rightarrow S(n) < S(o)$$

Oben haben wir gezeigt, dass jede Sequenz im Minimum in C2 Blöcke zerfällt. Diese optimale Zahl kann nur erreicht werden, wenn pro Klasse nur einmal synchronisiert wird. Die oben definierte Regel garantiert genau diese Eigenschaft. Würde die Regel verletzt, so müsste unnötigerweise auf Knoten derselben Klasse mehr als einmal synchronisiert werden.

7.2.6. Eine bestmögliche Sequentialisierungsfunktion

Nun sind alle Voraussetzungen gegeben, um die tatsächlich verwendete Sequentialisierungsfunktion bzw. den verwendeten Sequentialisierungsalgorithmus zu formulieren und dessen Eigenschaften zu untersuchen.

Der Algorithmus funktioniert nach dem folgenden einfachen Prinzip: Wähle einen Knoten aus der Menge aller Knoten aus, deren Vorgänger bereits eine tiefere Position in der Sequenz zugeteilt bekommen haben, vergebe die aktuelle Sequenz-

nummer an ihn und erhöhe die aktuelle Sequenznummer. Beginne mit der Sequenznummer 0 und wiederhole den Iterationsschritt für alle Knoten im Graphen.

```
aktuelleSequenzNummer := 0;
WHILE aktuelleSequenzNummer < |N| DO
   Berechne Menge der Kandidaten K(aktuelleSequenzNummer);
   Wähle einen Kandidaten aus;
   Vergib aktuelleSequenzNummer an den gewählten Knoten;
   INC(aktuelleSequenzNummer);
END;
```

Die Menge aller Kandidaten K(i) für eine bestimmte Position in der Sequenz berechnet sich einfach:

$K: Z^+ \rightarrow \mathcal{P}(N)$, sodass $K(i) = \{n \in N \mid \forall m \leq^* n: \; S(m) < i \wedge \neg(S(n) < i)\}$

Der erste Term in der Definition von K garantiert, dass alle Vorgänger von n berechnet sind und der zweite Term sorgt dafür, dass Knoten, an die schon eine Sequenznummer vergeben wurde, nicht wieder als Kandidaten auftauchen. Die Definition wird gebraucht, um die Sequenzfunktion S zu definieren, verwendet S aber auch selbst. S wird also rekursiv definiert. Betrachtet man die Definition von K genauer, so sieht man, dass nur kleinere Sequenznummern als i gebraucht werden. Da die Sequenznummern im Algorithmus von 0 an in aufsteigender Reihenfolge vergeben werden, gibt es keine Probleme mit dieser Rekursion.

Wird im gezeigten Algorithmus der Kandidat zufällig aus der Kandidatenmenge ausgewählt, so ist durch die gewählte Definition von K sichergestellt, dass die Grunddefinition von S aus Abschnitt 7.2.1 "Graph und Sequenz" erfüllt ist.

Wie lässt sich die Auswahl der Kandidaten verbessern, sodass aus diesem Algorithmus eine im obenerwähnten Sinne *gute Sequentialisierungsfunktion* wird? Der Kandidat n wird aus der Kandidatenmenge nach folgenden Prioritäten ausgewählt:

Priorität:	Bedingung:	Beschreibung:
1	$n \in N_2 \wedge \neg(n \in N_S)$	Zweiphasige Instruktionen, die nicht zu einer Synchronisation führen
2	$\neg(n \in N_S)$	Andere Instruktion, die nicht zu einer Synchronisation führen, potentielle Zwischeninstruktionen
3	$n \in N_S$	Synchronisationsinstruktionen

Tabelle 7.1 Prioritätsklassen im Sequentialisierungsalgorithmus

Diese Prioritätensetzung versucht heuristisch, zwischen zweiphasigen Instruktionen und ihren Synchronisationsinstruktionen möglichst viele Zwischeninstruktionen einzusetzen. Selbstverständlich kann es in jeder Prioritätsklasse mehrere Knoten zur Auswahl geben. Für diesen Fall gibt es zwei Kriterien:

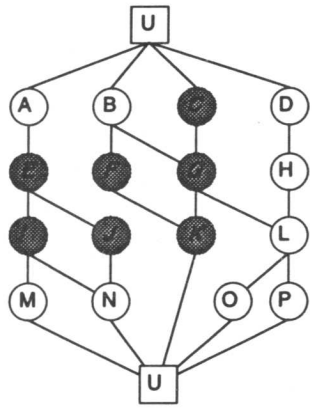
1. Es wird der Knoten mit maximalem Abstand $D_2(n, U)$ ausgewählt. Dieses Kriterium garantiert, zusammen mit dem ersten die Regel einzuhalten, dass alle zweiphasigen Instruktionen der gleichen D_2-Klasse (vgl. oben) vor der ersten Synchronisationsinstruktion für diese Klasse positioniert werden und somit eine minimale Anzahl von Blöcken resultiert.

2. Falls auf Grund des ersten Kriteriums kein Entschied getroffen werden kann, wird jener Knoten ausgewählt, der am meisten Nachfolger (auch indirekte) hat. Dabei hofft man, dass Knoten mit vielen Nachfolgern die Kandidatenmenge am meisten vergrössern und somit zusätzliche Möglichkeiten zur Plazierung von Zwischeninstruktionen schaffen, wobei in der letzten Prioritätsklasse (3) die Nachfolger für zweiphasige Instruktionen getrennt von den anderen gezählt werden. Falls zweiphasige Synchronisationsinstruktionen zur Verfügung stehen, so wird unter ihnen jene mit der höchsten Anzahl Nachfolger ausgewählt. Sonst wird jene der übrigen Instruktionen mit den meisten Nachfolgern selektiert.

Kann nach Anwendung aller Kriterien noch immer nicht entschieden werden, welcher Knoten ausgewählt werden soll, entscheidet der Zufall.

Die Figur 7.4 zeigt den Ablauf des Algorithmus für ein Beispiel. Für jeden Schritt ist die Kandidatenmenge vor der Auswahl und das angewandte Kriterium bei der Auswahl dokumentiert. Man sieht, dass tatsächlich die minimale Anzahl von Blöcken entsteht und dass die Zwischeninstruktionen nach einer guten Heuristik verteilt werden. Jedenfalls ist es auch für diesen kleinen Graphen nicht einfach, von Hand eine Sequenz zu finden, die mehr Zwischeninstruktionen enthält.

Graph: N_2 = {*C, E, F, G, I, J, K*}



S(n)	n	K(S(n))	Kommentar
0	C	{A,B,C,D}	Einziger Knoten mit PK 1
1	B	{A,B,D}	B hat mehr Nachfolger als A bei gleichem D_2 in PK 2
2	F	{A,D,F,G}	Einziger Knoten mit PK 1
3	A	{A,D,G}	Maximales D_2 in PK 2
4	E	{E,D,G}	Einziger Knoten mit PK 1
5	D	{D,G,I,J}	Einziger Knoten mit PK 2
6	H	{G,H,I,J}	Einziger Knoten mit PK 2
7	G	{G,I,J}	Maximales D_2 in PK 3
8	L	{I,J,K,L}	Einziger Knoten mit PK 2
9	O	{I,J,K,O,P}	Zufällige Auswahl in PK 2
10	P	{I,J,K,P}	Einziger Knoten mit PK 2
11	I	{I,J,K}	I hat mehr Nachfolger als J bei gleichem D_2 in PK 3
12	M	{J,K,M}	Einziger Knoten mit PK 2
13	J	{J,K}	J hat mehr Nachfolger als K bei gleichem D_2 in PK 2
14	K	{K}	Einziger Kandidat

PK = Prioritätsklasse
40 Zwischeninstruktionen,
4 Blöcke mit Blockstarts bei C, G, K und U am Schluss (Synch. zu K, nicht in Sequenz)

Fig. 7.4: Beispiel für den Sequentialisierungsalgorithmus

Der Algorithmus ist ein gutes Beispiel für die Anwendung einer allgemein verwendbaren Technik zur schnellen und näherungsweisen Lösung von Optimierungsproblemen. Die Aufteilung der Instruktionen in die drei Prioritätsklassen und die Bevorzugung von Knoten mit mehr Nachfolgern innerhalb der gleichen Prioritätsklasse ist eine typische Form einer gierigen Heuristik (*Greedy Algorithm* [AhHoUl83]). Durch einfache zusätzliche Bedingungen wurde sogar optimales Verhalten für das eine Optimierungskriterium erreicht.

7.3. Codegenerierung

7.3.1. Registerzuteilung

In der jetzigen Version des MFL-Compilers werden alle Zwischenresultate bei der Berechnung eines Graphen in Frame-Registern (vgl. 2.4. "Das Register-Frame: Synchronisation über Daten") zwischengespeichert. Dafür muss jedem Knotenausgang ein Register für das Resultat der entsprechenden Operation zugeordnet werden.

Zu diesem Zweck wird für jedes Register im aktuell zu generierenden Codeblock ein *Referenzzähler* geführt, welcher festhält, wieviele Operationen dieses noch als Operanden brauchen. Am Anfang werden die Referenzzähler für alle Register auf

Null zurückgesetzt. Wird der Code für einen Knoten generiert, so wird ihm für das Resultat ein Register ohne Referenzen zugewiesen. Der Referenzzähler dieses Registers wird nun auf die Anzahl der mit diesem Ausgang verbundenen Eingänge gesetzt. Wird bei der Generierung der Folgeinstruktionen das betreffende Register als Operand benutzt, dekrementiert der Codegenerator den entsprechenden Referenzzähler. Sobald alle Operanden gebraucht worden sind, die sich auf diesen Ausgang bezogen haben, ist das Register wieder frei und kann einem neuen Ausgang zugeordnet werden.

Für grössere Graphen ist die beschriebene Methode der Registerzuteilung mit einer begrenzten Zahl von vorhandenen Registern untauglich. Das Problem verschärft sich zusätzlich dadurch, dass der vorgeschlagene Sequentialisierungsalgorithmus mehr Register verbraucht als ein konventioneller.

In einer späteren Implementation von MFL müsste deshalb unbedingt die Möglichkeit vorgesehen werden, lokale Werte nicht nur in Registern, sondern auch im Hauptspeicher abzuspeichern, falls keine weiteren Register mehr frei sind. Man nennt diesen Vorgang "*Register Spilling*" und entsprechende Methoden sind in der Fachliteratur bekannt [Chaitin82]. Reicht die Anzahl Register nicht für den Code eines bestimmten Codeblocks, muss am Anfang ein genügend grosses, lokales Objekt alloziert werden, welches als zusätzlicher Speicher für lokale Daten dient. Ueber LOAD- und STORE-Operationen werden dann die Register auf dieses Objekt ausgelagert, beziehungsweise von diesem zurückgeholt.

Allerdings können die bekannten Register-Austausch-Algorithmen nicht unbesehen bei Codegeneratoren für die ADAM-Architektur übernommen werden, da die Register eine doppelte Rolle spielen. Einerseits dienen sie dem Zwischenspeichern lokaler Daten und andererseits dienen sie der Synchronisation paralleler Tätigkeiten über Daten. Die minimale Anzahl Speicher-Register-Transfers ist somit nicht das einzige Kriterium, welches bei der Auswahl der auszulagernden Register von Bedeutung ist. Ebenso muss berücksichtigt werden, welche Register tatsächlich der Synchronisation zweiphasiger Instruktionen und welche nur dem einfachen Zwischenspeichern von lokalen Daten dienen. Das frühzeitige Auslagern von Synchronisationsregistern zerstört unter Umständen Parallelität.

Die Erforschung eines für die ADAM-Architektur geeigneten Registeraustauschalgorithmus wäre ein möglicher Gegenstand weitergehender Forschung.

7.3.2. Zusammengesetzte Knoten

Zusammengesetzte Knoten werden wie alle anderen Knoten in der Sequenz plaziert. Ihre Subgraphen werden ebenfalls nach der beschriebenen Methode sequentialisiert. Der Code für einen zusammengesetzten Knoten besteht aus den durch seine Subgraphen definierten Instruktionssequenzen und eingestreuten Instruktionen zur Kontrollflusssteuerung. Der Code zur Kontrollflusssteuerung für die verschiedenen Arten von zusammengesetzten Knoten wird analog zu konventionellen Compilern generiert [AhSeUl86]. Einzig die Codegenerierung für FORALL-Knoten ist etwas aussergewöhnlich. Sie wird im Abschnitt 5.4.1 "Uebersetzung von FORALL-Schleifen" beschrieben.

7.4. Codegenerierung an einem Beispiel: Livermore-Loop 7

In diesem Kapitel wird der durch den neuen Algorithmus generierte Code mit dem entsprechenden Code, der durch Tiefentraversierung entstehen würde, verglichen. Als Beispiel wurde der Livermore-Loop 7 [McMaho86] gewählt, ein häufig zum Zweck des Leistungsvergleichs zwischen verschiedenen Maschinen verwendetes Beispiel. Die MFL-Implementation von Loop7 findet man im Anhang B.

Der MFL-Compiler generiert ohne Elimination gemeinsamer Unterausdrücke den Graph in Figur 7.5 für den Schleifenkörper dieses Programms. Die grauen Kästchen heben auch hier die zweiphasigen Instruktionen, in diesem Fall alles Objekt-Manager-Befehle, hervor. Die kursiv, fett gedruckten Zahlen in der linken, oberen Ecke entsprechen der Position des Knotens in der optimierten Sequenz (vgl. auch Fig. 7.6). Die Zahlen rechts unten zeigen die Position eines Knotens in einer durch Tiefentraversierung (leftmost depth first) entstandenen Sequenz (vgl. auch Fig 7.7).

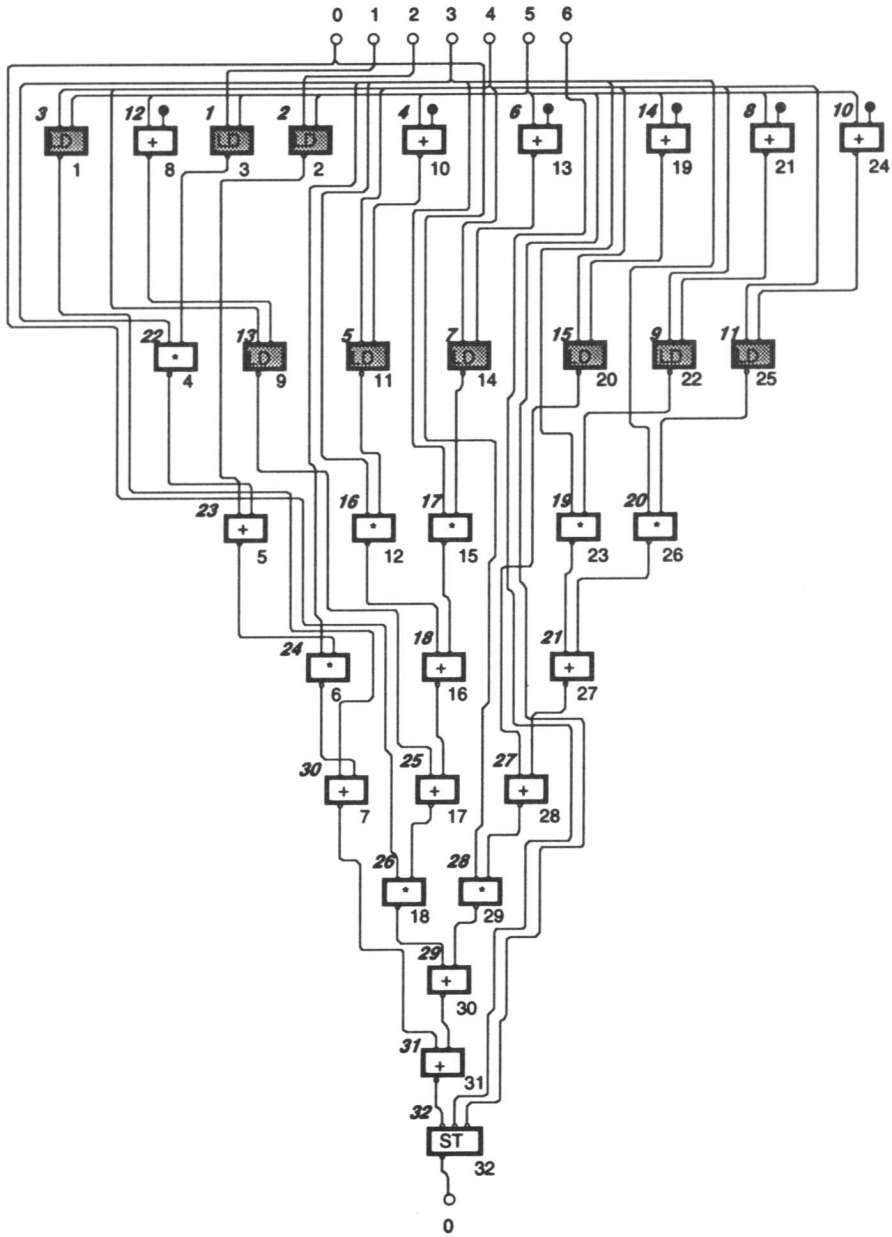


Figure 7.5: Graph und Sequentialisierung für den Livermore-Loop Nr. 7

Aus diesem Graph wird durch eine einfache Tiefentraversierung (left-most-depth-first) der in Figur 7.6 gezeigte Code generiert. Es wird dabei angenommen, dass

die Ausgänge des Umgebungsknotens bereits in den Registern 0 bis 6 vorhanden sind, und dass das Resultat am Schluss in das Register 0 zu stehen kommt.

```
[1] r7 := LD1(r4, r5);
[2] r2 := LD1(r2, r5);
[3] r1 := LD1(r1, r5);
[4] r1 := r3 * r1;
[5] r1 := r2 + r1;
[6] r1 := r3 * r1;
[7] r1 := r7 + r1;
[8] r2 := r5 + c;
[9] r2 := LD1(r4, r2);
[10] r7 := r5 + c;
[11] r7 := LD1(r4, r7);
[12] r7 := r3 * r7;
[13] r8 := r5 + c;
[14] r8 := LD1(r4, r8);
[15] r8 := r3 * r8;
[16] r7 := r7 + r8;
[17] r2 := r2 + r7;
[18] r2 := r0 * r2;
[19] r7 := r5 + c;
[20] r7 := LD1(r4, r7);
[21] r8 := r5 + c;
[22] r8 := LD1(r4, r8);
[23] r8 := r3 * r8;
[24] r9 := r5 + c;
[25] r4 := LD1(r4, r9);
[26] r3 := r3 * r4;
[27] r3 := r8 + r3;
[28] r3 := r7 + r3;
[29] r0 := r0 * r3;
[30] r0 := r2 + r0;
[31] r0 := r1 + r0;
[32] r0 := ST1(r0, r6, r5);
```

Fig. 7.6: Einfache Codegeneration

Unsere Sequentialisierungsfunktion generiert für den gleichen Graphen den in der Figur 7.7 gezeigten Code.

```
[1] r1 := LD1(r1, r5);
[2] r2 := LD1(r2, r5);
[3] r7 := LD1(r4, r5);
[4] r8 := r5 + c;
[5] r8 := LD1(r4, r8);
[6] r9 := r5 + c;
[7] r9 := LD1(r4, r9);
[8] r10 := r10 + c;
[9] r10 := LD1(r4, r10);
[10] r11 := r5 + c;
[11] r11 := LD1(r4, r11);
[12] r12 := r5 + c;
[13] r12 := LD1(r4, r12);
[14] r13 := r5 + c;
[15] r4 := LD1(r4, r5);
[16] r8 := r3 * r8;
[17] r9 := r3 * r9;
[18] r8 := r8 + r9;
[19] r9 := r3 * r10;
[20] r10 := r3 * r11;
[21] r9 := r9 + r10;
[22] r1 := r3 * r1;
[23] r1 := r2 + r1;
[24] r1 := r3 * r1;
[25] r2 := r12 + r8;
[26] r2 := r0 * r2;
[27] r3 := r4 + r9;
[28] r0 := r0 * r3;
[29] r0 := r2 + r0;
[30] r1 := r7 + r1;
[31] r0 := r1 + r0;
[32] r0 := ST1(r0, r6, r5);
```

Fig. 7.7: Durch den beschriebenen Algorithmus generierter Code

Beide Sequenzen brauchen die gleiche, minimale Anzahl von Instruktionen. Für die einfache Codegenerierung beträgt die Summe der Zwischeninstruktionen 30, für die optimierte 133. Der optimierte Code zerfällt in die minimale Anzahl von zwei Blöcken, wobei der zweite Block bei der Instruktion [16] beginnt. Der einfach generierte Code zerfällt in fünf Blöcke mit Blockanfängen bei [1], [12], [15], [24] und [26]. Pro Iteration ist beim optimierten Code mit einem Kontextwechsel,

im Gegensatz zu deren vier beim einfachen Code, zu rechnen. Für diese Verbesserung wird auch ein Preis bezahlt: Der optimierte Code braucht 13 anstelle von 10 Registern. Die Experimente im nächsten Abschnitt zeigen, dass sich diese Codeoptimierung bei der Laufzeit deutlich auszahlt.

7.5. Experimente mit dem ADAM-Simulator

Im folgenden Experiment wurde eine Reihe von MFL-Programmen mit konventioneller (Tiefentraversierung) und mit neuer Sequentialisierungsstrategie übersetzt. In der Tabelle 7.2 sind als Resultate des Experiments der Name des Programms, die zur Ausführung benutzte Anzahl Prozessoren, die Laufzeiten in Zyklen, die totale Anzahl Kontextwechsel für beide Varianten und die prozentualen Unterschiede zwischen den Varianten dokumentiert. Man beachte, dass es sich in der Spalte Zyklen um die Laufzeit des entsprechenden Programmes auf einer ADAM-Maschine mit der angegebenen Anzahl Prozessoren handelt. In der Spalte Kontextwechsel ist hingegen die gesamte Anzahl Kontextwechsel auf allen Prozessoren gemeint. So lässt sich erklären, dass bei gewissen Programmen, die auf vielen Prozessoren laufen, die Anzahl der Kontextwechsel höher liegt als die Anzahl der Zyklen. Ein Kontextwechsel dauert pro Prozessor schon mehrere Zyklen. Es können aber auch mehrere Prozessoren gleichzeitig den Kontext wechseln.

Programm	Anzahl	DFS		BFS		Verbesserung in %	
	Prozes-soren	Kontext-wechsel	Zyklen	Kontext-wechsel	Zyklen	Kontext-wechsel	Zyklen
BinInt	32	4121	20521	3159	17182	23.34	16.27
FFT	64	13913	98843	8896	85980	36.06	13.01
Fibonacci	16	34023	109196	34023	109196	0.00	0.00
Gauss	64	769464	2322419	738788	2195454	3.99	5.47
Loop1	32	3078	21171	2274	20677	26.12	2.33
Loop7	32	5965	32540	2270	24947	61.94	23.33
Loop7B	64	102633	187224	26234	147399	74.44	21.27
Mandelbrot	256	128063	791283	128234	743554	-0.13	6.03
MatMul	256	331537	158492	331537	158492	0.00	0.00
MergeSort	8	1810	799233	6408	317151	-254.03	60.32
ParMerge	32	6492	121874	6596	116534	-1.60	4.38

Tabelle 7.2: Konventionelle und neue Codegenerierung

Mit der neuen Sequentialisierungsstrategie sind alle Programme mindestens so schnell wie mit der alten, obwohl sich die Zahl der Kontextwechsel nicht in jedem Fall senkt, sondern in zwei Fällen ("MergeSort" und "ParMerge") sogar erhöht. Dies lässt sich einfach dadurch erklären, dass nicht jeder Kontextwechsel, der möglich ist, auch tatsächlich eintritt. Bei "MergeSort" ist der zu sortierende Array ein lokales Objekt auf dem Prozessor 0. Da im Falle von "MergeSort" genau das in Figur 7.1 beschriebene Problem eintritt und eine konventionelle Sequentiali-sierung sämtliche Parallelität dieses rekursiven Algorithmus zerstört, ist es nur logisch, dass nur wenige Kontextwechsel auftreten. Alle Speicherzugriffe beziehen sich nämlich auf lokale Objekte, so dass keine Latenz zu verstecken ist. Die mangelnde Parallelität zeigt sich dann aber deutlich beim Resultat für die Laufzeit. Bei "ParMerge" verschlechtert sich zwar die Anzahl der Kontextwechsel, das Programm wird aber dennoch schneller. Dies liegt an der besseren Plazierung der Zwischeninstruktionen.

Im übrigen sieht man, dass sich die Anzahl der Kontextwechsel teilweise drastisch verringert. Die Unterschiede wirken sich auf der ADAM-Architektur nicht so

drastisch auf die Laufzeiten aus. Man kann sich aber leicht vorstellen, dass die Bedeutung der neuen Sequentialisierung auf einer weniger geeigneten Architektur noch viel grösser wäre.

In einigen Fällen wirkt sich der Unterschied bei der Sequentialisierung gar nicht aus, da beide Strategien mangels Wahlmöglichkeiten zur selben Sequenz führen. Generell kann man sagen, dass sich die neue Art der Sequentialisierung bei jenen Programmen am meisten auszahlt, die aus den komplexesten Ausdrücken bestehen. Je komplexer die Ausdrücke sind, desto grösser und breiter werden die Graphen und desto mehr Wahlmöglichkeiten bestehen bei der Sequentialisierung.

7.6. Arbeiten mit ähnlichen Ansätzen

7.6.1. Codegenerierung aus gerichteten, azyklischen Graphen

Codegenerierung aus Bäumen, oder allgemeiner aus gerichteten, azyklischen Graphen ist ein altes Thema in der Informatikforschung. In [AhSeUl86] wird der derzeitige Stand der Technik dokumentiert. In [BruSet76], [AhoJoh76], [AhJoUl77] und anderen Publikationen werden optimale Algorithmen unter verschiedenen Rahmenbedingungen gezeigt. Unser Algorithmus fügt sich in diese Tradition ein und optimiert ein weiteres Kriterium. Gerade für moderne *RISC-Prozessoren* mit ihrer internen Parallelität (Pipelines, mehrere funktionale Einheiten) dürften in Zukunft ähnliche Codegenerierungsmethoden eine grosse Rolle spielen.

Es gibt schon eine Reihe von Untersuchungen, die sich mit der Wahl einer geeigneten Codesequenz für Prozessoren mit Pipelines und mehrfachen funktionalen Einheiten befassen. Dabei gibt es sogar Arbeiten, die sich auf die Sequentialisierungsmethode des Compilers verlassen, damit trotz Pipeline kein Register gelesen wird, bevor es geschrieben ist [Hennes83]. In diesem Ansatz muss zwischen dem Schreiben und dem Lesen des gleichen Registers eine der Länge der Pipeline entsprechende Anzahl von zusätzlichen Instruktionen eingesetzt werden. Falls möglich, werden sinnvolle Instruktionen eingesetzt, falls nicht, muss künstlich mit NOPs (No operation) aufgefüllt werden. Der Vorteil dieses Verfahrens ist, dass die Hardware viel einfacher wird. In heutigen RISC-Prozessoren [Motoro88] unterbricht die Hardware einfach die Pipeline, falls ein Operand noch nicht geschrieben ist (Scoreboarding). Von besonderer Bedeutung sind ähnliche Codegenerierungsstrategien, wenn damit längere grössere Speicheroperationen, insbesonde-

re das Verschieben von *Vektorregistern* von oder zum Speicher überdeckt werden können.

In [GooHsu88] wird die Problematik des Widerspruchs der Optimierungsziele "Maximale Parallelität" und "Minimale Anzahl Register" ausführlich diskutiert. Es wird dort auch auf den Unterschied zwischen Registervergabe vor und nach der endgültigen Sequentialisierung des Codes eingegangen. Die meisten Compiler versuchen, den Code erst im letzten Durchgang, nach der *Zuordnung der Register* zu reorganisieren [Hennes83, GibMuc86]. Dies hat den Nachteil, dass durch die Registervergabe zusätzliche Zuweisungsabhängigkeiten (vgl. Kap. 4 "Datenflussgraphen als Zwischencode für MFL") eingeführt wurden, die dann die Freiheit der Codereorganisation in der letzten Phase beeinträchtigen. Solange man viele Register zur Verfügung hat (was heute bei vielen Prozessoren der Fall ist) gibt es keinen Grund, Register zu sparen. In unserem Compiler wird deshalb die Registervergabe gleichzeitig mit der Sequentialisierung gemacht.

Im Unterschied zur ADAM-Architektur ist in allen oben beschriebenen Ansätzen die zu versteckende Latenz a priori bekannt. Im Falle der Pipeline-Latenz bei RISC-Prozessoren ist sie zudem sehr klein, so dass eine Reorganisation des Codes im letzten Durchgang vollauf genügt. [Krishna90] erzielt mit den oben beschriebenen Methoden auf bereits optimiertem SPARC-Code noch Verbesserungen von 18-25 Prozent. Sorgfältige Wahl der Codesequenz ist also auch für bereits existierende, moderne Mikroprozessoren von herausragender Bedeutung.

7.6.2. Fork-Join-Parallelität

Ein ganz ähnliches Problem wie der gezeigte Algorithmus wird in [Sarkar90] gelöst. Obwohl die dort gezeigte Lösung Teil eines optimierenden SISAL-Compilers ist, wird ein unterschiedliches Rechenmodell verwendet. Es wird angenommen, dass zwar FORK-Operationen für einzelne Prozesse, aber keine selektiven JOIN-Operationen zur Verfügung stehen. Mit anderen Worten gesagt, kann bei einem JOIN nicht auf einen bestimmten Prozess, sondern nur auf alle im Moment vom jeweiligen Vater-Prozess gestarteten Prozesse gemeinsam gewartet werden. Dieses Modell schränkt die Parallelität gegenüber dem ADAM-Modell zusätzlich ein und verlangt dementsprechend unterschiedliche Optimierungskriterien. Zudem wird angenommen, dass von allen Prozessen die Laufzeit bekannt ist, damit diese optimiert werden kann. Die Netto-Laufzeit wäre zwar annäherungsweise auch in unserem Compiler bekannt (vgl. Kap. 5 "Partitionierung von Datenflussgraphen in Codeblöcke"), sagt aber bei guter Maschinenauslastung nicht viel darüber aus, wie

lange ein Codeblock tatsächlich für seine Ausführung braucht. Auch Sarkar kommt zum Schluss, dass eine wirklich optimale Sequentialisierungsfunktion im Vergleich zum Ertrag mit unsinnig hohem Aufwand verbunden ist und schlägt ebenfalls eine heuristische Lösung vor. Bei seinen Experimenten stellt er den grössten Effekt bei der doppelt rekursiven Fibonacci-Funktion fest, die dem in Figur 7.1 gezeigten Beispiel sehr ähnlich ist.

7.6.3. List Scheduling

Aus der Betriebssystemtheorie ist ein ähnliches Problem schon seit langer Zeit bekannt. Beim sogenannten *List Scheduling* werden Prozesse, deren partielle Ordnung gegeben ist, in einer Liste angeordnet. Sie werden dann so von einer Anzahl Prozessoren ausgeführt, dass jeder Prozessor, sobald er mit der Bearbeitung eines Prozesses fertig ist, den in der Liste vordersten, noch nicht bearbeiteten Prozess holt und ausführt.

Es gibt eine Reihe von Algorithmen, die bezüglich Ausführungszeit optimale Listen unter allen denkbaren Rahmenbedingungen generieren. All diesen Algorithmen ist gemeinsam, dass sie mit riesigem Aufwand verbunden sind. In [Coffma76], [CofDen73], [RaChGo72] und vielen anderen Publikationen sind die entsprechenden Erkenntnisse veröffentlicht.

Interessant ist in diesem Zusammenhang vor allem eine Arbeit, die optimale Lösungen für das List- Scheduling-Problem mit einer Reihe von Heuristiken vergleicht und eine beste Heuristik findet [AdChDi74].

Vergleichen wir unser Problem mit List-Scheduling, so wird klar, dass das Unterproblem der Sequentialisierung der zweiphasigen Knoten ähnlich dem List-Scheduling-Problem ist, da jede zweiphasige Instruktion einen Prozess auf einem Prozessor startet. Obwohl bei einem Objekt-Manager-Befehl der Zielprozessor durch Objektreferenz und Index fest bestimmt ist, kann man in einer ersten Näherung annehmen, dass bei der Abarbeitung der Codesequenz jede zweiphasige Instruktion einen Prozess auf dem nächsten freien Prozessor auslöst. Mit zu berücksichtigen sind aber auch die synchronen Instruktionen, die immer auf dem lokalen Prozessor abgearbeitet werden.

Die im Algorithmus verwendete Heuristik, dass die Knoten mit grösstem D_2 priorisiert werden, entspricht genau der in [AdChDi74] als beste hervorgegangenen Heuristik "*High Levels First*" für zweiphasige Instruktionen.

Leer - Vide - Empty

8. Implementation des MFL-Compilers

In diesem Kapitel soll die praktische Umsetzung der in den vorangegangenen Kapiteln gezeigten Konzepte kurz beschrieben werden. Zur Implementation des MFL-Compilers wurde als Programmiersprache eine Version von Modula-2 mit objekt-orientierten Erweiterungen [P189] im "MacIntosh Programmer's Workshop" [Apple89a] verwendet. Die gesamte Applikation wurde auf dem objekt-orientierten Rahmen MacApp® [Apple89b] für den MacIntosh-Computer realisiert.

Im ersten Teil wird gezeigt, wie die verschiedenen Phasen des Codegenerators (vgl. Abschnitt 1.5 "Ueberblick über den Inhalt") auf Module aufgeteilt wurden und wie diese miteinander verknüpft sind. Die Funktion jedes Moduls wird kurz beschrieben.

Der zweite Teil befasst sich damit, wie die MFL-Codegenerierung in die Klassenstruktur des inkrementellen Compilers eingebettet wurde.

Im dritten Teil soll schliesslich ein Eindruck vermittelt werden, wie sich der MFL-Compiler dem Benutzer präsentiert.

8.1. Modulstruktur

In Figur 8.1 ist der Modulgraph mit den wesentlichsten 12 Modulen des MFL-Codegenerators dargestellt. Die gesamte MFL/FOOL-Programmierumgebung besteht aus insgesamt 49 Modulen mit ca. 57000 Programmzeilen in nahezu 200 Klassen. Neben den in Figur 8.1 gezeigten und weiter unten kurz beschriebenen Modulen, die nur der Codegenerierung dienen, sind auch in einer Reihe von anderen Modulen gewisse Funktionen der Codegenerierung enthalten (z. B. Benutzerschnittstelle).

In der Figur 8.1 sind die Import-Relationen zwischen den Modulen dargestellt. Ein Pfeil bedeutet, dass das Modul am Ende des Pfeils vom Modul an der Spitze des Pfeils importiert. Es fällt auf, dass der Modulgraph an zwei Stellen Zyklen enthält (UCodeGenDocument <-> Codegenerator und FunctionExpansion <-> CostAssignment). Diese Zyklen sind kein Zeichen schlechten Designs, sondern wurden absichtlich eingeführt, wie weiter unten noch gezeigt wird.

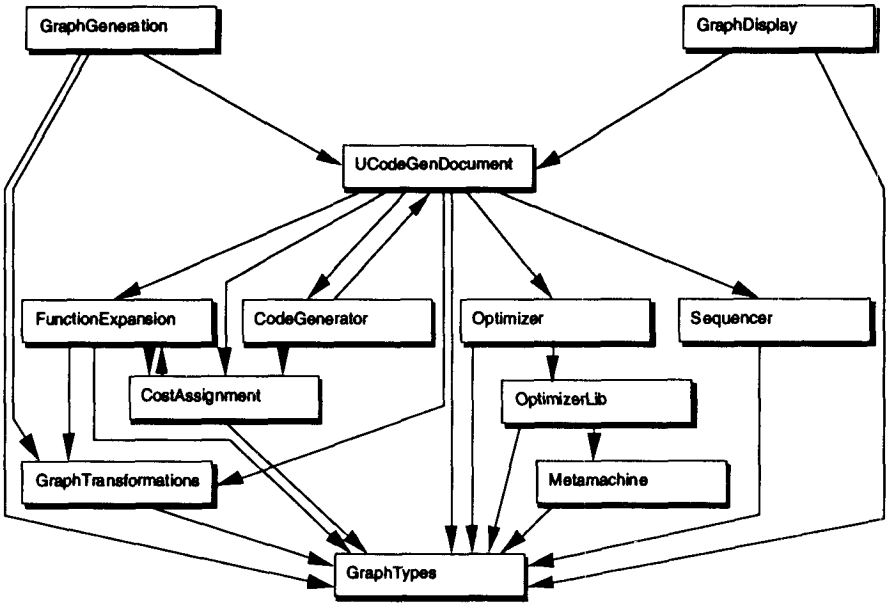


Fig. 8.1: Modulstruktur des Codegenerierungsteils im MFL-Compiler

Im folgenden soll jedes Modul kurz beschrieben werden:

Basis des ganzen Codegenerators ist das Modul *GraphTypes*, welches die im Abschnitt 4.6. "Datenstrukturen zur Repräsentation der Graphen" beschriebenen Typen, sowie eine Reihe von gemeinsamen Hilfsprozeduren zum Aufbau und zur Verwaltung von Graphen enthält.

Am Anfang der Codegenerierung steht die Generierung der hierarchischen Datenflussgraphen aus den syntaktischen Strukturen, welche das Compiler-Front-End liefert. Das Modul *GraphGeneration* enthält einen Satz rekursiver Prozeduren zur Analyse der wesentlichen, syntaktischen Strukturen und zur Synthese eines entsprechenden Datenflussgraphen.

Nicht direkt Teil der Codegenerierung, aber dennoch ein wichtiger Teil der Programmierumgebung ist das Modul *GraphDisplay*. Es enthält eine von der MacApp-Standardklasse "TView" abgeleitete View-Klasse zur interaktiven, graphischen Darstellung der erzeugten Datenflussgraphen. Neben der eigentlichen View-Klasse sind dort auch eine Reihe von Maus- und Menu-Kommandos implementiert, die es dem Benutzer erlauben, sich interaktiv durch die Graph-Hierarchie zu bewegen. Das Modul baut in seinem Innern eine eigene, zur graphischen Darstellung geeignete Datenstruktur zur Repräsentation der Graphen auf. Ein interessantes Problem, dessen Ab-

handlung aber den Rahmen dieser Dissertation sprengen würde, ist die Frage, wie sich Datenflussgraphen effizient und lesbar graphisch darstellen lassen. Die gewählte Lösung wird in einer anderen Publikation im Detail beschrieben [MitMur90].

Der ganze Ablauf der Codegenerierung wird von einem Dokument einer speziellen Dokumentenklasse (TCodeGenDocument) im Modul *UCodeGenDocument* gesteuert. Die Klasse stellt einerseits Methoden zur Verfügung, um den Code aufzubauen. Unter anderem umfassen diese Methoden die Verwaltung der Register in einem Codeblock, die automatische Auflösung von Sprungzielen im Code und die Bildung von Codewörtern für die ADAM-Architektur aus benutzerfreundlicheren Datentypen. Zudem gibt es Methoden, um alle notwendigen Datenstrukturen für den Debugger auf der ADAM-Architektur zu generieren. Ebenso werden aus den entsprechenden syntaktischen Strukturen automatisch die Sprungtabellen und die Typ-Deskriptoren (vgl. Abschnitt 6.4.2 "Typ-Deskriptoren") generiert. Andererseits wird von einer Methode dieser Klasse aus der ganze Ablauf der Codegenerierung in Phasen gesteuert. Die Klasse ist so gestaltet, dass sie unverändert auch für die konventionelle Codegenerierung der sequentiellen Teile in FOOL [MurMar92] verwendet werden kann.

Nach der Graph-Generierung wird der Graph ein erstes Mal konventionell optimiert. Die drei Module *Optimizer, OptimizerLib* und *Metamachine* enthalten alle notwendigen Prozeduren für die drei optimierenden Graph-Transformationen "Entfernung gemeinsamer Unterausdrücke", "Vorausberechnung konstanter Werte" und "Berechnung schleifeninvarianter Ausdrücke ausserhalb der Schleifen". Das Module *Optimizer* führt mit der Hilfe zusätzlicher Funktionen aus dem Modul *OptimizerLib* die notwendigen Transformationen auf dem Graph durch. Das Modul *Metamachine* enthält eine genaue Nachbildung aller Instruktionen der ADAM-Architektur zur Auswertung konstanter Ausdrücke zur Uebersetzungszeit. Zusätzliche Informationen zu den Optimierungen und zu der Implementation finden sich in [SkeWel85] und [Boldin91].

Für die Partitionierung der Graphen in Codeblöcke nach den in Kapitel 5 "Partitionierung von Datenflussgraphen in Codeblöcke" gezeigten Methoden erfolgt in den Modulen *FunctionExpansion* und *CostAssignment*. Das Modul CostAssignment enthält alle Funktionen für die Kostenzuweisung (vgl. Abschnitt 5.2 "Kostenzuweisung"). Es bestimmt auf Grund der Kostenzuweisung auch für jeden FORALL-Ausdruck anhand der gesetzten Parameter die Anzahl paralleler Codeblöcke. Im Modul FunctionExpansion wird der Aufrufgraph für das ganze Programm gebildet und zu kleine Funktionen expandiert. Die Beschreibung einer ersten Version dieser Module

finden sich in [Marchi90]. Die Module *FunctionExpansion* und *CostAssignment* importieren sich gegenseitig (Fig. 8.1), weil die Expansion von Funktionen die Kostenstruktur in einem Graphen verändert und umgekehrt die Expansion von Funktionen durch die Kostenstruktur gesteuert wird.

Das Modul *GraphTransformations* enthält eine Reihe von Transformationen auf Graphen, die von verschiedenen anderen Modulen gebraucht werden können. Direkt in diesem Modul sind jene Transformationen zu finden, welche die Knoten zur Verwaltung grosser Datenstrukturen einsetzen (vgl. Kapitel 6 "Effiziente Verwaltung von Datenstrukturen").

Im Modul *Sequencer* werden die Graphen nach den in Kapitel 7 "Sequentialisierung und Codegenerierung" gezeigten Methoden sequentialisiert (in Listenform überführt).

Das Modul *Codegenerator* schliesslich generiert aus diesen Listen den Code, indem es die zusammengesetzten Knoten in geeignete Kontrollfluss-Instruktionen umsetzt. Es werden dafür die Methoden aus *UCodeGenDocument* verwendet, um den eigentlichen Maschinencode aus abstrakteren Instruktionen zu generieren. Dies ist auch der Grund dafür, dass sich *Codegenerator* und *UCodeGenDocument* gegenseitig importieren.

Die Tabelle 8.1 gibt zur Information die Grössen der Module in Zeilen an, um einen Eindruck über die Komplexität der einzelnen Programmteile zu vermitteln, wobei sich der Autor bewusst ist, dass die Anzahl der Programmzeilen nur ein oberflächliches Mass für die Komplexität eines Programms ist. So steckt beispielsweise ein hoher konzeptioneller Aufwand hinter den Algorithmen im Modul "Sequencer", der schliesslich zu einem sehr kompakten Algorithmus führte.

Modulname	Anzahl Zeilen im Definitionsmodul	Anzahl Zeilen im Implementationsmodul
CodeGenerator	17	2208
CostAssignment	27	1033
FunctionExpansion	54	677
GraphDisplay	96	2313
GraphGeneration	14	1230
GraphTransformations	44	1161
GraphTypes	286	314
Metamachine	155	555
Optimizer	23	360
OptimizerLib	97	1311
Sequencer	13	210
UCodeGenDocument	562	2772

Tabelle 8.1: Grössen der einzelnen Module

8.2. Einbettung in den inkrementellen Compiler

Der beschriebene Codegenerator für MFL ist nicht Teil eines konventionellen Compilers, sondern Bestandteil einer gesamten *Programmierumgebung*. Im Zentrum der Programmierumgebung steht eine gemeinsame Datenstruktur (vgl. Fig. 8.2), die im folgenden mit *Dokument* bezeichnet werden soll. Das Dokument kann mit Hilfe verschiedener *Programmierwerkzeuge* (Texteditor, graphischen Editoren, Browser, Codegenerierung, Generierung von Debugger-Informationen) inspiziert und verändert werden. Gegenstand dieser Arbeit sind nur die in Figur 8.2 grau markierten Werkzeuge. Eine genauere Beschreibung der Umgebung findet sich in [MarMur92b].

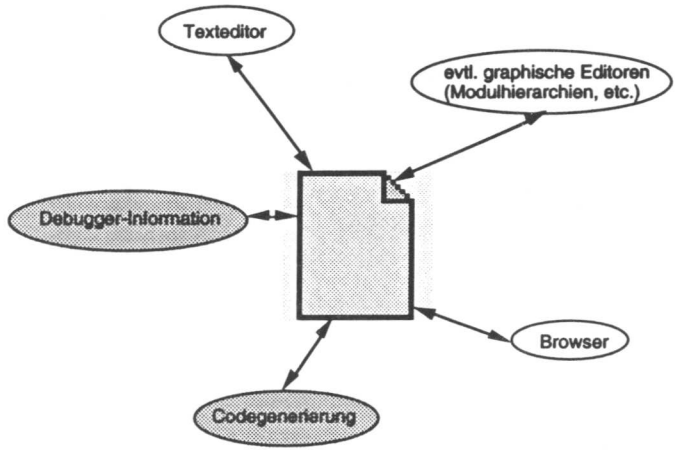


Fig. 8.2: Verschiedene Werkzeuge arbeiten auf gemeinsamer Datenbasis (Dokument)

Entscheidend für die stetige Nachführung aller Sichten auf das Dokument ist, dass es sich dabei nicht um ein gewöhnliches Text-Dokument handeln kann. Ein solches würde es beispielsweise nicht erlauben, eine Browser-Sicht (vgl. Fig. 8.4) dauernd nachzuführen. Deshalb muss dass Dokument eine ganze Reihe von internen Darstellungen ständig nachführen. In diesem Sinne kann man von *inkrementeller Compilation* sprechen.

Die Datenstruktur des Dokuments ist als Klassenhierarchie aufgebaut (vgl. Fig. 8.3):

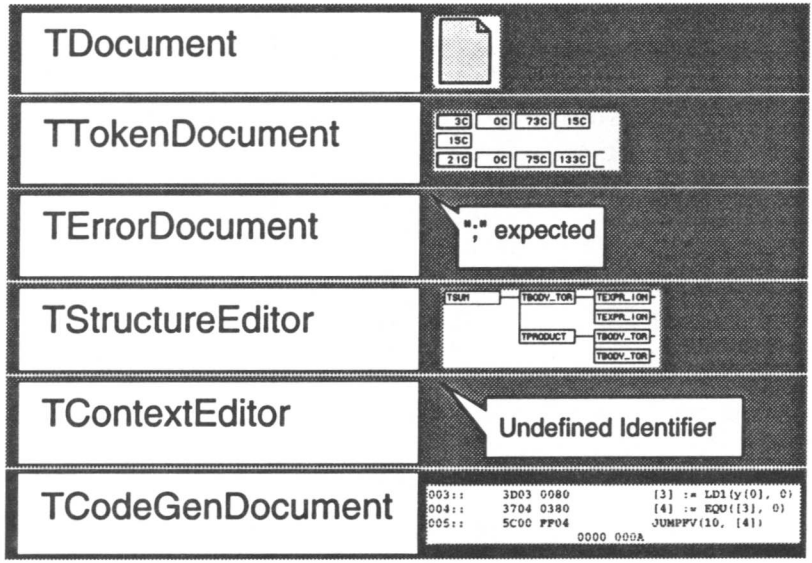


Fig. 8.3: Struktur des Dokuments

Zuoberst in der Klassenhierarchie kommt "TDocument", die MacApp-Standardklasse für Dokumente, welche eine Reihe von Grundfunktionen (Oeffnen, Speichern, etc.) für Dokumente erlaubt.

Davon abgeleitet ist die Klasse "TTokenDocument", welches den Programmtext als zweidimensionale Liste von lexikalischen Elementen (Bezeichner, Schlüsselwörter, etc.) verwaltet und darauf gewisse primitive Editieroperationen (Ausschneiden, Kopieren, Einsetzen, etc.) implementiert.

Auf der nächsten Ableitungsstufe folgt die Klasse "TErrorDocument". Sie erlaubt es, die lexikalischen Elemente mit Fehlermeldungen zu attributieren, welche sich mit Hilfe eines speziellen Werkzeugs anzeigen lassen.

Die Klasse "TStructureEditor" auf der nächsten Stufe verwaltet das ganze Programm als Baum syntaktischer Strukturen entsprechend der Syntax der Programmiersprache. Das Besondere an diesem Strukturediteor ist, dass er auch unvollständige oder syntaktisch falsche Strukturbäume handhaben und dies über die Methoden von "TErrorDocument" auch anzeigen kann. Auch auf dieser Ebene sind die oben beschriebenen, primitiven Editierfunktionen implementiert.

Die Klasse "TContextEditor" schliesslich verwaltet Kontextbeziehungen der Sprache, welche sich nicht über die Syntax darstellen lassen. Als typisches Beispiel dafür stellt es die Beziehungen zwischen programmiersprachlichen Objekten (Variablen, etc.) und Designatoren (Namen, die diese Variablen in einem Ausdruck bezeichnen) her.

Auf der höchsten Ableitungsstufe folgt schliesslich die Klasse "TCodeGenDocument". Ihre Aufgabe ist es, bei jedem Abspeichern (Save) des Dokuments, den Code zu generieren und in geeigneter Form abzuspeichern (vgl. Abschnitt 8.1 "Modulstruktur").

Die in dieser Arbeit beschriebenen Konzepte und Funktionen passen an zwei Stellen in dieses Gesamtkonzept. Einerseits gibt es die Klasse "TCodeGenDocument", welche die grundlegenden Mechanismen zur Speicherung von Maschinencode und Debugger-Information zur Verfügung stellt. Andererseits gibt es eine Reihe von Werkzeugen (Graph-Darstellung, Graph-Generierung, Optimierungen, Sequentialisierung, Codegenerierung), welche eine spezielle Sicht auf das Programm (Graph) erlauben und die Coderepräsentation des Programms verändern können.

Die gezeigte Form von Programmierumgebung wurde im Kontext des FOOL/MFL-Projekts entworfen und implementiert. Sie ist jedoch so allgemein gestaltet, dass sie sich leicht auch für andere Programmiersprachen verwenden lässt. Zu diesem Zweck wurden Werkzeuge geschaffen, die es erlauben, an Hand einer Sprachbeschreibung, eine solche Umgebung automatisch zu generieren [Murer92].

8.3. Benutzerschnittstelle

Die MFL/FOOL-Programmierumgebung präsentiert sich dem Benutzer hauptsächlich über den *System-Browser*. Das *Brower-Fenster* (vgl. Fig. 8.4) ist in vier Teile gegliedert. Im linken, oberen Teil des Fensters ist eine Liste der aktuell im System geladenen Module. Links unten befindet sich das *Editierfenster* des inkrementellen Compilers, in welchem die aktuell angezeigten Programmteile verändert werden können. Links neben dem Editierfenster gibt es eine Palette, um vorbereitete Strukturen (Funktion, Schleife, etc.) auf Knopfdruck einzusetzen. Der Editor stellt alle nach den Benutzerschnittstellen-Standards des MacIntosh gängigen Operationen zur Verfügung. Rechts oben wird eine Liste aller programmiersprachlichen Objekte (Variablen, Konstanten, Funktionen, etc.), welche auf der aktuellen Ebene dargestellt sind, angezeigt. Durch Anklicken eines Objekts wird der im Editierfenster angezeigte Ausschnitt des Programms auf dieses Objekt eingeschränkt. Mit Hilfe des *Filter-Menus*, links unten in diesem Teil, lässt sich die Klasse der anzuzeigenden Objekte einschränken. Gleich rechts daneben befindet sich ein Menu, um aus den Tiefen der Objekt-Hierarchie wieder aufzutauchen. Im unteren rechten Teil lassen sich verschiedene graphische Sichten auf das Programm darstellen. Im gezeigten Fenster ist die Sicht auf den Datenflussgraph eingestellt, wobei aktuell der THEN-Graph des vollständig sichtbaren IF-THEN-ELSE-Konstrukts im linken Teil des Fensters angezeigt wird. Mit dem Menu links unten in diesem Teil, lässt sich die gewünschte, graphische Sicht auf das Programm auswählen.

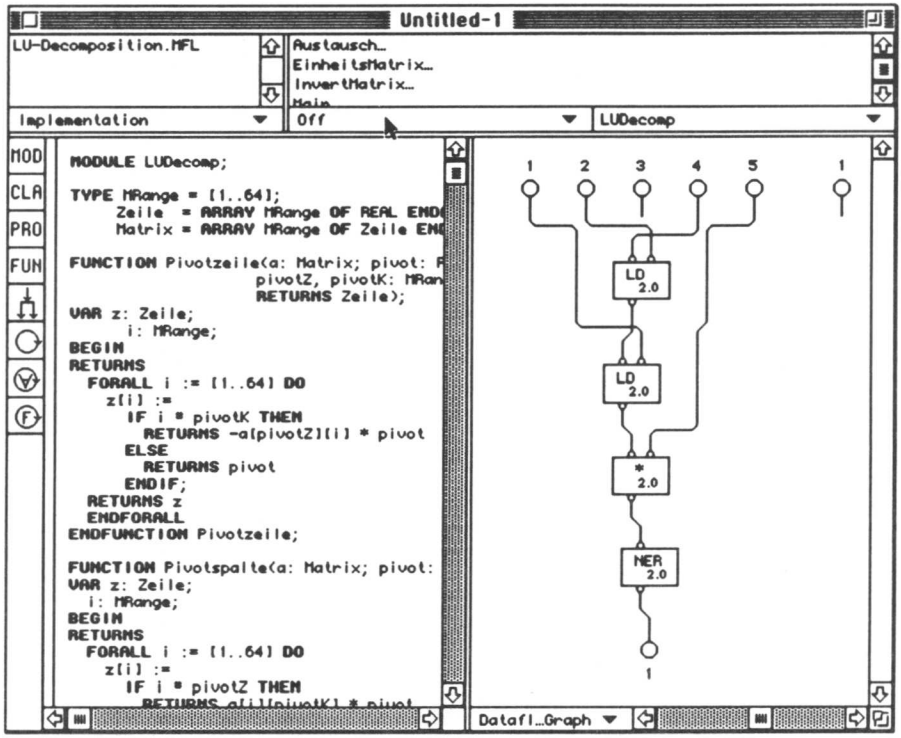


Fig. 8.4: System-Browser mit Graph-Anzeige

Neben dem Browser-Fenster steht dem Benutzer auch noch ein *Hypertext-Fenster* (Fig. 8.5) zur Verfügung. In diesem Fenster können eine Reihe von Dialogen, die sich auf bestimmte syntaktische Strukturen im Editierfenster beziehen, geöffnet werden. Mit Hilfe dieser Dialoge lassen sich abhängig von der Klasse der zugehörigen syntaktischen Struktur gewisse Attribute setzen. Unter diesen Attribute befinden sich auch Hinweise für den Codegenerator. Im gezeigten Beispiel ist je ein Dialog auf ein Typ-Objekt und ein Variablen-Objekt geöffnet. An diesem Beispiel ist für die Codegenerierung interessant, dass beim Typ-Objekt angegeben werden kann, ob es seine Unterfelder expandieren soll (vgl. Abschnitt 6.3 "Objekte mit Unterobjekten"), und beim Variablen-Objekt die Objektklasse (vgl. Abschnitt 6.4. Auswahl der Objektklasse") gesetzt werden kann. Neben dem Setzen von Attributen haben diese Dialoge auch Hypertext-Fähigkeiten. Durch Anklicken des "Declaration"-Knopfs rechts unten im Dialog lässt sich ein neues Browser-Fenster öffnen, welches direkt auf die Deklaration des entsprechenden Objekts fokussiert ist. Die Details zur Implementation dieser Hypertext-Fenster findet man in [Murer91].

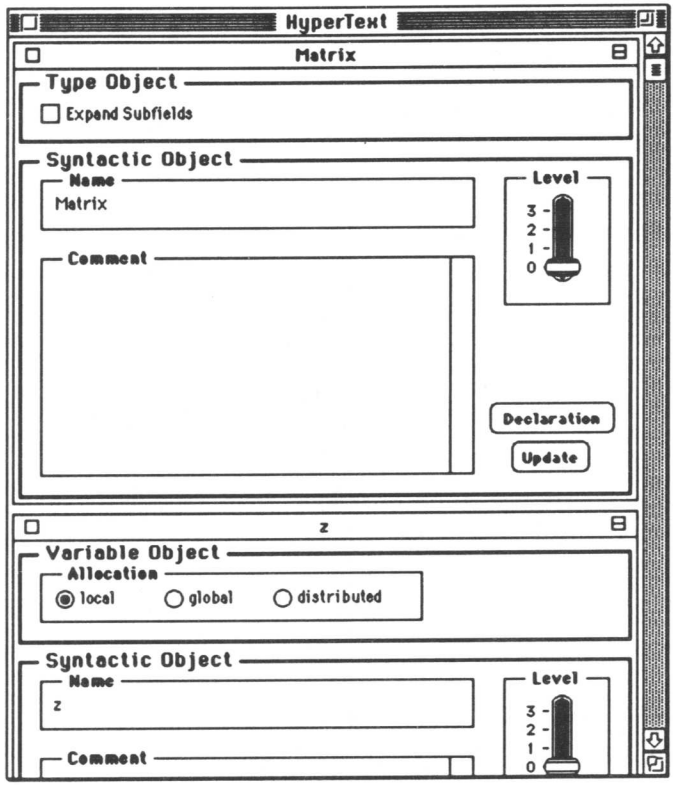


Fig. 8.5: Hypertext-Dialoge zum Setzen der Codegenererierungs-Parameter

Neben den Codegenerierungsattributen für einzelne syntaktische Strukturen, wie sie sich im Hypertext-Fenster setzen lassen, gibt es noch eine Reihe von Parametern, die den Codegenerator als Ganzes betreffen. Sie lassen sich mit Hilfe eines "Preferences"-Dialogs (Fig. 8.6) setzen. Im unteren Teil dieses Dialogs sind alle vom Benutzer wählbaren Parameter des Codegenerators aufgeführt, wobei sie entsprechend der einzelnen Phasen der Codegenerierung gruppiert sind. Im Teil "Partitioning" lassen sich alle für die Partitionierung (vgl. Kapitel 5 "Partitionierung von Datenflussgraphen in Codeblöcke") benötigten Parameter setzen. Bei "Optimizations" wählt man, welche Graph-Optimierungen vor und welche nach der Partitionierung tatsächlich ausgeführt werden sollen. Bei "Checks" lässt sich wählen, ob der Compiler Code generieren soll, um zur Laufzeit die Einfachzuweisungsbedingung (vgl. Abschnitt 3.4.2. "Regeln für die Einfachzuweisung") für Arrays bzw. die Gültigkeit der Indices bei Array-Zugriffen (Range-Check) zu überprüfen. Im Kästchen "Sequentialization" kann man für Versuchszwecke zwischen konventioneller (DFS)

und optimierter (BFS) Sequentialisierung (vgl. Kapitel 7 "Sequentialisierung und Codegenerierung") wählen.

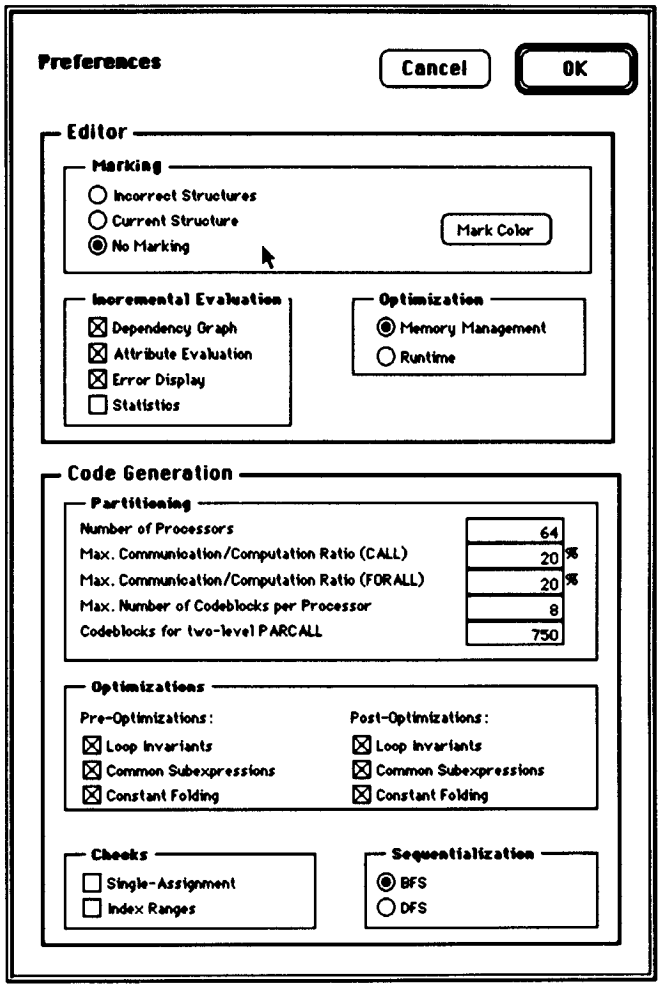


Fig. 8.6: Einstellung der Codegenerierungsparameter

Leer - Vide - Empty

9. Schlussfolgerungen und Ausblick

In diesem Kapitel sollen noch einmal in kompakter Form alle wesentlichen Resultate dieser Dissertation zusammengestellt werden. Es wird zudem versucht, die gezeigten Methoden in einem grösseren Kontext zu sehen.

Im Laufe der Zeit tauchten viele Ideen auf, deren genauere Auswertung den Rahmen einer Dissertation gesprengt hätten. Im zweiten Teil dieses Kapitels werden einige mögliche Fortsetzungen der hier angefangenen Arbeiten skizziert.

9.1. Schlussfolgerungen

In dieser Arbeit wird am Beispiel von MFL und der ADAM-Architektur gezeigt, mit welchen Methoden eine einfache, funktionale Programmiersprache zu parallelem Maschinencode übersetzt werden kann. Die Arbeit ist nur ein Teil des gesamten ADAM-Projekts, dessen Ziel es ist, einen neuen Ansatz für die Architektur paralleler Computer zu erproben. Computerarchitektur wird in diesem Projekt nicht als ein reines Hardware-Entwurfsproblem, sondern als eine umfassende Optimierung verschiedenster Faktoren (Konzept, Theorie, Programmiersprachen, Hardware, etc.) betrachtet. Ueber die Jahre hinweg ist so in mehreren Iterationsschritten aus einem neuen, aber theoretischen Konzept [Wytten90], eine realisierbare Computerarchitektur mit den zugehörigen Programmierwerkzeugen entstanden. Während in [Wytten90] die theoretischen und konzeptionellen Aspekte des Projekts in aller Breite besprochen wurden, lag das Schwergewicht der letzten Jahre eher bei der Verfeinerung und Realisierung der Konzepte, was im Detail noch eine Reihe von Aenderungen notwendig machte. Die vorliegende Arbeit zeigt einen Ausschnitt aus diesem Gesamtkonzept und muss im Zusammenhang mit den anderen Arbeiten im Rahmen des ADAM-Projekts gesehen werden.

Das Ziel, parallele Algorithmen in einer geeigneten Programmiersprache unabhängig von den Details der Maschinenarchitektur (Anzahl der Prozessoren, Netzwerk-Topologie) auszudrücken, wird für eine einfache Sprache im wesentlichen erreicht. Zwar muss der Compiler die Anzahl der Prozessoren auf der Zielmaschine kennen, um einen effizienten Code zu erzeugen, doch am Quellenprogramm muss nichts verändert werden, egal ob das Programm auf einem oder 256 Prozessoren laufen soll. Der Programmierer muss sich, ausser den abstrakten Entscheiden für die Objektklasse und die Abbildung des Typbaums auf den Objektbaum, welche in Zukunft ebenfalls automatisiert werden können, nicht um die Verteilung der Daten und der parallelen Aktivitäten kümmern. Die funktionale

Semantik von MFL garantiert zudem, dass die Berechnungen unabhängig von der Anzahl beteiligter Prozessoren immer dasselbe Resultat liefern.

Eine weitere Erkenntnis dieser Arbeit besteht darin, dass Datenflussgraphen sich als gemeinsamer Zwischencode für die verschiedenen Teile des Compilers eignen, obwohl diese auf grobgranularen Datenflussmaschinen nicht direkt als Maschinencode verwendet werden können. Diese Erkenntnis steht im Einklang mit in den letzten Jahren erzielten Resultaten bei der Uebersetzung konventioneller Programmiersprachen für parallele Maschinen, wo ebenfalls Graphen und Datenabhängigkeiten im Zentrum der entwickelten Theorie stehen (vgl. Abschnitt 4.1. "Funktionale Sprachen und Datenflussgraphen"). Es wurde zudem erläutert, welche wesentlichen Probleme im Laufe der Codegenerierung für grobgranulare Datenflussmaschinen gelöst werden müssen. Es muss dabei betont werden, dass die Wahl der Methoden weniger stark von der Art der Sprache (MFL) als von der Architektur (ADAM) abhängt. Es sollte deshalb nicht schwierig sein, die gezeigten Ergebnisse wenigstens teilweise auf konventionelle Programmiersprachen für die ADAM-Architektur zu übertragen.

Die Codegenerierung für die ADAM-Architektur umfasst neben den üblichen Optimierungen die wesentlichen drei Teile Partitionierung, grosse Datenstrukturen und Sequentialisierung.

Im *Kapitel 5* "Partitionierung" wird das Partitionierungsproblem in einer allgemeinen Form eingeführt, der Sinn einer präzisen Lösung diskutiert und eine heuristische Lösung dafür vorgestellt. Das Problem des Ausgleichs zwischen der eigentlichen Rechenarbeit und dem Zusatzaufwand, welcher benötigt wird, ein Programm parallel anstatt sequentiell auszuführen, ist von allgemeiner Bedeutung beim parallelen Rechnen. Es wird gezeigt, dass zur parallelen Ausführung eines Programms immer auch die sinnvolle Einschränkung der Parallelität gehört, ein Problem, womit die herkömmlichen Datenflussmaschinen Schwierigkeiten haben.

Dass der Parallelisierung von grossen Datenstrukturen eine ebenso zentrale Rolle zukommt wie der Parallelisierung des Programms, wird in *Kapitel 6* "Effiziente Verwaltung von Datenstrukturen" deutlich. Es wird gezeigt, wie sich für Datenstrukturen die *Value-Semantik* auf der Sprachebene mit Hilfe von Referenzzählern effizient auf eine *Referenz-Semantik* auf der Maschinenebene abbilden lässt, wobei in einer parallelen Umgebung der Reduktion von Referenzzähler-Operationen eine grosse Bedeutung zukommt. Ausserdem werden die Konsequenzen der ADAM-

Speicherarchitektur (Menge von Datenobjekten beliebiger Grösse und unterschiedlichen Zugriffscharakteristiken) auf die Codegenerierung aufgezeigt.

Die Frage "Kopieren oder Referenzen zählen" ist für die effiziente Implementation von Datenstrukturen in funktionalen Sprachen von grosser Bedeutung, da dem Benutzer in der Regel keine Wahlmöglichkeiten für die Repräsentation der Daten (keine Zeigertypen) zur Verfügung stehen, wie die neuesten Arbeiten auf diesem Gebiet zeigen (vgl. Abschnitt 6.6 "Arbeiten mit ähnlichen Ansätzen"). Interessant ist zudem die Beobachtung, dass es auf der ADAM-Architektur tatsächlich möglich ist, Speicherlatenz mittels anderer Aktivitäten zu verstecken, so dass deren Minimierung an Bedeutung verliert. Der begrenzende Faktor ist nur die Speicherbandbreite. Darum ist es auch möglich, die Daten statisch zu verteilen (verteilte Objekte), obwohl die Last dynamisch verteilt wird, und somit die Lokalität der Daten im Durchschnitt sehr schlecht ist.

Das *Kapitel 7* "Sequentialisierung und Codegenerierung" beschreibt die verwendeten Methoden, um die Sequenz der Instruktionen zu bestimmen. Diese Sequenz ist in der ADAM-Architektur mit ihren parallelen Funktionseinheiten, den asynchronen Instruktionen und der Datensynchronisation von grosser Bedeutung. Die Folgen einer falschen Sequenz der Instruktionen reichen bis zur vollständigen Vernichtung der Parallelität in einem Programm. Es wird ausserdem bewiesen, dass der verwendete Sequentialisierungsalgorithmus eine in bezug auf die Minimierung der möglichen Kontextwechsel optimale Sequenz liefert.

Parallele Funktionseinheiten sind schon heute Bestandteil jeder leistungsfähigen Prozessorarchitektur. Es ist ausserdem absehbar, dass zukünftige Parallelrechner über logisch gemeinsame Speicher mit variabler Zugriffslatenz verfügen werden. Deshalb wird der Aspekt der optimalen Instruktionssequenz bei der Ausnutzung der Parallelität auf Instruktionsebene in Zukunft eine grössere Rolle spielen als heute. Aehnliche Algorithmen wie die gezeigten werden dann Bestandteil jedes guten Compilers sein.

Die wichtigen qualitativen Aussagen dieser Arbeit werden durch Simulationsexperimente quantitativ untermauert, welche nach Themen geordnet im hinteren Teil jedes Kapitels zu finden sind. Man muss allerdings berücksichtigen, dass mit den gegebenen Mitteln nur Simulationen von bis zu 256 Prozessoren über eine Simulationszeit von maximal einer Sekunde möglich waren. Eine derartige Simulation dauert bereits mehrere Tage und stösst damit an die Grenzen der Zuverlässigkeit der zur Verfügung stehenden Hardware. "Echte" Applikationen sind natürlich

grösser, obwohl man betonen muss, dass in unseren Experimenten schon bedeutend grössere Programme verwendet werden, als dies bei ähnlichen Experimenten in anderen Arbeiten der Fall ist [Sarkar89]. Weitergehende Resultate sind nur noch mit einer Hardware-Implementation der ADAM-Architektur erreichbar, worunter aber die Beobachtbarkeit und die Flexibilität leidet.

Im Zusammenhang mit den Experimenten stellt sich auch die Frage, inwiefern die im Simulator variierbaren Hardware-Parameter (Netzwerktopologie, Bandbreiten, etc.) die Resultate beeinflussen. Da es in dieser Arbeit nicht um die Architektur, sondern um den Codegenerator geht, wurden die Hardware-Parameter, abgesehen von der Anzahl Prozessoren, immer gleich auf Werten, die sich im Zusammenhang mit anderen Experimenten als "vernünftig" herausgestellt haben, gelassen. Experimente, bei denen die Architektur-Parameter verändert werden, findet man in anderen Publikationen aus dem ADAM-Projekt [Maquel90, Maquel92, Mitrov90, Mitrov92, MurFär92].

9.2. Weiterführende Arbeiten

Mit jedem Resultat in der Forschung eröffnet sich eine Reihe von weiteren Fragestellungen. Diese Arbeit ist davon keine Ausnahme. In der Folge werden, kapitelweise nach Themen geordnet, eine Reihe von möglichen Fortsetzungen der hier begonnenen Arbeit aufgezeigt.

Die in *Kapitel 3* "MFL - Eine einfache, funktionale Sprache" gezeigte Sprache ist sehr primitiv und liesse sich für praktische Anwendungen kaum in unveränderter Form verwenden. Gewisse Möglichkeiten, die Ausdruckskraft der Sprache zu erweitern, finden sich bereits im Abschnitt 3.6 "Grenzen von MFL und weiterführende Ideen". Eine weitere überprüfenswerte Möglichkeit, wäre der Entwurf einer Sprache, die die Funktionalität nur auf der Ebene der Funktionen erzwingen würde. Lokale Variablen und Zuweisungen wären dann innerhalb der Grenzen einer Funktion beliebig möglich. VAR-Parameter, Zeigervariablen und globale Variablen blieben wegen der Gefahr von Seiteneffekten verboten. Die grundlegenden Resultate dieser Dissertation liessen sich auch auf eine solche Sprache anwenden, wobei die Idee der Datenflussgraphen nicht direkt übernommen werden könnte. Einzelne "Basic Blocks" [AhSeUl86] der Funktionen lassen sich aber durchaus als gerichtete, azyklische Graphen darstellen, so dass beispielsweise die Methoden der Sequentialisierung aus *Kapitel 7* "Sequentialisierung und Codegenerierung" weiter verwendet werden können. Die Erkenntnisse aus den anderen Kapiteln können mit entsprechenden Anpassungen ebenfalls für die Uebersetzung

anderer Klassen von Programmiersprachen verwendet werden. In [GraKoc92] wird sehr schön erläutert, wie und zu welchem Preis verschiedene, imperative Konstrukte in funktionale Sprachen integriert werden können. Nach der dort vorgeschlagenen Terminologie wäre MFL eine t_0-Sprache. Die neu vorgeschlagene Sprache würde zu den t_1-Sprachen gehören.

In *Kapitel 6* "Effiziente Verwaltung von Datenstrukturen" wird gezeigt, wie sich die Auswahl der verschiedenen Objektklassen und die Abbildung der Typ-Struktur auf die Objektstruktur auf die Laufzeit des erzeugten Codes auswirken. In der aktuellen Version des MFL-Compilers ist es noch die Aufgabe des Benutzers, die geeignete Klasse bzw. Objektstruktur auszuwählen. In diesem Sinne ist es nicht vollständig gelungen, das Ziel, die Details der Architektur vor dem Programmierer zu verbergen, zu erreichen. Es ist aber denkbar, diesen Vorgang in einer weiteren Arbeit, teilweise (mit Benutzerunterstützung) oder ganz zu automatisieren.

Die Deallokation und Allokation verteilter und replizierter Objekte muss, da diese auf allen Prozessoren gemeinsam angelegt werden, über atomare Transaktionen erfolgen. Falls häufig neue Objekte angelegt bzw. entfernt werden müssen, kann dies zu einem Flaschenhals führen. Die eingangs erhobene Forderung nach Skalierbarkeit ist dann nicht erfüllt. Dieses Problem betrifft zwar nicht direkt das Thema dieser Arbeit, sondern eher die ADAM-Architektur im Bereich des Objekt-Managers. Eine Lösung für dieses Problem wäre, die Speicherarchitektur zu hierarchisieren. Man hätte dann replizierte und verteilte Objekte auf jeder Hierarchiestufe. Entsprechend müsste auch die Strategie der Lastverteilung so angepasst werden, dass die Last bevorzugt innerhalb der Grenzen einer Prozessorgruppe (Cluster) bleibt. Vermutlich würden schon zwei Hierarchiestufen genügen, um sehr grosse Maschinen zu konstruieren. Indirekt auf die Codegenerierung würde sich eine derartige Aenderung der Architektur auswirken, indem eine grössere Auswahl von Objektklassen zur Verfügung stehen würde.

Die in *Kapitel 7* "Sequentialisierung und Codegenerierung" gezeigten Prinzipien lassen sich in abgewandelter Form auch für die Codegenerierung für moderne RISC-Architekturen verwenden. Zumindest für einzelne "Basic Blocks" [AhSeUl86] generieren auch Compiler für konventionelle Sprachen gerichtete azyklische Datenflussgraphen. In modernen, superskalaren RISC-Architekturen sind meistens mehrere funktionale Einheiten für die verschiedenen Typen von Operationen und Operanden (Addition und logische Befehle, Multiplikation bzw. Fliesskomma, Ganzzahl, Adressen, etc.). Diese Einheiten können aber in der Regel nur wirklich parallel arbeiten, wenn der Instruktionsfluss eine entsprechende

Mischung von Operationenen enthält. Kommen zwei Befehle hintereinander in der Instruktionssequenz, die von derselben funktionalen Einheit ausgeführt werden sollen, so blockiert der Prozessor, bis diese wieder frei ist, auch wenn alle anderen Einheiten frei wären. Es gibt zwar erste Ansätze, die Instruktionen zur Laufzeit dynamisch umzugruppieren [Inmos91]. Eine solche Lösung ist aber komplex und teuer im Sinne von Chipfläche und kann niemals dieselben Informationen über ein Programm nutzen wie der Compiler. Bei der Sequentialisierung gerichteter, azyklischer Graphen hat hingegen der Compiler die Möglichkeit, mit ähnlichen Algorithmen wie in *Kapitel 7* "Sequentialisierung und Codegenerierung" für eine optimale Auslastung der parallelen funktionalen Einheiten zu sorgen. Es lässt sich dadurch sicher nicht viel Parallelität gewinnen, aber die Methode ist trotzdem sinnvoll, da sie mehr oder weniger unabhängig von der Applikation auf heute vorhandenen, weit verbreiteten Maschinen viel bringt. Immerhin beschleunigt eine simple Anwendung derartiger Ideen bereits optimierte Programm auf der SPARC-Architektur [Garner88] um 18-25% [Krishna90].

Sehr primitiv wurde die Registerallokation im MFL-Compiler gelöst, und es zeigt sich schon heute, dass diese Methode zu primitiv ist. Aus diesem Grund sollte einer der Algorithmen für das Auslagern von Registern in den Hauptspeicher (*Register Spilling*) vorgesehen werden. Den klassischen Algorithmus dafür findet man in [Chaitin82]. Allerdings optimiert dieser Algorithmus die Anzahl der Zugriffe auf den Hauptspeicher. Dies ist sicherlich auch für die ADAM-Architektur ein Kriterium. Ebenso wichtig ist aber, dass durch das Auslagern von Registern auf den Hauptspeicher nicht plötzlich unnötig früh synchronisiert wird. Deshalb müssen wohl für Maschinen mit Datensynchronisation über die Register neuartige Auslagerungsmechanismen entwickelt werden.

Um die Resultate nicht zu verfälschen, lief in allen Experimenten nur ein einziges Programm. Dies ist selbstverständlich nicht die Methode, um eine ADAM-Maschine optimal auszulasten. In der Praxis müsste eine solche Maschine im Multiprogramming-Betrieb gefahren werden, was sich als besonders einfach und billig erweist, da die primitiven Konzepte zur Verwaltung mehrerer Prozesse direkt in der ADAM-Architektur zur Verfügung stehen. Eine wichtige zukünftige Aufgabe wäre, ein Betriebssystem für die ADAM-Architektur zu schreiben, welches Massenspeicher und Kommunikation zu anderen Rechnern unterstützt.

Anhang A: Syntax von MFL

In diesem Anhang ist die gesamte Syntax von MFL in drei Teilen dokumentiert. Der erste Teil umfasst die Definition der lexikalischen Elemente, der zweite und der dritte Teil enthalten die syntaktischen Produktionen für den Deklarations- und den Anweisungsteil der Sprache.

A.1. Lexikalische Elemente

letter = {'a'..'z'} + {'A'..'Z'}.

digit = {'0'..'9'}.

ident = letter {digit | letter}.

integer = digit {digit}.

real = digit {digit} "." digit {digit} |
digit {digit} "E" ("+"|"-") digit {digit}.

A.2. Syntax des Deklarationsteils

FOOL = "MODULE" ident ";" ModuleBlock "ENDMODULE"
ident "." .

ModuleBlock = [ImportDecl] {Declaration} .

ImportDecl = "IMPORT* Import {"," Import } ";".

Import = ident [DefLevel] ["ALIAS" ident].

DefLevel = "(" integer ")".

Declaration = ConstDecl | TypeDecl | VariableDecl | FunctionDecl.

FunctionDecl = "FUNCTION" ident FunctionBlock
"ENDFUNCTION" ident ";" .

FunctionBlock = Signature ";" {Declaration} FunctionBody.

```
ProcedureDecl = "PROCEDURE" ident ProcedureBlock
                "ENDPROCEDURE" ident ";" .

ProcedureBlock = [Signature] ";" {Declaration} ProcedureBody .

Signature = "(" [FormalParameters]
            ["RETURNS" ResultTypeNameList] ")".

ResultTypeNameList = Designator {"," Designator } .

FormalParameters = FormalParamSection {";" FormalParamSection}.

FormalParamSection = FormalParamList ":" Designator.

FormalParamList = ident {"," ident }.

ConstDecl = "CONST" {ConstSection} .

ConstSection = ident "=" Expression ";" .

Literal = Designator | Number.

TypeName = Designator .

TypeDecl = "TYPE" {TypeSection} .

TypeSection = ident "=" Type ";" .

Type = TypeName | Subrange | Enumeration | TupleStructure |
       ArrayStructure.

Subrange = "[" Literal ".." Literal "]".

Enumeration = "(" ident {"," ident } ")".

TupleStructure = "TUPLE" TupleBlock "ENDTUPLE".

TupleBlock = {VarSection} .

ArrayStructure = "ARRAY" IndexType "OF" ElementType
                 "ENDARRAY".

IndexType = Subrange | Designator.
```

ElementType = Designator.

VariableDecl = "VAR" {VarSection} .

VarSection = VariableList ":" TypeName ";" .

VariableList = ident {"," ident}.

A.3. Syntax des Anweisungsteils

FunctionBody = "BEGIN" StatementSeq "RETURNS" Expression .

StatementSeq = {Assignment ";"} .

DesignatorList = Designator {"," Designator}.

Assignment = DesignatorList ":=" Expression.

Expression = ComplexExpression {"," ComplexExpression}.

ComplexExpression = IFExpression | FORExpression | FORALLExpression | LOOPExpression | Relation.

Relation = Sum [RelOperator Relation].

Sum = SignedTerm [AddOperator Sum].

SignedTerm = ['+'|'-'] Product.

Product = SignedFactor [MulOperator Product].

SignedFactor = Number | "NOT" SignedFactor | "("Expression")" | Designator.

IFExpression = "IF" Expression "THEN"
StatementSeq "RETURNS" Expression
{"ELSIF" Expression "THEN"
StatementSeq "RETURNS" Expression}
"ELSE" StatementSeq "RETURNS" Expression
"ENDIF".

```
FORExpression     =  "FOR" FORIndex ["INIT" StatementSeq] "DO"
                     StatementSeq
                     "RETURNS" Expression "ENDFOR".

FORALLExpression  =  "FORALL" FORALLIndex "DO"
                     StatementSeq "RETURNS"
                     Expression "ENDFORALL".

LOOPExpression    =  "LOOP" StatementSeq "DO"
                     StatementSeq
                     "EXIT" Expression ";"
                     StatementSeq
                     "RETURNS" Expression "ENDLOOP" .

Range             =  "[" Expression ".." Expression "]" .

FORIndex          =  Designator ':=' Range.

FORALLIndex       =  Designator ':=' Range {"CROSS" Designator ':=' Range}
                     .

RelOperator       =  "=" | "#" | "<" | ">" | ">=" | "<=" .

AddOperator       =  "+" | "-" | "OR" | "||" .

MulOperator       =  "*" | "/" | "DIV" | "MOD" | "&" .

Designator        =  ["NEW"] ident {QualifiedIIndexedICall}.

Qualified         =  "." ident.

Indexed           =  "[" Expression "]".

Call              =  "(" [Expression] ")".

Number            =  real | integer.
```

Anhang B: MFL-Beispiele

In diesem Anhang werden die MFL-Quellenprogramme aller in der Arbeit verwendeten Beispielprogramme angegeben. All die gezeigten Programme lassen sich durch den MFL-Compiler übersetzen und laufen korrekt auf dem ADAM-Simulator. Zu jedem Programm ist kurz angegeben, woher es stammt, was es tut und weshalb es als Beispiel interessant ist. Die Programmausführung beginnt immer bei der Funktion "Main".

B.1. Ackermann-Funktion

Die Ackermann-Funktion ist ein klassisches Beispiel aus der Berechnungstheorie. Interessant daran ist, dass diese Funktion trotz mehrfachen rekursiven Aufrufen nicht parallel ist und somit von der Lastverteilung auf der ADAM-Architektur auch nicht auf mehrere Prozessoren verteilt werden sollte.

```
MODULE Ackermann;

FUNCTION A (x,y : INTEGER RETURNS INTEGER);
   VAR b, c: INTEGER;
BEGIN
   RETURNS
      IF x = 0 THEN
         RETURNS y + 1
      ELSIF (y = 0) THEN
         RETURNS A(x-1, 1)
      ELSE
         RETURNS A(x-1, A(x, y-1))
      ENDIF
ENDFUNCTION A;

FUNCTION Main (RETURNS INTEGER);
BEGIN
   RETURNS A (3,2)
ENDFUNCTION Main;

ENDMODULE Ackermann.
```

B.2. Adaptive Binäre Integration

Diese Funktion integriert eine Funktion numerisch, indem der zu integrierende Bereich rekursiv halbiert und die Fläche darunter nach der Trapezregel berechnet wird. Die Rekursion wird abgebrochen, sobald eine gewisse Genauigkeit erreicht worden ist. Als Beispiel ist dieser Algorithmus interessant, da sich nicht a priori sagen lässt, welche Rekursionstiefe an den verschiedenen Stellen der Funktion notwendig ist, so dass eine dynamische Lastverteilung notwendig ist. Ausserdem ent-

hält das Programm eine Reihe von kleinen Funktionen, an denen sich der Effekt der Partitionierung leicht zeigen lässt (vgl. Abschnitt 5.5.1. "Experimente mit Funktionsexpansion").

```
MODULE BinInt;

FUNCTION F(X: REAL RETURNS REAL);
VAR squareX: REAL;
BEGIN
   squareX := X*X;
RETURNS
   3.0*X*squareX + 2.0*squareX + 5.0
ENDFUNCTION F;

FUNCTION Trap(L, R: REAL RETURNS REAL);
BEGIN
RETURNS
   (R-L) * (F(L) + F(R))/2.0
ENDFUNCTION Trap;

FUNCTION Area(L, R, Est, Tol: REAL RETURNS REAL);
VAR Mid, A1, A2, Newest, halfTol: REAL;
BEGIN
   Mid := (L + R)/2.0;
   A1, A2 := Trap(L, Mid), Trap(Mid, R);
   Newest := A1 + A2;
RETURNS
   IF ABSR(Est - Newest) < Tol THEN
      RETURNS Newest
   ELSE
      halfTol := Tol/2.0;
      RETURNS Area(L, Mid, A1, halfTol) + Area(Mid, R, A2, halfTol)
   ENDIF
ENDFUNCTION Area;

FUNCTION Main(RETURNS REAL);
VAR A, B, ITol: REAL;
BEGIN
   A := 1.0;
   B := 5.0;
   ITol := 0.01;
RETURNS
   Area(A, B, Trap(A,B), ITol)
ENDFUNCTION Main;

ENDMODULE BinInt.
```

B.3. Explosion - Rekursive Parallelität

Die Explosion ist ein synthetisches Programm mit regelmässiger, baumartiger Parallelität und kann als Prototyp für verschiedene ähnliche Algorithmen (Summe, Maximum, Minimum, etc.) gelten. Es wird hauptsächlich für Experimente zum Thema Lastverteilung auf der ADAM-Architektur benutzt [Maquel90].

```
MODULE Explosion;

FUNCTION Explode(i: INTEGER RETURNS INTEGER);
BEGIN
RETURNS
   IF i = 0 THEN
      RETURNS 1
   ELSE
      RETURNS Explode(i-1) + Explode(i-1)
   ENDIF
ENDFUNCTION Explode;

FUNCTION Main(RETURNS INTEGER);
BEGIN
   RETURNS Explode(10)
ENDFUNCTION Main;

ENDMODULE Explosion.
```

B.4. Parallele FFT

Die FFT und ihre Derivate sind wohl die wichtigsten parallelen Algorithmen. Die hier gezeigte MFL-Version basiert auf einer SISAL-Version von Vivek Sarkar, welche wiederum aus einem Public-Domain-Programm stammt.

```
MODULE FFT;

TYPE
   Samples = ARRAY [1..512] OF REAL ENDARRAY;
   ComplexArray = TUPLE re, im: Samples; ENDTUPLE;

FUNCTION AddC(aRe, aIm, bRe, bIm: REAL RETURNS REAL, REAL);
BEGIN
   RETURNS aRe + bRe, aIm + bIm
ENDFUNCTION AddC;

FUNCTION SubC(aRe, aIm, bRe, bIm: REAL RETURNS REAL, REAL);
BEGIN
   RETURNS aRe - bRe, aIm - bIm
ENDFUNCTION SubC;

FUNCTION MulC(aRe, aIm, bRe, bIm: REAL RETURNS REAL, REAL);
BEGIN
   RETURNS
      aRe * bRe - aIm * bIm,
      aIm * bRe + aRe * bIm
ENDFUNCTION MulC;

FUNCTION AbsC(aRe, aIm: REAL RETURNS REAL);
BEGIN
   RETURNS SQRT(aRe * aRe + aIm * aIm)
ENDFUNCTION AbsC;
```

```
FUNCTION Fourier(z: ComplexArray RETURNS ComplexArray);
VAR pi, theta: REAL;
   n, m, l, i, k, j: INTEGER;
   t: ComplexArray;
   reT, imT: Samples;
   expValRe, expValIm, diffRe, diffIm: REAL;
BEGIN
   pi, n, m := 3.1415926, 512, 256;
   theta := -pi/FLOAT(m);
   RETURNS
      LOOP
         l := 1;
         t := z;
      DO
         NEW l := l + l;
         NEW t.re, NEW t.im :=
            FORALL i := [1..n] DO
               k := ((i-1)/NEW l) * l;
               j := k + l;
               reT[i], imT[i] :=
                  IF ((i-1)/l) MOD 2 = 0 THEN
                     RETURNS AddC(t.re[i-k], t.im[i-k],
                                  t.re[(i-k)+m], t.im[(i-k)+m])
                  ELSE
                     expValRe := COS(FLOAT(k) * theta);
                     expValIm := SIN(FLOAT(k) * theta);
                     diffRe, diffIm := SubC(t.re[i-j], t.im[i-j],
                                            t.re[(i-j)+m], t.im[(i-j)+m]);
                     RETURNS MulC(expValRe, expValIm, diffRe, diffIm)
                  ENDIF;
            RETURNS reT, imT
            ENDFORALL;
      EXIT l > m;
      RETURNS t
      ENDLOOP
ENDFUNCTION Fourier;

FUNCTION Main(RETURNS ComplexArray);
VAR sample: ComplexArray;
   reS, imS: Samples;
   pi: REAL;
   i: INTEGER;
BEGIN
   pi := 3.1415926;
   sample.re, sample.im :=
      FORALL i := [1..512] DO
         reS[i] := SIN(FLOAT(i)*pi/8.0);
         imS[i] := 0.0;
      RETURNS
         reS, imS
      ENDFORALL;
   RETURNS Fourier(sample)
ENDFUNCTION Main;

ENDMODULE FFT.
```

B.5. Fibonacci-Zahlen

Das Fibonacci-Programm gewinnt seine Parallelität wiederum durch Rekursion, wobei der Aufrufbaum stark asymmetrisch ist. Auch diese Funktion dient hauptsächlich dem Test der Lastverteilung af der ADAM-Architektur.

```
MODULE Fibonacci;

FUNCTION Fib(n: INTEGER RETURNS INTEGER);
BEGIN
RETURNS
   IF n <= 1 THEN
      RETURNS 1
   ELSE
      RETURNS Fib(n-1) + Fib(n-2)
   ENDIF
ENDFUNCTION Fib;

FUNCTION Main(RETURNS INTEGER);
BEGIN
   RETURNS Fib(20)
ENDFUNCTION Main;

ENDMODULE Fibonacci.
```

B.6. Livermore-Loop 1

Die synthetischen Livermore-Loops [McMaho86] sind beliebte Benchmarks, um die Leistungsfähigkeit eines Compilers zu demonstrieren. Aus diesem Grund haben auch zwei parallele Livermore-Loops Eingang in diese Sammlung von Algorithmen gefunden. Sie dienen hauptsächlich für die Experimente zum Thema "Partitionierung von FORALL-Schleifen" (vgl. Abschnitt "5.5.2. Experimente mit FORALL-Schleifen").

```
MODULE LLL1;

TYPE ARange = [1..1001];
     OneDimReal = ARRAY ARange OF REAL ENDARRAY;

FUNCTION Loop1(i: INTEGER; Q, R, T: REAL; Y,Z: OneDimReal RETURNS OneDimReal);

VAR k: INTEGER;
    X: OneDimReal;
BEGIN
   RETURNS
      FORALL k := [i..990] DO
         X[k] := Q + (Y[k] * ((R * Z[k+10]) + (T * Z[k+11])));
      RETURNS X
      ENDFORALL
ENDFUNCTION Loop1;
```

```
FUNCTION FillWithOnes(RETURNS OneDimReal);
VAR X: OneDimReal;
    k: INTEGER;
BEGIN
   RETURNS
      FORALL k := [1..1001] DO
         X[k] := 1.0;
      RETURNS X
      ENDFORALL
ENDFUNCTION FillWithOnes;

FUNCTION Main(RETURNS OneDimReal);
VAR allOnes: OneDimReal;
BEGIN
   allOnes := FillWithOnes();
RETURNS
   Loop1(2.0, 3.0, 21.7, allOnes, allOnes)
ENDFUNCTION Main;

ENDMODULE LLL1.
```

B.7. Livermore Loop 7

Bemerkungen zu diesem Programm sh. oben (B.6.).

```
MODULE LLL7;

TYPE ARange = [1..1001];
     OneDimReal = ARRAY ARange OF REAL ENDARRAY;

FUNCTION Loop7(R, T: REAL; U, Y, Z: OneDimReal RETURNS OneDimReal);
VAR X: OneDimReal;
      k: ARange;
BEGIN
   RETURNS
      FORALL k := [1..995] DO
         X[k] := U[k] +
            (R * (Z[k] + (R * Y[k]))) +
            (T * (U[k + 3] + (R * U[k + 2] + (R * U[k + 1])))) +
            (T * (U[k + 6] + ((R * U[k + 5] + (R * U[k + 4])))))));
      RETURNS X
      ENDFORALL
ENDFUNCTION Loop7;

FUNCTION FillWithOnes(RETURNS OneDimReal);
VAR X: OneDimReal;
      k: ARange;
BEGIN
   RETURNS
      FORALL k := [1..1001] DO
         X[k] := 1.0;
      RETURNS X
      ENDFORALL
ENDFUNCTION FillWithOnes;
```

```
FUNCTION Main(RETURNS OneDimReal);
VAR allOnes: OneDimReal;
BEGIN
   allOnes := FillWithOnes();
RETURNS
   Loop7(2.0, 3.0, allOnes, allOnes, allOnes)
ENDFUNCTION Main;

ENDMODULE LLL7.
```

B.8. Matrix-Inversion nach der Gauss-Methode

Der Gauss-Algorithmus zur Lösung von Gleichungssystemen oder zum Invertieren von Matrizen ist einer der ältesten Algorithmen in der Numerik. Obwohl er normalerweise so implementiert wird, dass die neu berechneten Werte auf die alte Matrix zurückgeschrieben werden, was in MFL nicht möglich ist, gibt es eine mehr oder weniger elegante MFL-Implementation. Das Beispiel wurde ausgewählt, da sich daran die subtilen Einflüsse der Auswahl der Objektklassen und der Abbildung der Typenbäume auf die Objektbäume gut zeigen lassen (vgl. Abschnitt 6.5.3. "Experimente mit Gauss").

```
MODULE Gauss;

TYPE MRange = [1..64];
     Zeile  = ARRAY MRange OF REAL ENDARRAY;
     Matrix = ARRAY MRange OF Zeile ENDARRAY;

FUNCTION Pivotzeile(a: Matrix; pivot: REAL;
                    pivotZ, pivotK: MRange
                    RETURNS Zeile);
VAR z: Zeile;
    i: MRange;
BEGIN
RETURNS
   FORALL i := [1..64] DO
      z[i] :=
         IF i # pivotK THEN
            RETURNS -a[pivotZ][i] * pivot
         ELSE
            RETURNS pivot
         ENDIF;
   RETURNS z
   ENDFORALL
ENDFUNCTION Pivotzeile;
```

```
FUNCTION Pivotspalte(a: Matrix; pivot: REAL; pivotZ, pivotK: MRange
                     RETURNS Zeile);
VAR z: Zeile;
    i: MRange;
BEGIN
RETURNS
   FORALL i := [1..64] DO
      z[i] :=
         IF i # pivotZ THEN
            RETURNS a[i][pivotK] * pivot
         ELSE
            RETURNS pivot
         ENDIF;
      RETURNS z
   ENDFORALL
ENDFUNCTION Pivotspalte;

FUNCTION Austausch(a: Matrix; z,k: MRange RETURNS Matrix);
VAR
   pivotZ, pivotK, zzz: Zeile;
   newA: Matrix;
   i, j: MRange;
   pivot: REAL;
BEGIN
   pivot := 1.0 / a[z][k];
   pivotZ := Pivotzeile(a, pivot, z, k);
   pivotK := Pivotspalte(a, pivot, z, k);
RETURNS
   FORALL i := [1..64] DO
      newA[i] :=
      IF i # z THEN
         RETURNS
            FORALL j := [1..64] DO
               zzz[j] :=
                  IF j # k THEN
                     RETURNS a[i][j] + pivotZ[j] * a[i][k]
                  ELSE
                     RETURNS pivotK[i]
                  ENDIF;
            RETURNS zzz
            ENDFORALL
         ELSE
         RETURNS pivotZ
      ENDIF;
   RETURNS newA
   ENDFORALL
ENDFUNCTION Austausch;

FUNCTION InvertMatrix(a: Matrix RETURNS Matrix);
VAR
   aa: Matrix;
   i: MRange;
BEGIN
   RETURNS
      FOR i := [2..64] INIT
         aa := Austausch(a, 1, 1);
      DO
         aa := Austausch(aa, i, i);
      RETURNS
         aa
      ENDFOR
ENDFUNCTION InvertMatrix;
```

```
FUNCTION EinheitsMatrix(RETURNS Matrix);
VAR   i,j: MRange;
   c: Matrix;
BEGIN
   RETURNS
      FORALL i := [1..64] CROSS j := [1..64] DO
         c[i][j] :=
            IF i >= j THEN
               RETURNS 1.0
            ELSE
               RETURNS 0.0
            ENDIF;
      RETURNS c
   ENDFORALL
ENDFUNCTION EinheitsMatrix;

FUNCTION Main(RETURNS Matrix);
BEGIN
   RETURNS InvertMatrix(EinheitsMatrix())
ENDFUNCTION Main;

ENDMODULE Gauss.
```

B.9. Berechnung der Mandelbrot-Menge

Die Berechnung der Mandelbrot-Menge ist ebenfalls ein Klassiker unter den parallelen Algorithmen. Seine Besonderheit ist, dass die Anzahl der Interationen in der inneren Schleife mit den Daten stark schwankt und somit die einzelnen Codeblöcke der parallelen Schleife verschieden lange laufen. Aus diesem Grund lässt sich die Last für dieses Beispiel nur dynamisch verteilen, wo für die ADAM-Architektur sehr geeignet ist.

```
MODULE Mandelbrot;

TYPE
   Picture = ARRAY [1..256],[1..256] OF BOOLEAN ENDARRAY;

FUNCTION Apfelmann (left, right, bottom, top: REAL RETURNS Picture);
VAR pict: Picture; count : INTEGER;
    xPos, yPos, x2, y2, x, y, hStep, vStep: REAL;
    i, j: INTEGER;
BEGIN
   hStep := (right - left) / 256.0;
   vStep := (top - bottom) / 256.0;
RETURNS
   FORALL i:= [1..256] CROSS j:= [1..256] DO
      xPos, yPos:= FLOAT(i)*hStep + left, FLOAT(j)*vStep + bottom;
      pict [i, j] :=
         LOOP
            x, y := xPos, yPos;
            x2, y2:= x*x, y*y;
            count := 1;
         DO
         EXIT (x2 + y2 >= 4.0) OR (count >= 64);
            NEW x := (x2 - y2) + xPos;
            NEW y := (x * y * 2.0) + yPos;
            NEW x2:= x * x;
            NEW y2:= y * y;
            NEW count := count + 1;
         RETURNS count < 64
         ENDLOOP;
   RETURNS pict
   ENDFORALL
ENDFUNCTION Apfelmann;

FUNCTION Main (RETURNS Picture);
BEGIN
   RETURNS Apfelmann (-2.0, 0.5, -1.25, 1.25)
ENDFUNCTION Main;

ENDMODULE Mandelbrot.
```

B.10. Matrix-Multiplikation

In diesem Beispiel ist eine einfache Multiplikation zweier Matrizen implementiert. Das Beispiel wurde gewählt, da die Matrix-Multiplikation einerseits repräsentativ ist für eine ganze Reihe von Matrix-Algorithmen in der Numerik und andererseits keine einfache Möglichkeit besteht, die Daten so auf den Prozessoren zu verteilen, dass alle Zugriffe lokal sind. Die Matrix-Multiplikation stellt deshalb hohe Anforderungen an das Netzwerk, so dass sich die Effekte der Datenverteilung schön zeigen lassen.

```
MODULE MatMul;

TYPE Range = [1..64];
     Zeile  = ARRAY Range OF REAL ENDARRAY;
     Matrix = ARRAY Range OF Zeile ENDARRAY;

FUNCTION MatInit(RETURNS Matrix);
VAR i,j: Range;
    c: Matrix;
BEGIN
RETURNS
   FORALL i:= [1..64] CROSS j:= [0..7] DO
      c[i][j*8+1] := 1.0; (* loop unrolling *)
      c[i][j*8+2] := 1.0;
      c[i][j*8+3] := 1.0;
      c[i][j*8+4] := 1.0;
      c[i][j*8+5] := 1.0;
      c[i][j*8+6] := 1.0;
      c[i][j*8+7] := 1.0;
      c[i][j*8+8] := 1.0;
   RETURNS c
   ENDFORALL
ENDFUNCTION MatMul;

FUNCTION MatMul(a, b: Matrix RETURNS Matrix);
VAR i,j,k: Range;
      sum: REAL;
      c: Matrix;
BEGIN
RETURNS
   FORALL i := [1..64] CROSS j := [1..64] DO
      c[i][j] :=
         FOR k := [1..64] INIT
            sum := 0.0;
         DO
            NEW sum := sum + a[i][k] * b[k][j];
         RETURNS sum
         ENDFOR;
      RETURNS c
   ENDFORALL
ENDFUNCTION MatMul;

FUNCTION Main (RETURNS Matrix);
VAR a: Matrix;
BEGIN
   a := MatInit(8);
   RETURNS MatMul(a, a)
ENDFUNCTION Main;

ENDMODULE MatMul.
```

B.11. Matrix-Multiplikation (zeilenweise)

Dieses Beispiel ist ebenfalls eine Matrix-Multiplikation, wobei gegenüber dem Beispiel B.10 die Schleifenstruktur leicht verändert wurde, wo durch man eine andere Abbildung der Matrix auf den Objektbaum erzwingen kann (vgl. Abschnitt 6.5.2 "Experimente mit der Matrix-Multiplikation).

```
MODULE MatMul2;

TYPE Range = [1..64];
     Zeile  = ARRAY Range OF REAL ENDARRAY;
     Matrix = ARRAY Range OF Zeile ENDARRAY;

FUNCTION MatInit(RETURNS Matrix);
VAR i,j: Range;
    c: Matrix;
BEGIN
RETURNS
   FORALL i:= [1..64] CROSS j:= [0..7] DO
      c[i][j*8+1] := 1.0;
      c[i][j*8+2] := 1.0;
      c[i][j*8+3] := 1.0;
      c[i][j*8+4] := 1.0;
      c[i][j*8+5] := 1.0;
      c[i][j*8+6] := 1.0;
      c[i][j*8+7] := 1.0;
      c[i][j*8+8] := 1.0;
   RETURNS c
   ENDFORALL
ENDFUNCTION MatMul;

FUNCTION MatMul(a, b: Matrix RETURNS Matrix);
VAR i,j,k: Range;
      sum: REAL;
      c: Matrix;
BEGIN
RETURNS
   FORALL i := [1..64] CROSS j := [1..64] DO
      c[i][j] :=
         FOR k := [1..64] INIT
            sum := 0.0;
         DO
            NEW sum := sum + a[i][k] * b[k][j];
         RETURNS sum
         ENDFOR;
   RETURNS c
   ENDFORALL
ENDFUNCTION MatMul;

FUNCTION Main (RETURNS Matrix);
VAR a: Matrix;
BEGIN
   a := MatInit(8);
   RETURNS MatMul(a, a)
ENDFUNCTION Main;

ENDMODULE MatMul.
```

B.12. Rekursiver Mergesort

Im rekursiven Mergesort-Algorithmus in diesem Beispiel wird der Quellen-Array rekursiv in Stücke der Grössen eins oder zwei zerteilt, dann werden diese primitiven Stücke sortiert und beim Zurückkehren aus der Rekursion zum

Resultat-Array gemischt. Das Mischen ist im Gegensatz zum Beispiel in B.14 sequentiell implementiert, so dass man höchstens mit einem Speedup von O(log(n)) rechnen kann. Das Beispiel wurde gewählt, um zu zeigen, wie ein nicht trivialer Divide-et-Impera-Algorithmus in MFL implementiert werden kann.

```
MODULE MSort;

TYPE LargeRange  = [1..512];
     ArrayToSort = ARRAY LargeRange OF INTEGER ENDARRAY;

FUNCTION Random(lastRandom: INTEGER RETURNS INTEGER);
BEGIN
   RETURNS ((lastRandom * 521) + 17) MOD 256
ENDFUNCTION Random;

FUNCTION NewSample(limit: INTEGER RETURNS ArrayToSort);

VAR a: ArrayToSort;
    i: LargeRange;
    lastRandom, m: INTEGER;

BEGIN
RETURNS
  FOR i := [1..limit] INIT
    lastRandom := 13;
  DO
    a[i] := lastRandom;
    lastRandom := Random(lastRandom);
  RETURNS a
  ENDFOR
ENDFUNCTION NewSample;


FUNCTION SortTwo(low, high: INTEGER; stream: ArrayToSort RETURNS ArrayToSort,
INTEGER);

VAR larger: ArrayToSort;
    highFirst, highSecond, index: INTEGER;
BEGIN
   larger[1], larger[2], index :=
      IF high # low THEN
         RETURNS MIN(stream[low], stream[high]),
                 MAX(stream[low], stream[high]), 2
      ELSE
         RETURNS stream[low],0,1
      ENDIF;
   RETURNS larger, index
ENDFUNCTION SortTwo;
```

```
FUNCTION MergeTwo(first, second: ArrayToSort; maxFirst, maxSecond: INTEGER
                  RETURNS ArrayToSort, INTEGER);

VAR firstIndex, secondIndex, index, max: INTEGER;
    larger: ArrayToSort;
BEGIN
   max := maxFirst + maxSecond;
RETURNS
   FOR index := [1..max] INIT
      firstIndex, secondIndex := 1, 1;
   DO
      larger[index], firstIndex, secondIndex :=
      IF (firstIndex <= maxFirst) AND (secondIndex <= maxSecond) THEN
         RETURNS
            IF first[firstIndex] < second[secondIndex] THEN
               RETURNS first[firstIndex], firstIndex+1, secondIndex
            ELSE
               RETURNS second[secondIndex], firstIndex, secondIndex+1
            ENDIF
      ELSE
         RETURNS
            IF firstIndex > maxFirst THEN
               RETURNS second[secondIndex], firstIndex, secondIndex+1
            ELSE
               RETURNS first[firstIndex], firstIndex+1, secondIndex
            ENDIF
      ENDIF;
   RETURNS larger, max
   ENDFOR
ENDFUNCTION MergeTwo;

FUNCTION MergeSort(low, high: INTEGER; stream: ArrayToSort
                   RETURNS ArrayToSort, INTEGER);
VAR first, second : ArrayToSort;
    maxFirst, maxSecond : INTEGER;
BEGIN
RETURNS
   IF (high - low) > 1 THEN
      first, maxFirst, second, maxSecond :=
         MergeSort(low, (low + (high-low)/2), stream),
         MergeSort(low + 1 + (high-low)/2, high, stream);
      RETURNS MergeTwo(first, second, maxFirst, maxSecond)
   ELSE
      RETURNS SortTwo(low, high, stream)
   ENDIF
ENDFUNCTION MergeSort;

FUNCTION Main(RETURNS ArrayToSort);
VAR
   a, r: ArrayToSort;
   i: INTEGER;
BEGIN
   r := NewSample(512);
   a, i := MergeSort(1,512,r);
   RETURNS a
ENDFUNCTION Main;

ENDMODULE MSort.
```

B.13. Newton-Approximation der Quadratwurzel

Die Newton-Approximation der Quadratwurzel ist ein sehr einfaches, sequentiell iteratives Programm. Es wurde in diese Sammlung aufgenommen, da man an diesem Beispiel den LOOP-Ausdruck und den Gebrauch von Schleifenvariablen in MFL sehr schön sehen kann.

```
MODULE Newton;

FUNCTION Sqrt(x, eps: REAL RETURNS REAL);

VAR sqrt: REAL;

BEGIN
RETURNS
   LOOP
      sqrt := x / 2.0;
   DO
      NEW sqrt := (x / sqrt + sqrt) / 2.0;
   EXIT (sqrt * sqrt - x) < eps;
      RETURNS sqrt
   ENDLOOP
ENDFUNCTION Sqrt;

FUNCTION Main(RETURNS REAL);

VAR sqrt: REAL;
    i, j, k: INTEGER;

BEGIN
RETURNS
   Sqrt(2.0, 0.00001)
ENDFUNCTION Main;

ENDMODULE Newton.
```

B.14. Paralleles Mischen (Perfect Shuffle)

Sortieren ist zweifellos eines der wichtigsten und bestanalysierten Probleme der Informatik. Es gibt auch eine Reihe von schönen, parallelen Sortieralgorithmen. Der gezeigte Algorithmus funktioniert nach der Methode des "bitonischen Mischens" auf einem "Perfect Shuffle"-Netzwerk. Im Prinzip ist dies ein Teil eines parallelen Mergesort-Algorithmus. Eine schöne Einführung zum Thema findet man in [Sedgew88, p. 569ff.]

```
MODULE ParMerge;

TYPE LargeRange  = [1..512];
     ArrayToSort = ARRAY LargeRange OF INTEGER ENDARRAY;

FUNCTION Random(lastRandom: INTEGER RETURNS INTEGER);
BEGIN
   RETURNS ((lastRandom * 521) + 17) MOD 256
ENDFUNCTION Random;

FUNCTION NewSample(limit: INTEGER RETURNS ArrayToSort);
VAR a: ArrayToSort;
    i: LargeRange;
    lastRandom: INTEGER;
BEGIN
RETURNS
  FOR i := [1..limit] INIT
    lastRandom := 13;
  DO
    a[i] := lastRandom;
    lastRandom := Random(lastRandom);
  RETURNS a
  ENDFOR
ENDFUNCTION NewSample;

FUNCTION Merge(a: ArrayToSort RETURNS ArrayToSort);
VAR i, j: INTEGER;
    t, tt: ArrayToSort;
BEGIN
   RETURNS
      FOR i := [1..10] INIT
         t := a;
      DO
         NEW t :=
            FORALL j := [1..256] DO
               tt[2*j-1] := MIN(t[j], t[256 + j]);
               tt[2*j] := MAX(t[j], t[256 + j]);
            RETURNS
               tt
            ENDFORALL;
      RETURNS t
      ENDFOR
ENDFUNCTION Merge;

FUNCTION Main(RETURNS ArrayToSort);
VAR
   a, r: ArrayToSort;
   i: INTEGER;
BEGIN
   r := NewSample(512);
   a := Merge(r);
   RETURNS a
ENDFUNCTION Main;

ENDMODULE ParMerge.
```

Anhang C: Assembler-Code für Blockmove

In diesem Anhang wird der Code der Laufzeitfunktion "BlockMove" im High-Level-Assembler-Format [FogMil90] angegeben. Kommentare zum Gebrauch und zur Implementation dieser Funktion finden sich im Abschnitt 6.4 "Objekte mit Unterobjekten". In Abschnitt 6.5.1 "Experiment mit Blockmove".

Man beachte, wie der Algorithmus an Hand der Objektklasse des Quellen- und des Zielobjekts zur Laufzeit feststellt, ob der Blockmove parallel gemacht werden kann. Zudem ist interessant, wie die eigentlichen Kopierschleifen ausgerollt ("loop unrolling") sind, so dass möglichst viele LD-Instruktionen ausgeführtwerden können, bevor synchronisiert werden muss. Im Prinzip wird hier von Hand exakt dasselbe Prinzip angewendet wie vom Compiler bei der Sequentialisierung (vgl. Kapitel 7 "Sequentialisierung und Codegenerierung").

```
TYPE
   ParMoveParms =
      RECORD
         COUNT: INTEGER;
         FRAMEADDR: ANY;
         source, destination: ANY;
         srcIndex, destIndex: INTEGER;
      END;


BLOCK ParMove(p: ParMoveParms);
VAR t0, t1, t2, t3, t4, t5, t6, t7, t8,
    t9, t10, t11, t12, t13, t14, t15: ANY;
    source, destination: ANY;
    num [1], srcIndex, destIndex: INTEGER;
BEGIN
   srcIndex := p.srcIndex;
   destIndex := p.destIndex;
   source := p.source;
   destination := p.destination;
   srcIndex := srcIndex + num * 16;
   destIndex := destIndex + num * 16;
   t0 := LD(source, srcIndex);
   srcIndex := srcIndex + 1;
   t1 := LD(source, srcIndex);
   srcIndex := srcIndex + 1;
   t2 := LD(source, srcIndex);
   srcIndex := srcIndex + 1;
   t3 := LD(source, srcIndex);
   srcIndex := srcIndex + 1;
   t4 := LD(source, srcIndex);
   srcIndex := srcIndex + 1;
   t5 := LD(source, srcIndex);
   srcIndex := srcIndex + 1;
   t6 := LD(source, srcIndex);
   srcIndex := srcIndex + 1;
```

```
    t7 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t8 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t9 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t10 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t11 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t12 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t13 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t14 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t15 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t0 := STO(destination, destIndex, t0);
    destIndex := destIndex + 1;
    t1 := STO(destination, destIndex, t1);
    destIndex := destIndex + 1;
    t2 := STO(destination, destIndex, t2);
    destIndex := destIndex + 1;
    t3 := STO(destination, destIndex, t3);
    destIndex := destIndex + 1;
    t4 := STO(destination, destIndex, t4);
    destIndex := destIndex + 1;
    t5 := STO(destination, destIndex, t5);
    destIndex := destIndex + 1;
    t6 := STO(destination, destIndex, t6);
    destIndex := destIndex + 1;
    t7 := STO(destination, destIndex, t7);
    destIndex := destIndex + 1;
    t8 := STO(destination, destIndex, t8);
    destIndex := destIndex + 1;
    t9 := STO(destination, destIndex, t9);
    destIndex := destIndex + 1;
    t10 := STO(destination, destIndex, t10);
    destIndex := destIndex + 1;
    t11 := STO(destination, destIndex, t11);
    destIndex := destIndex + 1;
    t12 := STO(destination, destIndex, t12);
    destIndex := destIndex + 1;
    t13 := STO(destination, destIndex, t13);
    destIndex := destIndex + 1;
    t14 := STO(destination, destIndex, t14);
    destIndex := destIndex + 1;
    t15 := STO(destination, destIndex, t15);
    destIndex := destIndex + 1;
    WAIT6(t0, t1, t2, t3, t4, t5);
    WAIT6(t6, t7, t8, t9, t10, t11);
    WAIT4(t12, t13, t14, t15);
    SIGNAL(p);
END ParMove;
```

```
BLOCK BlockMove(source: ANY; srcIndex: INTEGER;
                destination: ANY; destIndex, length: INTEGER);
VAR t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, t14, t15: ANY;
   end, nrOfCBs: INTEGER;
   p: ParMoveParms;
BEGIN
   IF (0 IN destination) & ((0 IN source) | (1 IN source)) THEN
      (* Parallel kopieren *)
      nrOfCBs := length / 16;
      IF nrOfCBs > PROCCNT THEN
         p := NEWDUP(7);
      ELSE
         p := NEWOBJ(7);
      END;
      p.COUNT := 0;
      p.source := source;
      p.destination := destination;
      p.srcIndex := srcIndex;
      p.destIndex := destIndex;
      PCALL (MODREF, ParMove, p, nrOfCBs);
      p := SWAIT (p, nrOfCBs);
      DISPOSE (p, 1);
   ELSE
      (* Seriell kopieren *)
      end := srcIndex + length - 15;
      WHILE srcIndex < end DO
         t0 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t1 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t2 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t3 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t4 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t5 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t6 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t7 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t8 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t9 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t10 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t11 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t12 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t13 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t14 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t15 := LD(source, srcIndex);
         srcIndex := srcIndex + 1;
         t0 := STO(destination, destIndex, t0);
         destIndex := destIndex + 1;
         t1 := STO(destination, destIndex, t1);
         destIndex := destIndex + 1;
```

```
        t2 := STO(destination, destIndex, t2);
        destIndex := destIndex + 1;
        t3 := STO(destination, destIndex, t3);
        destIndex := destIndex + 1;
        t4 := STO(destination, destIndex, t4);
        destIndex := destIndex + 1;
        t5 := STO(destination, destIndex, t5);
        destIndex := destIndex + 1;
        t6 := STO(destination, destIndex, t6);
        destIndex := destIndex + 1;
        t7 := STO(destination, destIndex, t7);
        destIndex := destIndex + 1;
        t8 := STO(destination, destIndex, t8);
        destIndex := destIndex + 1;
        t9 := STO(destination, destIndex, t9);
        destIndex := destIndex + 1;
        t10 := STO(destination, destIndex, t10);
        destIndex := destIndex + 1;
        t11 := STO(destination, destIndex, t11);
        destIndex := destIndex + 1;
        t12 := STO(destination, destIndex, t12);
        destIndex := destIndex + 1;
        t13 := STO(destination, destIndex, t13);
        destIndex := destIndex + 1;
        t14 := STO(destination, destIndex, t14);
        destIndex := destIndex + 1;
        destination := STO(destination, destIndex, t15);
        destIndex := destIndex + 1;
        WAIT6(t0, t1, t2, t3, t4, t5);
        WAIT6(t6, t7, t8, t9, t10, t11);
        WAIT3(t12, t13, t14);
    END;
END;
length := MOD(length, 16);
IF length >= 8 THEN
    t0 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t1 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t2 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t3 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t4 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t5 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t6 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    t7 := LD(source, srcIndex);
    srcIndex := srcIndex + 1;
    length := length - 8;
    t0 := STO(destination, destIndex, t0);
    destIndex := destIndex + 1;
    t1 := STO(destination, destIndex, t1);
    destIndex := destIndex + 1;
    t2 := STO(destination, destIndex, t2);
    destIndex := destIndex + 1;
    t3 := STO(destination, destIndex, t3);
    destIndex := destIndex + 1;
    t4 := STO(destination, destIndex, t4);
    destIndex := destIndex + 1;
```

```
      t5 := STO(destination, destIndex, t5);
      destIndex := destIndex + 1;
      t6 := STO(destination, destIndex, t6);
      destIndex := destIndex + 1;
      destination := STO(destination, destIndex, t7);
      destIndex := destIndex + 1;
      WAIT6(t0, t1, t2, t3, t4, t5);
      WAIT1(t6);
   END;
   IF length >= 4 THEN
      t0 := LD(source, srcIndex);
      srcIndex := srcIndex + 1;
      t1 := LD(source, srcIndex);
      srcIndex := srcIndex + 1;
      t2 := LD(source, srcIndex);
      srcIndex := srcIndex + 1;
      t3 := LD(source, srcIndex);
      srcIndex := srcIndex + 1;
      length := length - 4;
      t0 := STO(destination, destIndex, t0);
      destIndex := destIndex + 1;
      t1 := STO(destination, destIndex, t1);
      destIndex := destIndex + 1;
      t2 := STO(destination, destIndex, t2);
      destIndex := destIndex + 1;
      destination := STO(destination, destIndex, t3);
      destIndex := destIndex + 1;
      WAIT3(t0, t1, t2);
   END;
   IF length >= 2 THEN
      t0 := LD(source, srcIndex);
      srcIndex := srcIndex + 1;
      t1 := LD(source, srcIndex);
      srcIndex := srcIndex + 1;
      length := length - 2;
      t0 := STO(destination, destIndex, t0);
      destIndex := destIndex + 1;
      destination := STO(destination, destIndex, t1);
      destIndex := destIndex + 1;
      WAIT1(t0);
   END;
   IF length = 1 THEN
      t0 := LD(source, srcIndex);
      destination := STO(destination, destIndex, t0);
   END;
   RETURN(destination);
END BlockMove;
```

Leer - Vide - Empty

Literatur

[Ackerm82] Ackerman, W. B. "Data Flow Languages" IEEE Computer (2-1982): 15ff.

[Ackerm84] Ackerman, W. B. "Efficient Implementation of Applicative Languages" PhD Thesis, MIT, Cambridge, MA, 1984.

[AdChDi74] Adam , T. L., K. M. Chandy and J. R. Dickson. "A Comparison of List Schedules for Parallel Processing Systems." CACM 12, 12 (12-1974): 685ff.

[AgeArv82] Agerwala, T. and Arvind. "Data Flow Systems." IEEE Computer (2-1982): 10ff.

[Agha89] Agha G. "Supporting Multiparadigm Programming on Actor Architectures." in Odijk E., Rem M. and Syre J.C., PARLE '89 Parallel Architectures and Languages Europe, Eindhoven, 6-1989. 1989.

[AhHoUl83] Aho, A. V., Hopcroft, J. E. und Ullman, J. D. "Data Structures and Algorithms", Addison-Wesley, 1983.

[AhoJoh76] Aho A. V. and Johnson S. C. "Optimal Code Generation for Expression Trees." JACM 23, 3 (7-1976):

[AhJoUl77] Aho A.V., Johnson S. C. and Ullman J. D. "Code Generation for Expressions with Common Subexpressions." JACM 24, 1 (1-1977):

[AhSeUl86] Aho, A. V., Sethi, R., Ullman, J. D. Compilers Principles, Techniques and Tools. Addison-Wesley, 1986.

[Amdahl67] Amdahl, G. M. "Validity of the single processor approach to achievinglarge scale computing capabilities", Proc. AFIPS, vol. 30, 1967, p. 483ff.

[Apple89a] Apple Computer Inc., "MPW: MacIntosh Programmer's Workshop Development Environment", Version 3, Apple Computer Inc, Cupertino, CA.

[Apple89b] Apple Computer Inc., "MacApp", Version 2, Apple Computer Inc, Cupertino, CA.

[ArCuMa88] Arvind, D. E. Culler and G. K. Maa. "Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs." in Proceedings Supercomputing '88, Orlando, FL, 11-1988.

[Arvind90] Arvind, Kursunterlagen, Summer School on Parallel Processing, Ascona, 1990.

[ArvCul86] Arvind and Culler D. E. "Dataflow Architectures." Ann. Rev. Comput. Sci 1 (1986): 225-53.

[ArCuEk88] Arvind, D. E. Culler and Ekanadham K. "The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures." in Proceedings of CONPAR, Manchester, GB, 9-1988.

[ArvEka88] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines, MIT, Laboratory of Computer Science, Cambridge, MA, 2-1988.

[ArHeNi88] Arvind, S. K. Heller and R. S. Nikhil. "Programming Generality and Parallel Computers." in Fourth International Symposium on Biological and Artificial Intelligence Systems, Trento, I, 9-1988.

[ArvIan87] Arvind and R. A. Ianucci. "Two Fundamental Issues in Multiprocessing." in Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, D, 6-1987.

[ArvNik89] Arvind and R. S. Nikhil. A Dataflow Approach to General-purpose Computing, MIT, Laboratory of Computer Science, Cambridge, MA, 7-7-1989.

[ArvNik89] Arvind and R. S. Nikhil. "Executing a Program on the MIT Tagged Token Dataflow Architecture", MIT, Laboratory of Computer Science, Cambridge, MA, 6-20-88.

[ArNiPi86] Arvind, R. S. Nikhil and K. K. Pingali. "I-Structures: Data Structures for Parallel Computing." in Workshop on Graph Reduction, Los Alamos, NM, 10-1-1986.

[Babb84] Babb II, R. G. "Parallel Processing with Large Grain Data Flow Techniques." IEEE Computer (7-1984):

[Backus78] Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." Communications of the ACM 21, 8 (8-1978): 613-641.

[Banerj89] Banerjee, U. "A Theory of loop permutations." in 2nd Workshop on Programming Languages and Compilers for Parallel Computing, Urbana, 8-1989.

[Banerj90] Banerjee, U. "Unimodular Transformations of Double Loops." in 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, 8-1990.

[Böhm83] Böhm C.P.W. "Dataflow Computation", Centrum voor Wiskunde en Informatica, 1983.

[Boldin91] Boldini, R "Optimierender MFL-Codegenerator", Diplomarbeit, Institut für Technische Informatik und Kommunikationsnetze, DA 91/3, 1991, Zürich.

[Boute86] Boute, R. T., Persönliche Mitteilung, 1986.

[BruSet76] Bruno, J. und Sethi, R. "Code Generation for a One-register Machine." JACM 23, 3 (3-1976):

[BBBBFH82] Bührer, R. E., H. Brundiers, H. Benz, B. Bron, H. Friess, W. Hälg, H. J. Halin, A. Isacson and M. Tadjan. "The ETH-Multiprocessor EMPRESS: A Dynymically Configurable MIMD System." IEEE Transactions on Computers C-31, 11 (11-1982): 1035-1044.

[BühEka87] Bührer, R. and K. Ekanadham. "Incorporating Dataflow Ideas into von Neumann Processors for Parallel Execution." IEEE Transactions on Computers C-36, 12 (1987-12): 1515-1522.

[Cann89] Cann D. C. "Compilation Techniques for High Performance Applicative Computation", Ph. D. Thesis, University of Colorado, 1989.

[CanFeo90] Cann D. C. and J. Feo, "SISAL versus FORTRAN: A comparision using the Livermore Loops", Lawrence Livermore National Laboratory Report, 1990.

[Chaitin82] Chaitin, G. J. "Register Allocation & Spilling via Graph Coloring", in Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, Boston, Juni, 1982.

[Coffma76] Coffman, E. G. Computer and Job-Shop Scheduling Theory, Wiley, 1976.

[CofDen73] Coffman, E. G. and Denning, P. J. Operating Systems Theory, Prentice-Hall, 1973.

[Cohen81] Cohen, J. "Garbage Collection of Linked Data Structures", ACM Computing Surveys, Vol. 13, No. 3, September 1981.

[CoDGPS78] Comte, D., G. Durrieu, O. Gelly, A. Plas and J. C. Syre. "Parallelism, Control and Synchronization Expression in a Single Assignment Language." SIGPLAN Notices 13, 1 (1978-1): 25-33.

[Culler89] Culler, D. E. Managing Parallelism and Resources in Scientific Dataflow Programs, MIT, Laboratory of Computer Science, Cambridge, MA, 6-1989.

[CulArv88] Culler, D. E. and Arvind. "Resource Requirements of Dataflow Programs." in The 15th International Symposium on Computer Architecture, Honolulu, HI, 6-1988. 5-1988.

[DavKel82] Davis A. L. and Keller R. M. "Data Flow Program Graphs." IEEE Computer (2-1982):

[Dennis75] Dennis, J. B. First Version of a Dataflow Procedure Language, MIT, Laboratory of Computer Science, Cambridge, MA, 5-1975.

[EngLäu88] E. Engeler und P. Läuchli, "Berechnungstheorie für Informatiker", Teubner-Verlag, Stuttgart, 1988

[FisERN84] Fisher, J. A. et al. "Parallel Processing: A Smart Compiler and a Dumb Machine", Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, Juni 1984.

[Flynn72] Flynn, M. J. "Some Computer Organizations and their Effectiveness", IEEE Transactions on Computers **C-21**, 1972

[FogMil90] Foglia, A. und Milesi, G. "High-Level-Assembler", Diplomarbeit, Institut für Technische Informatik und Kommunikationsnetze, DA 90/6, 1990, Zürich.

[GaPaKK82] Gajski D. D., Padua D. A., Kuck, D. J. und Kuhn R. H. "A Second Opinion on Data Flow Machines and Languages", IEEE Computer, Februar 1982

[Garner88] Garner R. et al. "Scalable Processor Architecture (SPARC)", IEEE COMPCON, San Francisco, März 1988.

[GauErc84] Gaudiot, J. L. und Ercegovac, M. D. "Performance Analysis of a Dataflow Computer with variable Resolution Actors", Proceedings of the 1984 IEEE Conference on Distributed Computer Systems, San Francisco, Mai 1984.

[GibMuc86] Gibbons, P. B. und Muchnik, S. S. "Efficient Instruction Scheduling for a Pipelined Architecture", Proceedings of the SIGPLAN'86 Symposium on Compiler Construction, Palo Alto, Juni 1986.

[GhSaHe88] Gharacharloo, K, Sarkar, V und Hennessy, J "A Simple and Efficient Implementarion Approach for Single Assignment Languages", in Proceedings of the 1988 ACM Conference on Lisp annctional Programming, Snowbird, UT, Juli 1988.

[GirPol88] Girkar, M. und Polychronopoulos C. "Partitioning Programs for Parallel Execution" in Proceedings of the 1988 International Conference on Supercomputing, St. Malo, ACM Press, 1988.

[GraKoc92] Graham, T. C. N. und Kock, G. "Domesticationg Imperative Constructs for a Functional World" in Structured Programmin (1992):**13**, Springer-Verlag, 1992.

[Gries81] Gries, D. "The Science of Programming" Springer, 1981.

[GolRob83] Goldberg, A. und Robson, D. "Smalltalk 80: The Language and its Implementation", Addison-Wesley, 1983

[GooHsu88] Goodman, J. R. und Hsu, W.-C. "Code Generation and Scheduling in Large Basic Blocks" in Proceedings of the 1988 International Conference on Supercomputing, St. Malo, ACM Press, 1988

[GuKiWa85] Gurd, J. R., Kirkham, C. C., Watson, I. 1985. The Manchester dataflow prototype computer, CACM 28(1):34-52.

[HenGro83] Hennessy, J. und Gross, T. "Postpass Code Optimization of Pipeline Constraints" in ACM Transactions on Programming Languages and Systems, 5(3), Juli 1983.

[Hirsch78] Hirschberg, D. S. "Fast Parallel Sorting Algorithms." Communications of the ACM 21, 8 (8-1978): 657-661.

[HiNSSY87] Hiraki, K., Nishida, K., Sekiguchi, S., Shimada, T. and Yuba, T. "The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems" in Journal of Information Processing 10, 4 (1987).

[Hudak86] Hudak, P. "A Semantic Model of Reference Counting and its Abstraction", Proceedings of the ACM Conference on LISP and Functional Programming, MIT, Cambridge, MA.

[HudBlo85] Hudak, P. and Bloss, A. "The Aggregate Update Problem in Functional Programming Systems" Proceedings of the 12th Annual ACM Conference on Principles of Programming Languages, 1985.

[HudGol85] Hudak, P. and B. Goldberg. "Serial Combinators: "Optimal" Grains of Parallelism." in Functional Programming Languages and Computer Architectures, 9-1985.

[HwaBri84] Hwang, K., Briggs, F. A. Computer Architecture and Parallel Processing. McGraw Hill, 1984.

[Ianucc88] Iannucci, R. A. "Toward a Dataflow / von Neumann Architecture." in The 15th Annual International Symposium on Computer Architecture, Honolulu, HI, 6-2-1988. 5-1988.

[Inmos91] Inmos, "The T9000 Transputer Products Overview Manual", First Edition, 1991.

[InStXi86] Indurkhya, B., Stone, H. S. und Xi-Cheng, L. "Optimal Partioning of Randomly Generated Distributed Programs", IEEE Transactions on Software Engineering, SE-12(3), März 1986

[Kahn74] Kahn, G., "The Semantics of a Simple Language for Parallel Processing", Proceedings of IFIP Congress, 1974

[Kelly89] Kelly, P. Functional Programming for Loosely-Coupled Microprocessors. London: Pitman Publishing, 1989.

[KUPWGD83] Kirchgässner W, Uhl J., Persch G., Winterstein G., Goos G., Dausmann M. and Drossopoulou S. An Optimizing Ada Compiler, Karlsruhe, 2-7-83.

[Krishn90] Krishnamurthy, S. M. "Static Scheduling of Multicycle Operations for a RISC Processor.", Bericht.

[KuKPLW81] Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe. "Dependence Graphs and Compiler Optimizations." in Proceedings of the 8th ACM Symposium on Principles of Programming Languages, 1-1981.

[Läuchl88] Läuchli, P. Skript: Algorithmische Graphentheorie, ETH Zürich, 1988.

[Lips88] Lips, H. "Redesign des MFL-Compilers", Diplomarbeit, Institut für Technische Informatik und Kommunikationsnetze, DA 88/12, 1988, Zürich.

[Marchi90] de Marchi, P. "Codeblock-Partitionierung für MFL", Diplomarbeit, Institut für Technische Informatik und Kommunikationsnetze, DA 90/23, 1990, Zürich.

[MarMur92a] Marti, R. und Murer, S. "FOOL: Sprachbeschreibung", TIK-Bericht, 92-x, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, 1992.

[MarMur92b] Marti, R. and Murer, T. "GIPSY: An Experimental Programming System Generator", TIK-Bericht, 92-x, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, 1992.

[McCGil89] McCreary, C. and H. Gill. "Automatic Determination of Grain Size for efficient Parallel processing." Communications of the ACM 32, 9 (9-1989): 1073-1078.

[McGraw82] McGraw, J. R. "The VAL Language: Description and Anylysis", ACM TOPLAS, 4(1), Januar 1982.

[McGraw85] McGraw, J. R. et al."SISAL: Streams and Iterations in a Single Assignment Language", Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, März 1985.

[McMaho86] McMahon, F. H., L. L. N. L. FORTRAN Kernels: MFLOPS, Lawrence Livermore National Laboratory, 1986.

[Maquel90] Maquelin, O. "ADAM: a Coarse-Grain Dataflow Architecture that Adresses the Load Balancing and Throttling Problems" in Proceedings of CONPAR 90-VAPP IV, Zürich, September 1990, Springer LNCS 457.

[Maquel92] Maquelin, O. "Load Balancing and Resource Management in the ADAM-Machine", Proceedings of the Dataflow Workshop, 1992.

[Mitrov90] Mitrovic, S. et al. "A Distributed Memory Multiprocessor Based on Dataflow Synchronization" in Proceedings of International Phoenix Conference on Computers and Communication, March 1990

[Mitrov92] Mitrovic, S. "Programming the ADAM Architecture with SISAL", Proceedings of the Dataflow Workshop, 1992.

[MöTeGr89] Mössenböck, H., Templ, J. und Griesemer, R. "Object Oberon - An object-oriented extension of Oberon", Informatik Berichte 109, Departement Informatik, ETH Zürich, 1989.

[Motoro88] Motorola, MC88100 RISC Microprocessor User's Manual, 1988

[Murer88] Murer, S. "Strategien zur Codegenerierung für parallele Ausdrücke", Diplomarbeit, Institut für Technische Informatik und Kommunikationsnetze, DA 88/4, 1988, Zürich.

[Murer90] Murer, S. "A Latency Tolerant Code Generation Algorithm for a Coarse Grain Dataflow Machine" in Proceedings of CONPAR 90-VAPP IV, Zürich, September 1990, Springer LNCS 457.

[Murer91] Murer, T. "Hypertext-Funktionen für den Strukturediteor von FOOL", Semesterarbeit, Institut für Technische Informatik und Kommunikationsnetze, SA 91/4, 1991, Zürich.

[Murer92] Murer, T. "Umgebung zur Spezifikation inkrementeller Compiler-Frontends", Diplomarbeit, Institut für Technische Informatik und Kommunikationsnetze, DA 92/6, 1992, Zürich.

[MurFär92] Murer, S. und Färber, Ph. "A Scalable Distributed Shared Memory System" in Proceedings of CONPAR 92-VAPP V, Lyon, September 1992.

[MurMar92] Murer, S. und Marti, R. "The FOOL Programming Language: Integrating Single-Assignment and Object-Oriented Paradigms", Proceedings of European Workshop on Parallel Computing 92, IOS Press, Amsterdam, 1992.

[Nikhil88] Nikhil, R. S. "ID Reference Manual", MIT, Laboratory of Computer Science, Cambridge, MA, 8-29-1988.

[NikArv89] Nikhil, R. S. and Arvind. "Can dataflow subsume von Neumann computing?" in The 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, 6-1989.

[NiPaAr91] Nikhil, R. S., G. M. Papadopoulos and Arvind. *T: a Killer Micro for the Brave New World, MIT LCS, Cambridge, MA, 1991-7.

[P189] P1, Gesellschaft für Informatik mbH, "Modula-2, Objektorientierter Compiler für Apple MacIntosh unter MPW", München, 1989.

[PadWol86] Padua, D. A. und Wolfe, M. J. "Advanced Compiler Optimizations for Supercomputers", Communication of the ACM 29(12), Dezember 1986.

[Parsyt92] Parsytec AG, "Jenseits des Supercomputers - Parsytec GC", Systembeschreibung, Parsytec AG, Aachen, 1992

[PetSil85] Peterson, J. L., Silberschatz, A. Operating System Concepts. Addison-Wesley, 1985.

[Plas76] Plas, A. et al. "LAU System Architecture: A Parallel Data-Driven Processor based on Single Assignment." in P. H. Enslow, Proceedings of the 1976 International Conference on Parallel Processing, Detroit, MI, 1976-8-24.

[RaChGo72] Ramamoorthy, C. V., K. M. Chandy and M. J. Gonzalez. "Optimal Scheduling Strategies in a Multiprocessor System." IEEE Transactions on Computers C-21, 2 (1972-2): 137-146.

[Reisig85] Reisig, W. "Petri Nets, An Introduction", Springer-Verlag, Heidelberg, 1985.

[Ruggie87] Ruggiero, C. A. "Throttle Mechanisms for the Manchester Dataflow Machine", Ph. D. Thesis, Department of Computer Science, University of Manchester, 1987.

[SaYHKY89] Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y., Yuba, T. "An Architecture of a Dataflow Single Chip Processor" in Proceedings of International Symposium on Computer Architecture, 1989.

[SaSkMi88] Sarkar, V., Skedzielewski, S. und Miller, P. "An Automatically Partitioning Compiler for SISAL" in Proceedings of CONPAR'88, Cambridge University Press, Cambridge, UK, 1988.

[SarHen86] Sarkar, V. und Hennessy J. "Partitioning Parallel Programs for Macro-Dataflow" in Proceedings of the 1986 ACM Conference on Lisp annctional Programming, Cambridge, MA, August 1986.

[Sarkar89] Sarkar, V. Partitioning and Scheduling Parallel Programs for Multiprocessors. London: Pitman Publishers, 1989.

[Sarkar90] Sarkar, V. "Instruction Reordering fo Fork-Join Parallelism." in Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, Whitplains NY, Juni 1990.

[Sedgew88] Sedgewick, R. "Algorithms", Addison-Wesley, 1988.

[SkeGla84] Skedzielewski S. and Glauert J. IF1, "An Intermediate Form for Applicative Languages", Lawrence Livermore Laboratory, 6-18-1984.

[SkeWel85] Skedzielewski S. and Welcome, M. L. IF1, "Dataflow Graph Optimization in IF1", in Functional Programming Languages and Computer Architecture, Springer LNCS 201, 1985.

[Sommer85] Sommerville, I. Software Engineering. Addison-Wesley, 1985.

[Stetter76] Stetter, H. J. Numerik für Informatiker. München: Oldenbourg, 1976.

[Stoy77] Stoy, J. E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Cambridge, MA: MIT Press, 1977.

[Strous86] Stroustrup, B. "The C++ Programming Language", Addison-Wesley, 1986

[Tarjan72] Tarjan, R. "Depth-First Search und Linear Graph Algorithms" in SIAM Journal on Computing 1(2), Juni 1972

[TeDuPe89] Terrano, A. E., S. M. Dunn and J. E. Peters. "Using an Architectural knowledge base to Generate Code for Parallel Computers." CACM 32, 9 (9-1989): 1065-1072.

[Tutte84] Tutte, W. T. Graph Theory, Addison-Wesley, 1984

[Traub91] Traub, K. R. "Multi-thread Code Generation for Dataflow Architectures form Non-Strict Programs", Motorola Technical Report MCRC-TR-14, Juni 1991, Cambridge, MA

[Ware72] Ware W. H., "The ultimate Computer" in IEEE Spectrum, März, 1972

[Wirth85] Wirth, N. "Programming in Modula-2", Springer, 1985

[Wirth86] Wirth, N. "Compilerbau. Stuttgart: Teubner", 1986

[Wüst90] Wüst, T. "Codegenerierung für MFL", Diplomarbeit, Institut für Technische Informatik und Kommunikationsnetze, DA 90/3, 1990, Zürich.

[Wüthri88] Wüthrich, H. "MFL-Compiler", Diplomarbeit, Institut für Technische Informatik und Kommunikationsnetze, DA 88/6, 1988, Zürich.

[Wytten90] Wyttenbach, J. Design of a Variable Grain Dataflow Machine and its Relation to a New Approach fo System Specification, Diss. ETH Nr. 9326, Zürich, 1990

[Wolfe90] Wolfe, M. "Automatic Parallelism Detection: What went wrong?", Notizen zu Vortrag am LLNL

Lebenslauf

Stephan Murer,
geboren am 26.2.63 in Zürich,
von Beckenried (NW) und Zürich

1970-1976	Primarschule in Zürich
1976-1982	Gymnasium an der Kantonsschule Wiedikon, Zürich, Matura Typus B
1982-1983	Diverse Stellen, Militärdienst
1983-1988	Studium an der Abteilung IIIC der ETH Zürich
1988	Diplom als Dipl. Informatik-Ing. ETH
1988-1992	Assistent am Institut für Technische Informatik und Kommunikationsnetze bei Prof. Kündig