RZTU

Rechenzentrum der

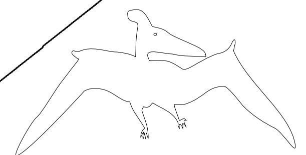
Technischen Universität Hamburg-Harburg

Programmentwicklung auf der Convex

Dokumentationsschlüssel:

Version M

2. Auflage, Oktober 1993



Herausgeber	:	Technische Universität Hamburg-Harburg – Rechenzentrum – Denickestr. 17 21073 Hamburg
Autor	:	Peter Junglas, Tel.: 3193

Inhaltsverzeichnis

1 Einleitung 2

2 Grundlegende Schritte bei der Programm-Entwicklung 3

2.1 Das Beispiel-Programm linalg 3
2.2 Compilieren und Linken 3
2.3 Das make-Programm 4
2.4 Verwalten verschiedener Versionen - Das RCS-System 6
2.5 Debuggen von linalg 8
2.6 Automatische Optimierung durch den Compiler 11
2.7 Application Compiler 12
2.8 CXpa 14
2.9 Einfache Handoptimierungen 17
2.9.1 Speicherbank-Konflikte 17
2.9.2 Vektorlänge 17
2.9.3 Bibliotheken 18
2.9.4 Algorithmen 19

3 Die Tools im einzelnen 20

3.1 CXdb 20
3.1.1 Die Oberfläche des CXdb 20
3.1.2 Schrittweiser Programmablauf 21
3.1.3 Betrachten und Verändern von Variablen 25
3.1.4 Haltepunkte 27
3.1.5 Die CXdb-Umgebung 30
3.1.6 Debuggen von optimiertem Code 33
3.2 Der Convex-Performance-Analyzer CXpa 37
3.2.1 Die Oberfläche des CXpa 37
3.2.2 Erzeugung eines Analyse-Files 37
3.2.3 Auswertung eines Analyse-Files 40
3.2.4 Die CXpa-Umgebung 44
3.3 Application Compiler 46
3.3.1 Arbeitsweise des Application-Compilers 46
3.3.2 Meldungen des Application Compilers 47
3.3.3 Das "build"-File 50

3.3.4 Benutzen von Bibliotheken . . . 51
3.3.5 Manuelle Kontrollmöglichkeiten des Application-Compilers . . . 52

4 Vektorisieren und Parallelisieren **55**
4.1 Architektur der Convex C3800 . . . 55
4.2 Automatische Vektorisierung . . . 58
4.2.1 Code-Umstellungen durch den Compiler . . . 58
4.2.2 Hindernisse für die Vektorisierung . . . 62
4.2.3 Der Optimierungs-Bericht . . . 64
4.3 Manuelle Verbesserung der Vektorisierung . . . 65
4.3.1 Vermeiden von Aliassen . . . 65
4.3.2 Vereinfachung von Schleifen . . . 67
4.3.3 Compiler-Direktiven . . . 68
4.3.4 Optimierung von Speicherzugriffen . . . 69
4.3.5 Benutzung von Bibliotheken . . . 70
4.4 Parallelisierung . . . 72
4.4.1 Automatische Parallelisierung . . . 72
4.4.2 Debuggen parallelisierter Programme . . . 73
4.4.3 Profilen parallelisierter Programme . . . 74
4.4.4 Compiler-Direktiven zur Parallelisierung . . . 77

5 Fallstudien **79**
5.1 POISSON - zweidimensionale Potentialprobleme in FORTRAN . . . 79
5.2 NKOERPER – Bahnkurven von Massepunkten unter Gravitationskraft in C . . 88

1 Einleitung

Die hohe Rechenleistung der Convex-Rechner bei technisch-naturwissenschaftlichen Anwendungen ist i. W. auf die Vektorrechner-Architektur zurückzuführen, die Vektor- und Matrizenberechnungen besonders unterstützt. Um die theoretisch mögliche Rechenleistung annähernd zu erreichen, muß die entsprechende Vektor-Hardware auch möglichst intensiv genutzt werden. Dazu dienen zunächst die Compiler, die Sprachkonstruktionen, die sich zur Vektorisierung eignen, automatisch erkennen und ggf. durch Umstellungen des Programmcodes möglichst effektiv in Vektorbefehle umsetzen. Zusätzlich zu solchen Optimierungsmöglichkeiten der Standard-Compiler führt der sogenannte "Application Compiler" von Convex noch wesentlich umfangreichere Analysen durch und erzielt dadurch eine noch bessere Anpassung eines Programms an die Rechner-Architektur. Das Ergebnis der automatischen Optimierung kann man durch geeigneten Programmierstil verbessern, indem man von vornherein bestimmte Konstruktionen vermeidet, die die Vektorisierung behindern können. Außerdem gibt es Hilfsmittel, um das erzielte Ergebnis zu überprüfen und die kritischen Bereiche zu analysieren, damit gezielt weitere Verbesserungen angebracht werden können.

Neben der Vorstellung der grundlegenden Hilfsmittel zur Programmentwicklung auf der Convex sollen in diesem Papier die Handhabung der relevanten Werkzeuge und die Vorgehensweise bei der manuellen Anpassung der Programme vermittelt werden, mit dem Ziel, mit selbstentwickelten Programmen eine möglichst hohe Ausnutzung der Rechnerleistung erreichen zu können. Dabei wird auch auf notwendige Hintergrundinformationen zur Rechner-Architektur eingegangen.

Die Darstellungsweise soll einerseits zur möglichst direkten Umsetzung auf eigene Programme anregen, andererseits aber auch genügend ausführlich sein, um das Nachblättern in den Referenz-Handbüchern in den meisten Fällen überflüssig zu machen.[1] Zu diesem Zweck werden in einem einführenden Abschnitt die wesentlichen Methoden und Hilfsmittel vorgestellt und an einem einfachen Beispielprogramm ausprobiert. Dann werden die weitergehenden Möglichkeiten der umfangreicheren Tools – der Debugger CXdb, der Profiler CXpa und der Application-Compiler-APC – besprochen.[2] Schließlich wird ausführlich auf Methoden zur Verbesserung der Vektorisierung und Parallelisierung eingegangen und ihre Anwendung an zwei Beispielprogrammen ausprobiert.
Natürlich mußte aus der Vielzahl der Standard-UNIX-Tools zur Programmentwicklung ausgewählt werden, wobei einige sehr spezielle Kommandos - etwa fsplit oder pmd - weggefallen sind. Gab es zur Lösung eines Problems mehrere Möglichkeiten, wurde das Programm mit den umfassenderen Möglichkeiten ausgewählt.[3] Aber nicht nur deswegen werden der alte Convex-Debugger csd, ein Derivat des weitverbreiteten dbx, und die Standard-Profiler der prof-Familie nicht behandelt. Sie werden nämlich, da durch CXdb und CXpa vollständig ersetzt, in zukünftigen Systemversionen wegfallen. Eine rechtzeitige Gewöhnung an die neuen Tools ist somit unerläßlich.

---

[1] Die Handbücher zu Compilern und Tools sind in der Bibliothek vorhanden.
[2] in den folgenden Versionen: CXdb 2.0, CXpa 1.3, APC 1.1
[3] Dabei kann man über die Wahl von RCS statt SCCS durchaus streiten.

2 Grundlegende Schritte bei der Programm-Entwicklung

2.1 Das Beispiel-Programm linalg

Das Programm LINALG [4] dient im ersten Kapitel als Beispiel für die verschiedenen Tools und Methoden. Es löst ein lineares Gleichungssystem und berechnet die Inverse der Koeffizienten-Matrix auf zwei verschiedene Weisen, nämlich durch Gauss-Jacobi-Elimination und mittels LU-Zerlegung. Das Hauptprogramm in main.f hat folgende Struktur:
Zunächst wird die Anzahl N der Gleichungen vorgegeben und mit GETMAT eine Koeffizienten-Matrix A mit zufälligen Koeffizienten gefüllt und mit GETVEC ein Zufalls-Vektor R für die rechten Seiten der Gleichungen erzeugt. Da die i.f. benutzten Verfahren die Koeffizienten-Matrix A und den Vektor R überschreiben, werden mit SETMAT bzw. SETVEC Arbeitskopien B und X angelegt. Nun wird mit der Routine GAUSSJ die Inverse bestimmt und die Gleichungen gelöst.[5] Die Ergebnisse stehen dann in B und X. Anschließend werden sie getestet: TESTMT untersucht, ob $A * B = 1$, und gibt eine entsprechende Meldung aus, ERNORM bestimmt die Größe des Fehlervektors $A * X - R$. Anschließend werden B und X wieder auf die Ausgangswerte gesetzt. Die Routine LUDCMP zerlegt B in $L * U$, wobei in INDEX eventuelle Zeilenvertauschungen angezeigt werden, in LUBKSB wird das Gleichungssystem gelöst, die Lösung steht in X. Die Routine LUINV bestimmt die Inverse BINV von $B = L * U$ durch sukzessives Aufrufen von LUBKSB für die Spalten der Einheits-Matrix als rechten Seiten. Schließlich werden mit TESTMT und ERNORM die Ergebnisse überprüft.
Matrix-Vektor- und Matrix-Matrix-Operationen, wie sie in LINALG durchgeführt werden, sind Bestandteil vieler numerischer Verfahren. Sie sind geradezu Grundmodelle für die Operationen, für die Vektorrechner konstruiert wurden. Die Verfahren, die wir zur Beschleunigung von LINALG kennenlernen werden, lassen sich daher auf viele Programme für Vektorrechner übertragen. Dabei kommt es uns natürlich mehr auf die Auswirkungen auf einzelne Routinen an als darauf, das Programm LINALG insgesamt zu beschleunigen.

2.2 Compilieren und Linken

Der Convex-Fortran-Compiler fc ist ein FORTRAN77-Compiler mit Convex-Erweiterungen. Er kann mit Optionen aber auch dazu gebracht werden, die Erweiterungen anderer Hersteller zu verstehen, nämlich von Cray (-cfc), Sun (-sfc) und VAX Fortran (-vfc). Außerdem sind mit der Option -f90 einige Erweiterungen von Fortran-90 (insbesondere die Array-Notation) verfügbar. Ausführliche Informationen zu allen Convex-spezifischen Erweiterungen kann man dem Reference-Manual entnehmen.
Der Convex-C-Compiler cc ist ein ANSI-C-Compiler mit Convex-Erweiterungen, der auch die POSIX-Library-Aufrufe unterstützt. Mit Hilfe von Optionen kann man Erweiterungen ausschalten, um Portabilität der Programme zu gewahrleisten: -std (nur ANSI C und POSIX) oder -str (nur ANSI C). Außerdem kann man mit der Option -pcc auf einen C-Compiler zurückgreifen, der noch nicht ANSI-konform ist, falls es Probleme mit alten C-Programmen gibt.

---

[4] Die Sourcen zu diesem und allen weiteren Beispiel-Programmen befinden sich unter `/tuhh/info/tu_beispiele/convex_kurs`

[5] Diese Routine stammt, wie auch die folgenden zur LU-Zerlegung (LUDCMP und LUBKSB), aus dem Buch "Numerical Recipes" von Press, Flannery, Teukolsky und Vetterling.

Weitere wichtige Optionen der Compiler sind:

Option	Bedeutung
`-c`	Es werden nur Object-Files erzeugt, der Loader wird nicht aufgerufen.
`-o` *name*	Das ausführbare Programm enthält den Namen "name", sonst heißt es a.out (eigentlich Loader-Option).
`-O`*n*	n = 0,1,2,3 : verschiedene Optimierungsstufen, (s. Abschnitt 2.6) Default: keine Optimierung[6]
`-or` *opt*	opt = none,loop,array,all : bestimmt die Art der Informationen, die der Optimierer ab Option -O2 ausgibt (s. Abschnitt 2.6) Default: loop
`-tm C`*n*	n = 1,2,38: Es wird Code für die C1, C2 oder C3800 erzeugt. Default: Maschinentyp, auf der der Compiler läuft
`-cs`	Indizes von Feldern werden zur Laufzeit auf Grenzüberschreitung überprüft; nützlich zum Debuggen (nur beim fc)
`-cxdb`	Code für den CXdb-Debugger wird erzeugt
`-pa, -pab, -par`	Code für den Profiler CXpa wird erzeugt (s. Abschnitt 2.8)

Einige Optionen für spezielle Optimierungen werden durch den Application Compiler (s. Abschnitt 2.7) überflüßig. Ebenso werden Optionen für den csd-Debugger und die diversen prof-Profiler wegen der neuen Tools CXdb und CXpa i.a. nicht mehr benötigt.
Die folgenden Optionen werden an den Loader übergeben, der (außer bei -c) automatisch aufgerufen wird:

Option	Bedeutung
`-lx`	Die Bibliothek libx.a wird nach bisher undefinierten Namen durchsucht und entsprechender Code hinzugebunden. Die Reihenfolge mehrerer -l-Optionen ist i.a. wichtig.
`-L`*dir*	Libraries werden auch im angegeben Verzeichnis "dir" gesucht. Default: "-L/lib -L/usr/lib -L/usr/local/lib"
`-m`	Bleiben undefinierte Namen übrig, werden die Libraries nochmals durchsucht.

Über weitere Optionen informiert das Online-Handbuch ("man fc/cc").

2.3 Das make-Programm

Im folgenden werden wir immer wieder leichte Änderungen an unserem Programm vornehmen. Dabei müssen natürlich entsprechende Teile des Source-Codes neu übersetzt werden. Um uns alle damit verbundene Arbeit abzunehmen, gibt es das Programm make: make liest ein Beschreibungsfile Makefile oder makefile (andere Namen können mit der Option -f angegeben werden), in dem

[6] Auf der C3 ist als Default -O2 eingestellt.

beschrieben wird, welche Programme man erzeugen möchte, wie die zugehörigen Files voneinander abhängen und wie man aus ihnen das Programm erhält.
Ein Makefile beginnt meistens mit Definitionen von Variablen, z.B.:

```
FC = /usr/convex/fc
FFLAGS = -O2 -pa
INCLUDE =
```

Diese Variablen können dann an beliebiger Stelle im Makefile in der Form $(FC) benutzt werden. Man kann sie auch erst beim Aufruf von make definieren, z.B.:

```
make FC=/usr/local/gnu/fc linalg
```

Diese Definition überschreibt eine im Makefile vorhandene. Außerdem können Kommentarzeilen eingefügt werden, die mit # beginnen.
Die Beschreibung der Aktionen geschieht in Zeilen der Form

```
Zielname: Abhaengigkeiten
<TAB>Aktion1
.
.
<TAB>AktionN
```

wobei weitere Aktionszeilen folgen können. Zu beachten ist, daß Aktionszeilen mit einem <TAB> (Tabulator-Zeichen) beginnen müssen; es dürfen dort keine Blanks stehen. Z.B. geben die Zeilen

```
main.o: $(INCLUDE) main.f
        $(FC) $(FFLAGS) -c main.f
```

an, daß das "Ziel" (File) main.o neu erstellt werden muß, wenn sich das File main.f oder ein mögliches Include-File geändert haben. Die zweite Zeile gibt an, was in einem solchen Fall zu tun ist, nämlich neu kompilieren. Es ist auch möglich, daß das Ziel von Files abhängt, die quasi als Zwischenziele ebenfalls im Makefile angegeben sind. Diese werden dann zunächst überprüft und ggf. neu erstellt. Etwa im Beispiel

```
linalg: $(OBJECTS)
        $(FC) $(OBJECTS) $(LIBS) -o $@ $(LFLAGS)
```

enthalte die Variable OBJECTS die Liste aller Objekt-Dateien, aus denen das Programm linalg bestehen soll. Um linalg auf den neuesten Stand zu bringen, müssen also zuerst alle Object-Files, deren Source-Files sich geändert haben oder die überhaupt noch nicht vorhanden sind, neu erzeugt werden, wobei entsprechende Makefile-Einträge wie oben für main.o verwendet werden. Erst dann wird der angegebene Link-Schritt ausgeführt. LFLAGS und LIBS stehen dabei für vorher definierte Variable, die spezielle Link-Optionen oder Libraries enthalten, die Variable $@ ist eine Abkürzung für den Namen des Ziels, hier also für linalg.
Möchte man nach einigen Änderungen linalg neu übersetzen, werden nach dem Aufruf "make linalg" oder einfach "make" (dann wird das erste Ziel im Makefile benutzt) alle nötigen Schritte unternommen. Hat sich nichts geändert, meldet sich make mit "`'linalg' is up to date.`"
Manchmal möchte man einen Schritt ausführen, obwohl das zugehörige File sich nicht geändert hat,

z.B. weil man in FFLAGS neue Optionen eingetragen hat. Eine Möglichkeit ist, das Makefile selbst in die Liste der Abhängigkeiten mit aufzunehmen. Allerdings wird dann bei jeder Änderung des Makefiles immer alles ganz neu übersetzt. Alternativ kann man auch einfach den Datumsstempel des aufzufrischenden Files mit "touch main.f" ändern.
Gewisse Aktionen kommen immer wieder in gleicher Weise vor, z.B. das Übersetzen von Source-Files in Object-Files. Um das Erstellen eines Makefiles weiter zu vereinfachen, kann man für solche Aktionen im Makefile Regeln definieren, die von File-Endungen abhängen. Zunächst muß man in einer "SUFFIXES"-Zeile angeben, welche Endungen besonders behandelt werden sollen, z.B.

```
.SUFFIXES: .o .f
```

Dann kann man die eigentlichen Regeln definieren, etwa:

```
.f.o:
        /usr/convex/fc $(FFLAGS) -c $*.f
```

Die 1. Zeile gibt an, daß die folgenden Aktionszeilen (bis zur nächsten Leerzeile) benutzt werden sollen, um aus .f-Files .o-Files zu erzeugen. Die Aktionszeilen haben die gleiche Form wie sonst, allerdings gibt es in Regeln zusätzliche Variable. $* etwa bezeichnet den Namen des Ziels ohne die Endung, d.h. $* = main, falls main.o erzeugt werden soll. Nach einer solchen Definition braucht man keine Abhängigkeiten von Object-Files mehr anzugeben, sondern make benutzt diese Regel, wenn es ein .o-File braucht und ein zugehöriges .f-File findet. Da einige Regeln eigentlich immer gebraucht werden (z.B. die oben angegebene), sind sie schon systemweit vordefiniert. Eine Liste dieser Regeln erhält man mit "make -p -q".

Eine nützliche Option von make ist "-n". Damit gibt make alle Kommandos aus, die es jetzt absetzen würde, ohne sie aber auszuführen. Das ist vor allem dann nützlich, wenn man Besonderheiten in seinem Makefile ausprobiert (wie eigene Regeln). Weitere Optionen und Möglichkeiten von make kann man dem Handbuch entnehmen.

Zum Schluß noch ein Beispiel, das zeigt, daß man mit make nicht nur kompilieren kann:

```
clean:
        -/bin/rm *.o
        -/bin/rm *~
```

Nach "make clean" werden immer die angegebenen Aufräum-Kommandos ausgeführt. "-" am Anfang einer Aktionszeile bewirkt, daß make mit der nächsten Zeile weiter macht, wenn eine Zeile einen Fehler meldet. (`rm` gibt z.B. einen Fehler zurück, wenn kein File gelöscht wird.)

2.4 Verwalten verschiedener Versionen - Das RCS-System

Bei der Entwicklung umfangreicherer Programme, und speziell bei der Optimierung, hat man oft das Problem, das man verschiedene Versionen seiner Programme aufbewahren möchte. Nun ist das mehrfache Speichern kompletter Pakete bei knappem Plattenplatz nicht sehr geschickt, außerdem auch nicht unbedingt sehr übersichtlich. Das Problem wird verschlimmert, wenn mehrere Leute an einem Programm arbeiten und sich bei gleichzeitigem Verändern ein wilder Versions-Verhau bildet.

Das Revision Control System (RCS) hilft dabei, hier die Übersicht zu bewahren. Es speichert in einem File die aktuelle Version, die sukzessiven Unterschiede zu früheren Versionen und zusätzliche

Verwaltungs-Informationen. Außerdem kontrolliert es schreibende Zugriffe mehrerer Benutzer und sperrt gegebenfalls, um Inkonsistenzen zu verhindern.

Die grundlegende Bedienung ist sehr einfach: Zunächst werden zu allen Source-Files die zugehörigen RCS-Files angelegt mit "ci *.f" (ci = "check in"). Für jedes File foo.f wird dadurch eine RCS-Datei "foo.f,v" angelegt, eine Kurzbeschreibung des Programms wird erfragt, die Versions-Nummer auf 1.1 gesetzt und das Original foo.f gelöscht. Die aktuelle Version wird wieder herauskopiert mit "co foo.f" (co = "check out").

Nur die Source-Files werden editiert, nie die RCS-Files selbst. Checkt man eine neue Version ein (wieder mit "ci foo.f"), wird die Versions-Nummer um eins erhöht (1.2, 1.3, ..). Bei jedem Einchecken wird noch ein Erklärungstext verlangt, der die Änderungen dieser Version kurz erläutern soll. Mit der Option -u kann man ein File einchecken, ohne daß es gelöscht wird, so daß man gleich damit weiterarbeiten kann. Eine neue Nummer kann man mit der Option -r angeben: "ci -r2.1 foo.f". Auch das Auslesen einer speziellen Version geht mit der Option -r: "co -r1.3 foo.f".

Um auch einem Source-File selbst ansehen zu können, zu welcher Version es gehört, kann man an beliebiger Stelle (am besten am Anfang in einem Kommentar) das Wort "\$Id\$" einfügen. Beim Einchecken wird es automatisch durch die Versions-Nummer sowie weitere Informationen ersetzt.[8] Auch die Zusammenarbeit mit make gestaltet sich sehr bequem, da meist entsprechende Regeln schon vordefiniert sind: Ist zu einem zu erzeugenden Object-File ein Source-File vorhanden, wird es kompiliert, ansonsten wird es aus dem RCS-File herausgezogen, kompiliert und wieder gelöscht. Um alle Files immer auf dem gleichen Versionsstand zu halten, auch wenn man nur einige davon geändert hat, geht man am besten folgendermaßen vor: Im Makefile definiert man eine Variable (z.B. SRC) mit den Namen aller Source-Files und fügt ein Ziel source ein:

```
source:
        co -r$(VERSION) $(SRC)
```

Mit "make source" holt man von allen Files die aktuelle Version heraus, ändert und fügt dann alle wieder ein mit "ci *.f". Einen vollständigen Satz einer alten Version erhält man mit "make VERSION=1.3 source". Bei größeren Programmprojekten wird man natürlich anders vorgehen und verschiedene Versionen für verschiedene Teile vorsehen.

Mit dem Programm rlog kann man sich über den Inhalt des RCS-Files informieren, so gibt "rlog foo.f" u.a. eine Liste aller Versionen mit zugehörigem Erklärungstext aus. Den Unterschied zwischen zwei Versionen erhält man mit "rcsdiff -rVERS1 -rVERS2 foo.f".

Das Programm rcs dient zur direkten Manipulation von RCS-Files. U.a. kann man damit auf folgende Weise eine Version aus einem RCS-File streichen:

```
rcs -o8.15 foo.f
```

Über weitergehende Möglichkeiten von RCS, z.B. das Erstellen von Verzweigungen im Versionsbaum, das Zusammenfaßen mehrerer Versionen zu einer oder das Verhindern von gleichzeitigen Schreibzugriffen mehrerer Benutzer auf ein File, informiert das Online-Manual unter den Stichworten ci, co, ident, rcs, rcsdiff, rcsmerge und rlog.

[8] Bei älteren RCS-Versionen muß man stattdessen "\$Header" benutzen.

2.5 Debuggen von linalg

Das Programm linalg enthielt am Anfang natürlich einige Fehler, die wir jetzt mit dem X-Window-fähigen Source-Debugger CXdb aufspüren wollen. In diesem Abschnitt sollen die Möglichkeiten des Debuggers nicht systematisch dargestellt werden – dies bleibt einem eigenen Kapitel vorbehalten –, sondern es werden die Ideen und Befehle beschrieben, die zum Debuggen unseres Beispielprogramms nötig sind.

1. Die fehlerhafte Ur-Version von linalg bekommen wir aus den RCS-Files mit "make VERSION=1.1 source". Zunächst muß das fehlerhafte Programm für den CXdb übersetzt werden. Dazu setzen wir die Variablen FFLAGS und LDFLAGS im Makefile auf "-cxdb" und übersetzen alles neu. Nach dem Aufruf von linalg erhalten wir die Fehlermeldung "Floating point exception" und ein Stack-Listing, dem man nach etwas Suchen entnehmen kann, daß der Fehler in der Routine GAUSSJ aufgetreten ist.
   Wir rufen nun den Debugger auf mit "cxdb linalg". Es erscheinen zwei Fenster: Ein Command-Fenster, in dem Debug-Kommandos eingegeben und die Antworten ausgegeben werden und das Menüs und Knöpfe für viele wichtige Debug-Befehle enthält, und das Source-Fenster, das den Source-Code des aktuellen Files enthält, im Moment den von main.f. Ein weiteres Fenster, das Help-Fenster, erscheint, wenn man im Command-Fenster den Help-Knopf drückt oder direkt das Kommando help eingibt. Hier kann man zu jedem Debug-Befehl und zu vielen Konzepten des CXdb ausführliche Informationen erhalten, über das Menü "Topics" Listen aller Suchbegriffe einsehen oder im Menü "HelpWindow" ein Online-Tutorial starten.
   Wir starten das Programm linalg im Debugger durch Eingabe des Kommandos "run". Zunächst erscheint ein neues Fenster, das Prozeß-Fenster, über das alle Ein- und Ausgaben des laufenden Programms abgewickelt werden. Bevor es aber zu irgendwelchen Ausgaben kommt, stoppt das Programm, und der CXdb informiert uns im Command-Fenster, daß in Zeile 69 der Routine GAUSSJ im File gaussj.f eine "Floating point exception" aufgetreten ist. An dieser Stelle befinden wir uns in einer L-Schleife (Z.68-70) in einer LL-Schleife (Z.64-75) innerhalb einer I-Schleife (Z.23-76).

   Um der Sache auf den Grund zu gehen, sehen wir uns zunächst die Werte aller lokalen Variablen an mit "info locals". U.a. sehen wir, daß

   ```
   I    = 1
   LL   = -2147091208
   L    = 86
   ```

   Der Wert für LL ist merkwürdig, denn LL sollte von 1 bis N laufen, und mit "print N" überzeugen wir uns von

   ```
   N    = 100
   ```

   Um uns anzusehen, wann LL diesen seltsamen Wert bekommt, setzen wir einen "Tracepoint" an den ersten Befehl der LL-Schleife (Z.65), d.h. wir wollen jedesmal, wenn diese Stelle passiert wird, einige Informationen erhalten. Wir definieren dazu:

   ```
   trace line 65 {echo/n "Zeile 65: LL = "; print LL; resume;}
   ```

   Damit werden jedesmal, wenn diese Stelle passiert wird, die Kommandos in den geschweiften Klammern ausgeführt, d.h. es wird - ohne anschließendes Newline - ein Text ausgegeben (echo/n), dann der Wert der Variablen LL angezeigt (print), und schließlich mit der

Programm-Ausführung fortgefahren (resume). Ohne "resume" würde bei Erreichen des Tracepoint das Programm angehalten. Der Tracepoint wird durch ein Symbol im Source-Fenster angezeigt.

Wir starten das Programm nun neu mit dem Kommando "run". Da das Programm vom letzten Lauf noch aktiv ist, fragt CXdb nach, ob der entsprechende Prozeß abgebrochen werden soll. Wir antworten mit "y" für ja. Als Ergebnis des Trace erhalten wir:

```
Zeile 65: LL = (INTEGER*4) 1
Zeile 65: LL = (INTEGER*4) 2
Zeile 65: LL = (INTEGER*4) 3
Zeile 65: LL = (INTEGER*4) 4
```

Danach erscheint wieder die alte Fehlermeldung. Daraus folgt, daß sich zwischen dem Tracepoint (Z.65) und dem Fehler (Z.69) der Wert von LL geändert hat, obwohl dort nur die Variablen DUM und A auftauchen. Wir sehen uns den Inhalt von A an mit "print A". CXdb gibt standardmäßig maximal 20 Werte eines Arrays auf einmal aus, in unserem Fall also nur den Anfang der ersten Zeile. Aber auch hier sieht man schon, daß darunter gigantisch große Zahlen sind, obwohl A am Anfang nur Zufallszahlen zwischen -1 und 1 enthalten soll. Mehrere Ursachen sind denkbar:

- SETMAT funktioniert nicht richtig
- Bis zum Aufruf von GAUSSJ wurde A geändert
- In GAUSSJ bekommt A diese Werte

Im letzten Fall müßte nicht einmal ein Fehler vorliegen. Es könnte ja sein, daß die Zufallszahlen in A in GAUSSJ zu numerischen Instabilitäten geführt haben. Allerdings ist der verwendete Algorithmus insbesondere wegen der vollständigen Pivot-Suche (Vertauschen von Zeilen und Spalten, um mit dem jeweils größten Element weiterzuarbeiten) sehr stabil, so daß diese Möglichkeit zumindest sehr unwahrscheinlich ist.
Um zu sehen, wann A so große Werte bekommt, wollen wir das Programm schrittweise durchgehen. Dazu müssen wir zunächst einen Haltepunkt ("breakpoint") am Anfang setzen, damit das Programm nach der Initialisierungsphase anhält:

```
break routine LINALG
```

Nach "run" (und der üblichen Abfrage) bleibt das Programm in Zeile 46 von main.f , der ersten ausführbaren Zeile, stehen. Der Breakpoint wird im Source-Fenster durch ein "B" vor der Zeile angezeigt. Mit den Kommandos "step" und "next" kann man nun in Einzelschritten durch das Programm gehen, wobei "next" einen Unterprogramm-Aufruf als einen Schritt ansieht und komplett ausführt, während "step" in das Unterprogramm hineingeht. Mit zwei next-Schritten führen wir die Routine GETVEC aus und sehen uns mit "print A" die ersten 20 Werte von A an: Sie liegen alle im richtigen Bereich. Mit drei weiteren next-Kommandos gehen wir bis vor den Aufruf von GAUSSJ. Die Werte von A sind erwartungsgemäß unverändert. Nun gehen wir mit "step" in die Routine GAUSSJ hinein. "print A" zeigt, daß jetzt A falsche Werte hat, es wird also nicht richtig an GAUSSJ übergeben. Mit "info args" erhalten wir Informationen über die Argumente von GAUSSJ; insbesondere sehen wir, daß A hier den Typ $REAL * 4$ hat, im Hauptprogramm aber als $REAL * 8$ deklariert ist. Damit ist der erste Fehler gefunden: Die Argumente und Variablen in GAUSSJ sind nicht deklariert

worden (die Routine wurde unverändert aus den "Numerical Recipes" übernommen).

Abhilfe: alle Variablen in gaussj deklarieren.

2. Die entsprechend korrigierte Version von linalg hat die Nummer 1.2, wir erhalten sie mit "make VERSION=1.2 source". Erneutes "make" und Starten von linalg bewirkt jetzt:

```
PAUSE: SINGULAR MATRIX
```

Wir starten wieder den Debugger mit "cxdb linalg" und das Programm selbst mit "run". Das Proceß-Fenster erscheint mit der PAUSE-Zeile und einer Eingabe-Einforderung. Wir antworten aber nicht dort - das wurde den Prozeß beenden -, sondern im Command-Fenster mit "Control-c", um den Prozeß anzuhalten. Das Source-Fenster zeigt die augenblickliche Position an: Das Programm steht in GAUSSJ in Zeile 41. Um zu sehen, warum die IF-Abfrage in Z.40 erfüllt wurde, sehen wir uns mit "info locals" die entsprechenden Variablen an:

```
K = (INTEGER*4) 51
IPIV = INTEGER*4(1:50) 0x8004c040
```

Der Fehler ist diesmal leicht zu finden: IPIV ist deklariert als 50-elementiges Feld, aber in Zeile 40 wird IPIV(51) abgefragt. Der Wert von K geht auf die Schleife in Z.33-43 zurück, in der K von 1 bis N=100 läuft. IPIV dagegen ist über den Parameter NMAX=50 deklariert.

Abhilfe: Parameter NMAX vergrößern (mindestens NMAX=100)

3. Die korrigierte Version 1.3 läuft durch GAUSSJ und die anschließenden Testroutinen durch, liefert aber eine "Floating Point Exception" in LUDCMP. Nach den bisherigen Erfahrungen ahnen wir die Fehlerquelle sofort: In LUDCMP - und in LUBKSB - werden die Variablen nicht deklariert, es kommt also wieder zu Inkonsistenzen ($REAL * 4 \Longleftrightarrow REAL * 8$).

Abhilfe: Für alle Variable Deklarationen einfügen.

Aus 1 - 3 lernen wir, daß man beim Übernehmen von Routinen aus anderen Quellen sehr vorsichtig sein muß, ob sie implizite Voraussetzungen enthalten. Besonders kritisch sind dabei die Typen von Argumenten (und evtl. passenden lokalen Variablen) und die Größen von Parametern.

4. Die letzte fehlerhafte Version ist 1.4. Sie läuft zwar ganz durch, aber der Test der durch LU-Zerlegung bestimmten inversen Matrix liefert eine große Abweichung vom erwarteten Ergebnis, während die Lösung des Gleichungssystems stimmt.
Wir setzen einen Breakpoint vor den 2. Aufruf von TESTMT ("break line 79"), lassen das Programm bis dahin laufen ("run") und sehen uns dann die linke obere Ecke der Matrix B an ("print B(1..5, 1..5)"). Einige Werte (z.B. B(2,2)) liegen außerhalb des Bereichs [-1,1], B kann also nicht mit der Ausgangsmatrix identisch sein. Ein Blick in main.f zeigt, daß B tatsächlich in Z.72 durch LUDCMP LU-zerlegt wurde. In TESTMT wird also nicht B, sondern A benötigt.

Nach dieser Änderung läuft das Programm problemlos durch. Es wird als Version 2.1 zur Ausgangsbasis für die nächsten Abschnitte.

2.6 Automatische Optimierung durch den Compiler

Ohne Optimierungsoption (-On, n=0,1,2,3) bzw. mit der Option -no führt der Compiler nur Grundoptimierungen durch, die den Code an die Hardware der Maschine anpassen. Im folgenden sollen die verschiedenen Optimierungsstufen kurz vorgestellt werden. Dabei beinhalten die höheren Stufen immer alle Operationen der niedrigeren Stufen.
In der Stufe -O0 werden maschinenunabhängige Optimierungen innerhalb eines Blocks durchgeführt, d.h. innerhalb einer Folge von Befehlen ohne irgendwelche Sprünge (DO, IF, ...). Beispiele sind das Eliminieren von mehrfachen Zuweisungen an eine Variable, die zwischendurch nicht benutzt wird oder das Einführen von temporären Variablen, um Zwischenschritte abzuspeichern, die mehrmals benutzt werden.
In der Stufe -O1 kommen skalare Optimierungen auf der Ebene von Prozeduren dazu, z.B. wird eine Variable, der eine Konstante zugewiesen wurde, durch diese Konstante ersetzt oder skalare Zuweisungen in Schleifen herausgezogen.
Erst mit Optimierungsstufe -O2 werden Vektorbefehle verwendet. Der Compiler analysiert verschiedene Formen von Schleifen und wandelt sie, soweit möglich, in Vektorkommandos um. Er führt zum Teil eine ganze Reihe recht komplizierter Umformungen durch, um diese "Vektorisierung" zu ermöglichen oder effektiver zu machen. Allerdings gibt es einige Programm-Konstruktionen, die das automatische Erkennen und Einführen von Vektor-Befehlen verhindern. Informationen darüber, welche Schleifen, ggf. mit welchen Methoden, vektorisiert werden konnten, und welche Faktoren dies bei den anderen Schleifen verhinderten, enthält der Optimierungs-Bericht, den der Compiler ab -O2 normalerweise ausgibt. Zusätzlich kann er noch Informationen über die Vektoren selbst enthalten. Mit der Compiler-Option -or kann man festlegen, ob man nur den Schleifen-Bericht (loop), nur die Vektor-Informationen (array), beides (all) oder gar nichts (none) haben möchte. Der Default ist "-or loop". Wir werden uns in Abschnitt 4.3 ausführlich damit beschäftigen, wie man solche Hindernisse für die Vektorisierung u.U. beseitigen kann.
Mit der Option -O3 versucht der Compiler, den Code zu parallelisieren, d.h. in Teile zu zerlegen, die unabhängig von verschiedenen Prozessoren bearbeitet werden können. Auf einer Maschine mit vier Prozessoren, auf der fast immer sechs bis zehn Jobs gleichzeitig laufen, wird man allerdings selten mehr als eine CPU zur Verfügung haben. Die Parallelisierung von Programmen ist Thema des Abschnitts 4.4
Die konkreten Auswirkungen der verschiedenen Optimierungsstufen auf das Programm linalg werden wir durch Vergleich der Laufzeiten von linalg nach Übersetzen aller Files mit "FFLAGS=-On" untersuchen. Die Zeiten messen wir mit dem /bin/time-Kommando: Der Aufruf "/bin/time linalg" bewirkt, daß zunächst das Programm linalg normal ausgeführt wird, danach werden zusätzlich drei Zeiten ausgegeben:

1. die "real"-Zeit, die angibt, wieviel Zeit vom Start des Kommandos bis zum Ende vergangen ist (sie interessiert uns hier nicht, da sie wesentlich von der Auslastung der Maschine durch andere Benutzer abhängt),
2. die "user"-Zeit, d.h. die CPU-Zeit, die zur Ausführung des Programms gebraucht wurde, wobei Zeiten für System-Aufrufe - etwa bei Ein- und Ausgaben - nicht mitzählen,
3. die "system"-Zeit, die gerade die CPU-Zeit von System-Aufrufen angibt.

Die tatsächliche CPU-Zeit ist also die Summe aus "user"- und "system"-Zeit.
Für die verschiedenen Optimierungsstufen ergaben sich folgende Ergebnisse:

C240:				C3840:			
-no	19.0 real	9.8 user	0.1 sys	-no	4.3 real	4.1 user	0.0 sys
-O0	38.2 real	8.2 user	0.0 sys	-O0	3.3 real	3.2 user	0.0 sys
-O1	12.8 real	4.3 user	0.0 sys	-O1	1.7 real	1.6 user	0.0 sys
-O2	2.3 real	1.7 user	0.0 sys	-O2	0.7 real	0.6 user	0.0 sys
-O3	2.5 real	2.0 user	0.1 sys	-O3	1.1 real	1.3 user	0.0 sys

Man sieht, daß schon die skalaren Optimierungen in -O1 in unserem Beispiel eine ganze Menge an Geschwindigkeitszuwachs bringen. Der größte Sprung geschieht aber beim Übergang zur Vektorisierung (-O2). Dies ist nicht weiter verwunderlich, denn die Convex ist als Vektorrechner von ihrer Hardware her speziell für Vektorbefehle ausgelegt, die aber erst ab -O2 benutzt werden. Daß beim Parallelisieren die Zeit wieder zunimmt, liegt an der Definition von "user"- und "system"-Zeit, die jeweils die Summen der Zeiten der einzelnen CPUs sind. Daß die real-Zeit hier niedriger ist als die user-Zeit, ist nur möglich bei Parallelisierung. Allerdings wurde diese Messung gemacht, als keine Batchjobs liefen. Normalerweise würde auch bei guter Parallelisierung die real-Zeit deutlich höher sein. Wie man auch bei normal ausgelasteter Maschine das Ausmaß an Parallel-Verarbeitung bestimmen kann, werden wir in Kapitel 4.4.3) lernen.

2.7 Application Compiler

Der Application-Compiler – eigentlich ein Pre-Compiler, denn er ruft den normalen Fortran- und/ oder C-Compiler sowie den Loader auf – ist ein neues Werkzeug von Convex zur Programm-Entwicklung. Er analysiert im Gegensatz zu herkömmlichen Compilern nicht einzelne Routinen, sondern ein ganzes Programm und legt eine umfangreiche Datenbasis darüber an. Auf diese Weise ist er in der Lage, wesentlich weitergehende Fehlerprüfungen und Optimierungen durchzuführen. In diesem Abschnitt wollen wir sehen, was er für unser Beispielprogramm linalg ausrichten kann, wenn wir ihn auf möglichst einfache Weise einsetzen, ohne weitere Analysen oder speziellere Möglichkeiten auszunutzen. Die Schnittstelle zum Application-Compiler ist das "build"-Programm: Ähnlich wie beim make muß man ein spezielles File namens "Buildfile" erstellen, das die Source-Files, Libraries und Optionen enthält. Allerdings braucht man im Gegensatz zu make keine Abhängigkeiten anzugeben, die findet der Compiler selbst heraus. Für gewiefte Optimierer gibt es viele Optionen und Flags (s. Abschnitt 3.3.3); er arbeitet aber sehr gut schon mit seinen Standard-Einstellungen. Um den Application-Compiler eizusetzen, muß man den Such-Pfad erweitern um das Verzeichnis tt /usr/convex/ipo/bin. Dann legt man im Verzeichnis mit den Sourcen ein Buildfile (s.u.) an und startet den Compiler mit "build".

Falls alle benötigten Source-Files (und nur diese) in einem Directory liegen und nur Fortran benutzt wird, genügt folgendes einfache Buildfile:

```
#
# alle Files im selben Directory wie dieses Buildfile, alle mit -O2
#
. -O2

link FORTRAN

options -o linalg -check all
```

Zunächst wollen wir den Application-Compiler zur Fehlersuche verwenden. Wir extrahieren also mit "make VERSION=1.1 source" die fehlerhafte Urversion von linalg und starten den Compiler mit

```
build |& tee Build.report
```

(Da die erzeugte Ausgabe umfangreich ist, leiten wir sie in das File Build.report um. Um aber auch noch unmittelbar sehen zu können, was passiert, verwenden wir das "tee"-Filter, das seine Eingabe in ein File und auf die Ausgabe schreibt.)
Zunächst erscheinen folgende Warnungen:

```
Warning: Argument number 1 of GAUSSJ has inconsistent type in
./main.f on line 57 and ./gaussj.f on line 1
Warning: Argument number 4 of GAUSSJ has inconsistent type in
./main.f on line 57 and ./gaussj.f on line 1
Warning: Argument number 1 of LUDCMP has inconsistent type in
./main.f on line 72 and ./ludcmp.f on line 1
Warning: Argument number 5 of LUDCMP has inconsistent type in
./main.f on line 72 and ./ludcmp.f on line 1
Warning: Argument number 1 of LUBKSB has inconsistent type in
./main.f on line 73 and ./lubksb.f on line 1
Warning: Argument number 5 of LUBKSB has inconsistent type in
./main.f on line 73 and ./lubksb.f on line 1
Warning: Argument number 1 of LUBKSB has inconsistent type in
./luinv.f on line 42 and ./lubksb.f on line 1
Warning: Argument number 5 of LUBKSB has inconsistent type in
./luinv.f on line 42 and ./lubksb.f on line 1
```

Nach einer Menge von Informationen zur Programm-Optimierung werden die gefundenen Fehler noch einmal tabellarisch zusammengefaßt:

Errors Detected

Procedure	Wrong Mis-Matched Args	Mis-Number Of Args	Matched Return Type	Invalid Aliases	Scalar Passed To Array	Invalid SubScript	Variables Not Initialized
LINALG	6						
GAUSSJ							
LUINV	2						
GETVEC							
ERNORM							
SETVEC							
TESTMT							
LUDCMP							
LUBKSB							
Totals	8						
build -check	types	types	types			arrays	init

Der Application-Compiler hat also einige der Fehler automatisch gefunden, die wir vorher mühsam mit dem Debugger aufgespürt hatten.
Nun wollen wir sehen, was es mit dem verbesserten Optimieren auf sich hat. Zunächst löschen wir die alte Datenbasis mit "build -c", dann holen wir uns die Version 2.1, und starten wieder mit "build". Wir ignorieren die umfangreiche Ausgabe bis auf die abschließende Fehlertabelle, die diesmal keine Fehler anzeigt, und starten unser Programm mit "/bin/time linalg". Mit dem Ergebnis erweitern wir unsere obige Laufzeittabelle der C3840 zu:

C3840:			
-no	4.3 real	4.1 user	0.0 sys
-O0	3.3 real	3.2 user	0.0 sys
-O1	1.7 real	1.6 user	0.0 sys
-O2	0.7 real	0.6 user	0.0 sys
Appl.Comp.	0.6 real	0.5 user	0.0 sys
-O3	1.1 real	1.3 user	0.0 sys

Ohne großen Aufwand hat der Application-Compiler das Programm linalg, das an sich schon vom normalen Vektorisierer gut optimiert wird, um 11% beschleunigt. Wie er das gemacht hat und wie man vielleicht noch mehr erreichen kann, werden wir uns in den Abschnitten 3.3.1 und 4.2 ansehen.

2.8 CXpa

Durch das Vektorisieren sind Beschleunigungen bis zum Faktor 6 durchaus erreichbar. Der Vergleich der Zeiten für O1- und O2-Optimierung zeigt aber, daß wir davon noch weit entfernt sind. Es macht allerdings keinen Sinn, nun wild "herumzuoptimieren", denn was nützt eine 10-fache Beschleunigung einer Routine, die nur 1 Promille der Gesamt-Rechenzeit verbraucht? Der nächste Schritt besteht also darin, die Stellen im Programm herauszufinden, die den größten Teil der CPU-Zeit benötigen. Zu diesem Zweck dienen sogenannte Profiler, das sind Programme, die das Laufzeitverhalten einzelner Programmteile (Routinen, Schleifen, Blöcke) analysieren und tabellarisch zusammenstellen. In diesem Abschnitt soll am Beispielprogramm linalg gezeigt werden, wie man mit dem Convex-Profiler CXpa ("ConveX Performance Analyzer") arbeitet. Ausführlich wird der CXpa im Abschnitt 3.2 vorgestellt. Natürlich macht es für unser Beispiel nicht allzuviel Sinn, den Programm-Ablauf als Ganzes beschleunigen zu wollen. Stattdessen wollen wir versuchen herauszufinden, welche Routinen schlecht vektorisieren, um im nächsten Abschnitt einige einfache Optimierungen auszuprobieren. Dazu vergleichen wir die Laufzeit-Daten für eine O1- und eine O2-optimierte Version miteinander. Um den CXpa einzusetzen, muß man zunächst mit der Option -pa übersetzen und linken, d.h. diese Option muß im Makefile bei FFLAGS und LDFLAGS eingetragen werden. Nach dem Neu-Übersetzen wird der Profiler mit "cpa linalg" aufgerufen. Daraufhin wird der Bildschirm neu aufgebaut, und es erscheint (neben Startinformationen) der Prompt "(cpa)", an dem CXpa-Kommandos eingegeben werden können. Nun muß man angeben, welche Informationen i.f. gesammelt werden sollen. Mit dem Kommando

```
monitor routine all
```

werden dazu alle Routinen zum "Profilen" ausgewählt. Als nächstes wird mit dem Kommando "run" das Programm unter CXpa-Kontrolle gestartet; dabei werden Laufzeit-Informationen in ein File namens "cpa.pdf" geschrieben. Mit dem Befehl "analyze routine" wird dieses File ausgewertet und einige Tabellen ausgegeben, von denen uns hier nur der Beginn der ersten interessiert:

Routine Performance Analysis
(sorted by CPU time (less children))

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
0.756	42.8%	0.756	42.8%	1	gaussj
0.575	32.6%	0.575	32.6%	2	testmt
0.226	12.8%	0.226	12.8%	101	lubksb
0.119	6.7%	0.119	6.7%	1	ludcmp
0.037	2.1%	0.037	2.1%	10100	drand_$n
0.034	1.9%	0.070	4.0%	1	getmat
6.362m	0.4%	6.362m	0.4%	2	ernorm
5.420m	0.3%	5.420m	0.3%	2	setmat
2.266m	0.1%	0.225	12.8%	1	luinv
0.334m	0.0%	0.692m	0.0%	1	getvec
0.147m	0.0%	0.225m	0.0%	2	_findenv
0.136m	0.0%	1.763	99.9%	1	main
0.133m	0.0%	0.146m	0.0%	4	lwrt_A
0.098m	0.0%	0.235m	0.0%	1	f_exit
0.092m	0.0%	0.316m	0.0%	4	fwrite
0.078m	0.0%	0.078m	0.0%	38	strncmp
0.075m	0.0%	0.361m	0.0%	3	ini_std
.	.	.	.	.	.
.	.	.	.	.	.
0.059m	0.0%	0.059m	0.0%	2	setvec

Zu jeder Routine – auch solchen, die aus der Fortran-Standard-Bibliothek stammen – ist hier angegeben, wieviel CPU-Zeit in der Routine selbst verbraucht wurde, und zwar absolut (in Sekunden oder, durch "m" markiert, Millisekunden) und relativ zur gesamten CPU-Zeit, danach, wieviel CPU-Zeit in der Routine und allen von ihr aufgerufenen Unterroutinen verbraucht wurde (wieder absolut und relativ) und schließlich, wie oft die Routine insgesamt aufgerufen wurde.
Wir beenden den CXpa durch "quit" und wiederholen die ganze Prozedur mit der Optimierungsstufe O2. Da bereits ein File cpa.pdf existiert, werden wir beim "run" gefragt, ob wir dieses File überschreiben wollen. Mit "y" bejahen wir und erhalten nach dem analyze-Schritt folgende Tabelle:

Routine Performance Analysis
(sorted by CPU time (less children))

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
0.442	57.2%	0.442	57.2%	1	gaussj
0.133	17.1%	0.133	17.2%	2	testmt
0.079	10.2%	0.079	10.2%	101	lubksb
0.048	6.3%	0.048	6.3%	1	ludcmp
0.037	4.8%	0.037	4.8%	10100	drand_$n
0.028	3.6%	0.065	8.4%	1	getmat
1.388m	0.2%	1.388m	0.2%	2	ernorm
0.889m	0.1%	0.889m	0.1%	2	setmat
0.769m	0.1%	0.078	10.1%	1	luinv
0.273m	0.0%	0.644m	0.1%	1	getvec
0.137m	0.0%	0.213m	0.0%	2	_findenv
0.134m	0.0%	0.145m	0.0%	4	lwrt_A
0.132m	0.0%	0.772	99.9%	1	main
0.100m	0.0%	0.243m	0.0%	1	f_exit
0.094m	0.0%	0.323m	0.0%	4	fwrite
0.077m	0.0%	0.383m	0.0%	3	ini_std
0.076m	0.0%	0.076m	0.0%	38	strncmp
.	.	.	.	.	.
.	.	.	.	.	.
0.016m	0.0%	0.016m	0.0%	2	setvec

Wir berechnen aus den 1. Spalten der beiden Tabellen die "Vektorisierungs-Beschleunigung" B = cpu(O1)/cpu(O2) und erhalten:

	CPU Time O1	CPU Time O2	B
setmat	5.420m	0.889m	6.1
ernorm	6.362m	1.388m	4.6
testmt	0.575	0.133	4.3
setvec	0.059m	0.016m	3.7
luinv	2.266m	0.769m	2.9
lubksb	0.226	0.079	2.9
ludcmp	0.119	0.048	2.5
gaussj	0.756	0.442	1.7
getmat	0.034	0.028	1.2
getvec	0.334m	0.273m	1.2

Legt man den Wert 6.0 als (nahezu) optimal zugrunde, dann kann man die Routinen nach ihrem Vektorisierungsgrad grob folgendermaßen einteilen:

testmt, ernorm, setvec, setmat	sehr gut
ludcmp, lubksb, luinv	mittelmäßig
gaussj	schlecht
getvec, getmat	gar nicht

Angesichts der Anteile der Routinen an der CPU-Zeit würde man sich normalerweise als nächstes der Routine gaussj als geeignetem Kandidaten für weitere Optimierungsversuche zuwenden.

2.9 Einfache Handoptimierungen

In diesem Abschnitt sollen einige wichtige Optimierungs-Ansätze vorgestellt und am Programm linalg ausprobiert werden, die häufig nutzbringend angewendet werden können, ohne daß ein tieferes Verständnis der Maschinen-Architektur nötig ist. Das Spektrum reicht dabei von einfachen Code-Änderungen bis zur Verwendung anderer Algorithmen. Im Einzelfall müssen die Auswirkungen jeweils durch CPU-Zeit-Messungen untersucht werden. Systematischere Untersuchungen folgenden im Kapitel 4.

2.9.1 Speicherbank-Konflikte

Bei vielen Programmen ist die Geschwindigkeit, mit der auf den Speicher zugegriffen werden kann, ein limitierender Faktor. Ein wichtiges Problem dabei sind die sogenannten "Bank-Konflikte": Die Speicherzellen sind in Gruppen eingeteilt, "Bänke" genannt, wobei schnell aufeinanderfolgende Zugriffe auf Speicherzellen innerhalb einer Bank wesentlich langsamer sind als zwischen verschiedenen Bänken. Ein Ziel muß es also sein, solche Zugriffe zu vermeiden. Ohne hier auf weitere Einzelheiten einzugehen (s. Abschnitt 4.3.4), kann man als Faustregel festhalten, daß man bei der Deklaration von Arrays ungerade Dimensionen verwenden sollte. Damit verschenkt man zwar etwas Speicherplatz, gewinnt aber bei manchen Problemen einiges an Geschwindigkeit.
Im Programm linalg, Version 2.2, wurde der Parameter NMAX in main.f, gaussj.f und ludcmp.f von 100 auf 101 gesetzt und damit die Speicher-Struktur der verwendeten Arrays geändert. Mit den Profiling-Methoden aus Abschnitt 2.8 und Optimierungsstufe O2 ergeben sich folgende Ergebnisse:

Routine	CPU-Zeit NMAX=100	CPU-Zeit NMAX=101	Zeitersparnis [%]
setmat	0.889m	0.880m	1.0
ernorm	1.388m	1.368m	1.5
testmt	0.133	0.115	15.7
setvec	0.016m	0.016m	0.0
luinv	0.769m	0.740m	3.9
lubksb	0.079	0.073	8.2
ludcmp	0.048	0.045	6.7
gaussj	0.442	0.414	6.8
getmat	0.028	0.027	3.7
getvec	0.273m	0.274m	-0.4
total(main)	0.772	0.716	7.8

2.9.2 Vektorlänge

Um die Geschwindigkeit eines Vektorrechners auszunutzen, muß man natürlich die Vektoreinheiten gut ausnutzen. Da mit jeder Vektoroperation eine Startzeit verknüpft ist, die unabhängig von der Vektorlänge ist, heißt das insbesondere, daß man kurze Vektorlängen möglichst vermeiden bzw. für möglichst lange Vektoren sorgen muß.

Eine Methode, dies zu erreichen, die manchmal sehr gut funktioniert, ist das Umschreiben von Array- auf Vektor-Operationen. Dies läßt sich in unserem Beispiel vor allem an den Routinen getmat und setmat ausprobieren: Beide Routinen beziehen sich nicht auf die Array-Struktur der Argumente, sondern greifen einfach auf alle Elemente zu. Daher können die übergebenen Arrays als riesige Vektoren aufgefaßt und entsprechend indiziert werden. Allerdings muß man jetzt die "physikalischen" Arraygrößen (NMAX) verwenden, was die Zahl der Operationen entsprechend vergrößert (hier von 100x100 auf 101x101).

Routine	CPU-Zeit (NMAX=100)	CPU-Zeit (NMAX=101)
getmat	0.027	0.028
setmat	0.880m	0.879m

Die Ausbeute ist mager, weil bei 100x100-Matrizen die Vektoren schon die Länge 100 haben, verglichen mit der Maximallänge der Vektoreinheit von 128 also schon fast optimal. Außerdem vektorisiert die Schleife in getmat wegen der Aufrufe von drand sowieso nicht. Bei kleineren Matrizen lassen sich aber z.T. recht erhebliche Beschleunigungen erzielen.

2.9.3 Bibliotheken

Die bestmögliche Ausnutzung der Rechner-Architektur im Rahmen eines vorgegebenen Algorithmus ist ein sehr kompliziertes Problem und von einem Anwender in endlicher Zeit nicht zu leisten. Daher wird für Vektorrechner i.a. eine große Bibliothek mitgeliefert, die für die wichtigsten numerischen Probleme handoptimierte, oft in Assembler geschriebene Routinen zur Verfügung stellt. Eine solche Bibliothek für die Convex ist die Veclib, die mit -lveclib angebunden werden kann. Sind zur Lösung eines Problems geeignete Routinen in der Veclib vorhanden, sollte man diese verwenden, statt sie selbst zu schreiben. Die dadurch erzielten Geschwindigkeitsgewinne sind meistens dramatisch, wie das folgende Beispiel zeigt:
Im Programm linalg werden die Rümpfe von LUDCMP, LUBKSB und LUINV durch die entsprechenden Veclib-Routinen DGEFA, DGESL und DGEDI ersetzt, wobei kleinere Parameter-Anpassungen vorzunehmen sind. Normalerweise würde man sich einen Unterprogramm-Aufruf sparen und die Veclib-Routinen direkt benutzen. Eine Messung der Laufzeiten ergibt für die betroffenen Routinen folgendes Ergebnis:

Routine	CPU-Zeit eigene Routine	CPU-Zeit Veclib	Zeitersparnis [%]
LUDCMP	0.048	0.012	75.0
LUINV	0.081	0.024	70.4
LUDCMP + LUINV + $1 * $LUBKSB	0.130	0.036	72.3

Dabei wurden diesmal die Zeiten incl. der von den Routinen aufgerufenen Unterprogramme verglichen, da in den LU-Routinen selbst ja nichts mehr passiert. Die dritte Zeile enthält den gesamten Anteil der LU-Routinen, wobei zu berücksichtigen ist, daß von den 101 LUBKSB-Aufrufen der alten Version nur einer mitgezählt werden darf, da die anderen im LUINV-Anteil mitgerechnet werden.

Der Zeitgewinn ist beträchtlich: Gegenüber dem Faktor 3, der bei den LU-Routinen beim automatischen Vektorisieren erzielt wurde, kann i.a. nicht mehr als ein weiterer Faktor 2 durch geeignete Hand-Optimierung gewonnen werden (d.h. ein Faktor 6 beim Vektorisieren ist fast optimal), hier wurde aber ein Faktor 4 erreicht!

2.9.4 Algorithmen

Einen wesentlichen Einfluß auf das Laufzeit-Verhalten hat natürlich der verwendete Algorithmus; man denke z.B. nur an den Unterschied zwischen normaler und "Fast" Fourier-Transformation ($N*N$ gegen $N*\log(N)$ bei N Gitterpunkten). Zwar wird man einen Algorithmus auch nach den Kriterien der Genauigkeit und der numerischen Stabilität beurteilen müssen, man sollte aber den Geschwindigkeits-Aspekt immer berücksichtigen. Dies wird allerdings dadurch erschwert, daß die Rechner-Architektur zusätzlich eine Rolle spielt. Es gibt zum Beispiel Algorithmen, die theoretisch sehr schnell sind, aber trotzdem nicht eingesetzt werden, weil sie nicht vektorisieren.
Im Beispiel-Programm linalg werden zur Lösung eines linearen Gleichungssystems und zur Invertierung einer Matrix zwei verschiedene Algorithmen verwendet: Gauss-Jordan mit vollständiger Pivot-Suche und LU-Zerlegung mit partieller Pivot-Suche. Der oben angestellte Rechenzeit-Vergleich zeigt, daß die LU-Zerlegung deutlich schneller ist, erst recht, wenn man die Veclib-Routinen benutzt. Dies hat mehrere Ursachen: Zum einen ist vollständige Pivot-Suche sehr langsam und verhindert gutes Vektorisieren. Da partielle Pivot-Suche fast genauso stabil ist, wird man i. d. R. darauf zurückgreifen. Zum zweiten ist die LU-Zerlegung bei einer rechten Seite nicht langsamer als Gauss-Jordan, bei vielen Gleichungen aber schneller: Man muß nur einmal zerlegen und kann dann mit der Rücksubstitution sehr schnell viele rechte Seiten auswerten. Da auch sonst nichts für das Gauss-Jordan-Verfahren spricht, wird es heute nicht mehr eingesetzt, es ist auch in den gängigen Libraries (NAG, IMSL, Veclib) nicht enthalten.
Als weiteres Beispiel betrachten wir die Routinen GETVEC bzw. GETMAT. Sie besetzen einen Vektor bzw. eine Matrix mit gleichverteilten Zufalls-Zahlen zwischen -1 und 1. Bisher wurde dazu die Standard-Routine DRAND benutzt, die jeweils einen Wert zurückgibt. Die entsprechenden Schleifen können daher nicht vektorisiert werden. Die Veclib-Routine RANV dagegen gibt einen ganzen Vektor von gleichverteilten Zufallszahlen auf einmal zurück; außerdem ist sie der Convex-Architektur optimal angepaßt. Ohne genauere Kenntnis der jeweils verwendeten Algorithmen läßt sich über die Güte der Zufallszahlen von RANV bzw. DRAND nichts sagen, für unsere einfachen Zwecke sind aber beide sicher gleichwertig. In Version 2.5 sind GETVEC und GETMAT entsprechend geändert. Mit dem CXpa erhält man für die CPU-Zeiten incl. aufgerufener Unterroutinen folgende Ergebnisse:

Routine	CPU-Zeit DRAND	CPU-Zeit RANV	Zeitersparnis [%]
getmat	0.028	1.020m	96.4
getvec	0.879m	0.020m	97.7

3 Die Tools im einzelnen

3.1 CXdb

3.1.1 Die Oberfläche des CXdb

Der CXdb-Debugger ist vor allem für die Arbeit unter dem X11-Window-System gedacht und präsentiert sich dann mit mehreren Fenstern. Er kann aber auch auf normalen Alpha-Terminals benutzt werden, wobei ein textorientiertes Fenstersystem, die sogenannten Maryland-Windows, benutzt wird. Das Erscheinungbild und das Arbeiten unter diesen beiden Umgebungen sowie das integrierte Hilfe-System werden in diesem Abschnitt vorgestellt.

Die X-Window-Schnittstelle Die normale Schnittstelle zum CXdb-Debugger ist das X11-System: Ein- und Ausgaben und die Anzeigen vielfältiger Informationen geschehen in mehreren Fenstern, die wiederum mehrfach unterteilt sind. Die oberste Zeile (unter der Titelzeile des Window-Managers) enthält jeweils durch Anklicken aufklappbare Menüs, mit denen i.a. weitere Fenster geöffnet werden können, sowie rechts eine Nummer, die das Fenster im CXdb identifiziert.
Das wichtigste Fenster ist das Command-Fenster, das immer als erstes erscheint. Es enthält unter der Window-Menü-Leiste eine weitere Zeile mit Menüs, mit denen viele Funktionen des CXdb mit der Maus direkt oder über Unter-Menüs aufgerufen werden können. Darunter befindet sich der eigentliche Kommando-Bereich, in dem hinter dem Prompt "(CXdb) " Debugger-Befehle eingegeben werden können und die zugehörigen Ausgaben sowie andere CXdb-Meldungen erscheinen. Unvollständige Eingaben können durch Drücken der "TAB"-Taste ergänzt werden, falls dies eindeutig möglich ist; außerdem kann man die X-Window-Clipboard-Funktionen in gewohnter Weise nutzen. Der Kommando-Bereich ist von Schiebe-Balken umgeben, um alte und überlange Ausgaben sichtbar machen zu können. Schließlich befinden sich in der letzten Zeile einige Knöpfe, mit denen die wichtigsten Kommandos direkt durch Anklicken ausgeführt werden können.
Ein weiteres wichtiges Fenster ist das Source-Fenster, in dem das aktuelle Programm angezeigt wird. Es erscheint normalerweise beim Start des CXdb automatisch, kann aber vom Command-Fenster aus im Menü "CXdbWindows" mit "create Source" explizit gestartet werden. Auf dieselbe Weise kann man auch mehr als ein Source-Fenster auf einmal erzeugen. Unter der üblichen Menü-Zeile befindet sich eine Status-Zeile, die den Namen des Source-Files und Informationen über den laufenden Prozeß enthält. Darunter schließlich wird der aktuelle Source-Code angezeigt, versehen mit Schiebe-Balken und Zeilen-Nummern. Die aktuelle Zeile (genauer: Source-Unit, s.u.) ist invertiert dargestellt; neben den Zeilennummern zeigen kleine Symbole Breakpoints u. ä. an (s.u.).

Das Prozeß-Fenster ist ein normales xterm-Fenster, in das die Ein- und Ausgaben des laufenden Programms geleitet werden.

Mit dem "help"-Befehl (entweder direkt im Command-Fenster eingegeben oder über Menü oder Knopf aufgerufen) erzeugt man ein Help-Fenster, über das man Zugriff auf das umfangreiche Hilfe-System des CXdb hat, das unten vorgestellt wird.

Viele weitere Fenster werden automatisch erzeugt, um umfangreichere Informationen darzustellen, z.B. über Speicherbereiche, Files oder Register.

Maryland-Windows Um den CXdb auch den Benutzern zugänglich zu machen, die keinen X-Window-Schirm haben, gibt es eine Schnittstelle, um Fenster in normalen ASCII-Terminals zu emulieren, die sogenannten Maryland-Windows. Die Funktionalität ist gegenüber dem X-Interface etwas eingeschränkt, insbesondere werden mehr Ein- und Ausgaben über das Command-Fenster

abgewickelt statt über Menüs oder eigene Fenster.

Der Bildschirm ist zunächst in drei Bereiche eingeteilt, die jeweils sechs Zeilen umfassen, nämlich Source-, Process- und Command-Window. Nach dem Help-Befehl erscheint als viertes das Help-Window, das Hilfetexte enthält, und das Source- und Process-Fenster teilweise überlappt. Die oberste Zeile eines jeden Fensters ist seine Titelzeile; sie zeigt invertiert den Namen und die Nummer des Fensters an.

Jeweils ein Fenster ist aktiv und nimmt Eingaben entgegen; es ist durch "*" am linken Rand markiert. Mit Meta-n (Meta = Alternate oder Escape, je nach Tastatur) macht man das nächste Fenster aktiv, mit Meta-p das vorige. Das aktive Fenster kommt dabei, falls es verdeckt ist, nach vorne. In einem Fenster kann man mit den Pfeiltasten zeilenweise und mit Ctrl-v und Meta-v schirmweise vor- und zurückblättern.

Man kann die Größe und Position der Fenster auch verändern. Dazu muß man zunächst das Fenster, das man verändern möchte, aktiv machen. Zum Verkleinern oder Vergrößern drückt man jetzt Meta-z. Nun kann man mit den Tasten Ctrl-n, Ctrl-p, Ctrl-b und Ctrl-f die rechte untere Ecke verschieben und mit der Leertaste die erreichte Größe festhalten. Dies wird auch in einem kleinen Hilfs-Fenster angezeigt. Das Verschieben geht ähnlich, man startet es mit Meta-m. Ein Fenster darf allerdings nicht über den Rand des Schirms ragen. Man kann die Größe und Lage der Fenster bei Maryland-Windows auch schon beim Start über Parameter festlegen. Näheres findet man im Online-Manual ("man cxdb").

Schließlich kann man Fenster (außer dem Command-Fenster) auch schließen mit Meta-k.

Weitere Möglichkeiten von Maryland-Windows kann man dem CXdb-Help entnehmen mit "help maryland windows".

Das Hilfe-System Das Online-Hilfe-System des CXdb enthält das vollständige "CXdb Reference Manual" und ist bei grundlegendem Verständnis der Debugger-Funktionen als Nachschlagewerk völlig ausreichend. Man kann es zum einen durch "help irgendwas" im Command-Fenster ausfragen. Unter X11 kann man aber auch mit "help" ein eigenes Help-Fenster öffnen. Neben der direkten Suche nach Begriffen kann man sich durch hypertextartige verkettete Texte wühlen: Durch Anklicken eines unterstrichenen Worts erhält man jeweils eine zugehörige neue Hilfstafel.
Das Inhaltsverzeichnis, das man zunächst bekommt und das man mit dem "Contents"-Knopf immer erreichen kann, enthält eine Übersicht der vorhandenen Hilfskategorien:

"Concepts"	allgemeine CXdb-Konzepte
"Commands"	Referenz-Seiten aller Befehle
"Parameters"	Beschreibungen wichtiger Parameter
"Messages"	Erklärungen zu CXdb-Meldungen
"Tutorial"	Online-Tutorial, das an Beispielen die wichtigsten CXdb-Befehle vorstellt

3.1.2 Schrittweiser Programmablauf

In diesem Abschnitt werden neben den grundlegenden Befehlen zum Starten eines Programms unter CXdb-Kontrolle die vielfältigen Möglichkeiten vorgeführt, schrittweise durch das Programm zu gehen. Eine Besonderheit ist hier die variable "Schrittweite", die von ganzen Routinen bis zu einzelnen Teilausdrücken und schließlich zu Maschinen-Kommandos reicht. Als Beispiel dient wieder das Programm linalg in der Version 2.1, das hier und in den nächsten Abschnitten ohne

Optimierung zu übersetzen ist.

Grundlagen Um ein Programm mit dem CXdb untersuchen zu können, muß man es mit der Option "-cxdb" sowohl kompilieren als auch linken. Da sich diese Option auch mit allen Optimierungsstufen verträgt und weil die Ausführungszeiten eines Programms sich dadurch nicht erhöhen, empfiehlt es sich, sie grundsätzlich zu benutzen. Insbesondere die Möglichkeit, schon lange laufende Programme mitten im Lauf zu debuggen (s. 3.1.5), lohnt den kleinen Mehraufwand beim Übersetzen. Daß die Auswirkungen der cxdb-Option auf den normalen Programmlauf vernachlässigbar sind, liegt daran, daß die nötigen Informationen nicht in das ausführbare Programm mit eingebunden sind, sondern in einem Unterverzeichnis .CXdb getrennt vorliegen.

Nach dem Übersetzungsschritt ruft man den Debugger mit "cxdb Programmname" auf (wobei man bei Verwendung von X11 i.a. ein "&" anhängt, um den Prozeß im Hintergrund zu starten - schließlich macht er ja sein eigenes Fenster auf). Es erscheinen nun das Command- und das Source-Fenster. Mit dem Kommando "run [args]" startet man den Ablauf des Programms. "args" bezeichnet dabei irgendwelche Kommandozeilen-Parameter, die man sonst direkt beim Programmaufruf in der Form "programmname args" mitgeben würde. [9] Dabei wird auch das Prozeß-Fenster geöffnet, das die Ausgaben des laufenden Programms enthält und in dem auch ggf. erforderliche Programm-Eingaben erfolgen. Das Programm läuft nun normalerweise bis zum Ende durch bzw. bis es auf einen Fehler stößt, der ohne Debugger-Kontrolle zum Absturz des Programms führen würde. Jetzt wird dagegen nur die entsprechende Fehlermeldung im Command-Fenster ausgegeben und das Programm angehalten. Der augenblickliche Stand des Programms wird dabei durch Invertierung der entsprechenden Zeile im Source-Fenster angezeigt. Man kann nun durch Ansehen der Variablen (s.u.) versuchen, der Ursache des Fehlers auf die Spur zu kommen. Um jetzt - oder später - das Programm mit den zuletzt benutzten Parametern erneut aufzurufen, benutzt man das Kommando "rerun". Man kann das Programm während des Laufes unterbrechen, indem man Control-c im Command-Fenster (nicht im Process-Fenster) eingibt. Mit "continue &" kann man mit der Ausführung des Programms fortfahren, wobei es bei "&" im Hintergrund weiterarbeitet, so daß man im Command-Fenster Befehle eingeben kann. Insbesondere kann man nun mit "stop" den Prozeß wieder anhalten.
Mit "kill process" kann man den Prozeß abbrechen, etwa um von vorne zu beginnen, mit "quit" den CXdb verlassen. Dabei wird ggf. ein noch laufender Prozeß nach einer Sicherheitsabfrage beendet.

Durchgehen in Anweisungs-Schritten Meist beginnt man mit einem einfachen "run", um direkt zur Fehlerstelle vorzudringen. Aber zur genaueren Untersuchung geht man meist in kleineren Schritten durch das Programm. Bevor man das aber tun kann, muß man zunächst einen Haltepunkt setzen (s.u.), von dem aus man in Einzelschritten weitergehen möchte, und das Programm bis dahin ablaufen lassen. Um ganz vorne beginnen zu können, geben wir also ein:

```
break routine linalg
run
```

Mit den Befehlen "step" und "next" können wir nun schrittweise durch das Programm gehen, wobei ein Schritt eine Anweisung umfaßt. Im Source-Fenster wird dabei durch Invertieren jeweils angezeigt, an welcher Stelle sich das Programm befindet. Der Unterschied zwischen "step" und "next" besteht darin, daß für "next" ein Unterprogramm-Aufruf eine Anweisung ist, die in einem Schritt ganz ausgeführt wird, während man mit "step" in die Routine hineinspringt und auch dort in Einzelschritten weitergeht. Beiden Kommandos kann man noch die Anzahl der Schritte,

[9] Hier wie im folgenden werden optionale Parameter in [ ] angegeben.

die auf einmal ausgeführt werden, als Parameter mitgeben, etwa "step 4" oder "next 2000". Der Default-Wert ist natürlich 1.

Source-Units Neben der Anweisung gibt es noch weitere Einheiten verschiedener Größe, in die ein Programm-Text zerlegt werden kann. Der CXdb kennt fünf solcher sogenannter "Source Units":

Expression:	die kleinste Einheit, umfaßt jeweils einen Teilausdruck, aus der sich eine Anweisung zusammensetzt
Statement:	Anweisung, der Default-Wert
Block:	eine Folge von Anweisungen, die den Rumpf einer Routine, einer Schleife oder sonstiger bedingter Ausdrücke (if, case, ...) umfaßt
Loop:	eine Folge von Anweisungen, bestehend aus einem Block und einer umschließenden Schleifen-Konstruktion
Routine:	ein Unterprogramm oder eine Funktion

Um zu sehen, wie ein Programm in solche "Source Units" zerlegt wird, sehen wir uns als Beispiel die Zeile 33 in gaussj.f an. Mit dem Kommando "info line 33" erhalten wir eine Tabelle, die alle Source-Units angibt, die in dieser Zeile beginnen. In diesem Fall ergibt sich:

	Id	Address Boundaries	Start	End	Kind	
1.	(30)	8000141e:80001422	33 x 21	33 x 21	<EXPR>	1
2.	(29)	8000141e:80001428	33 x 19	33 x 21	<STMT>	K=1
3.	(31)	80001416:8000141a	33 x 23	33 x 23	<EXPR>	N
4.	(28)	80001416:8000151e	33 x 13	43 x 20	<LOOP>	DO 12 K=1,N < ... > 12 CONTINUE
5.	(27)	80001416:8000151e	33 x 13	44 x 15	<BLOCK>	DO 12 K=1,N < ... > ENDIF

Für jede Source-Unit ist folgendes angegeben: ihre Id, die Adressen im Programm, die sie umfaßt, der Bereich im Source-Code (Beginn und Ende jeweils in der Form Zeile x Spalte), die Art der Unit und der entsprechende Code. Die Größen "1" und "N" sind also Ausdrücke ("Expressions"), "K=1" ist eine Anweisung ("Statement"), die DO-Schleife ("Loop") bis zur Zeile 43 ist offensichtlich, und schließlich bildet der "THEN"-Teil der IF-Anweisung in Zeile 32 einen Block, der den Bereich der Zeilen 33 - 44 umfaßt.

Durchlaufen in Source-Units Mit step und next kann man auch in anderen Einheiten als "Statements" durch das Programm gehen. Dazu gibt man einfach die gewünschte Einheit ("Granularität") als Parameter mit an, also

```
step/next [Source-Unit] [Wiederholungsfaktor],
```

wobei Source-Unit folgende Werte haben kann: expression, statement (default), block, loop und routine. Man erreicht damit jeweils die nächste Source-Einheit der gewünschten Art im Programm, wobei bei next ein Unterprogramm-Aufruf wieder nur als eine einzige Anweisung gewertet wird, während man mit step im Unterprogramm weitermacht.

Befindet sich die Programmausführung des Programms beispielsweise in Zeile 33 der Routine gaussj, dann kommt man mit "step expression" bis vor die Auswertung der "1", mit "step statement" bis vor die Ausführung von "K=1", mit "step block" bis zum Beginn der Zeile 34 – das IF beginnt den nächsten Block –, mit "step loop" wieder an den Anfang der Zeile 33, aber diesmal mit J=2, und "step routine" verhält sich erstaunlicherweise genauso, statt zum Beginn der nächsten Routine zu springen.

Möchte man regelmäßig in anderen Einheiten als Anweisungen durch das Programm gehen, kann man die Standard-Schrittweite von "statement" auf jede andere Source-Unit setzen mit

```
set step SOURCE-UNIT.
```

Danach führen normale step- oder next-Schritte in der gewünschten Granularität weiter. Außerdem wird im Source-Fenster die Größe des schwarz markierten Bereichs entsprechend angepaßt. Statt zum Beginn der nächsten kann man auch zum Ende der aktuellen Source-Unit gehen mit dem Kommando

```
finish [SOURCE-UNIT].
```

Außerdem kann man die aktuelle Source-Unit vorgegebener Größe bis zur nächsten Einheit der Default-Größe durchlaufen mit

```
step/next over [SOURCE UNIT].
```

Damit kann man etwa mit "step over loop" die aktuelle Schleife beenden und bis zum nächsten Statement kommen (falls dies die Default-Schrittweite ist). Steht man vorher z.B. in Zeile 70 (DO-Schleife mit Label 21), kommt man damit zur Zeile 30, denn dies ist nach der Beenden der Schleife 21 und Iterieren von Schleife 22 die nächste Anweisung. Man hat damit eine große Auswahl an Möglichkeiten, in gezielten Schritten durch das Programm zu kommen, von denen man allerdings i.d.R. nur wenige häufiger benutzen wird.

Maschinen-Code Für Leute, die vor nichts zurückschrecken, gibt es natürlich auch die Möglichkeit, sich durch den Maschinencode zu hangeln. Dazu verschafft man sich am besten zunächst ein Assembler-Listing, entweder mit dem Befehl "disassemble" im Command-Fenster, oder indem man im Menü "processWindows" des Source-Fensters den Eintrag "Create Disassembly" auswählt. Dann erscheint ein neues Fenster mit dem Assembler-Listing, in dem der aktuelle Stand des Programm-Zählers durch "PC>" am Anfang der entsprechenden Zeile angezeigt wird. Dieses Fenster bietet außerdem durch Menüs weitere Möglichkeiten, z.B. kann man sich die Inhalte der verschiedenen Registertypen ansehen. Mit den Kommandos

```
step/next instruction [Faktor]
```

kann man das Programm analog zu step/next in Maschinenbefehl-Schritten durchlaufen.

3.1.3 Betrachten und Verändern von Variablen

Eine Grundfunktion jedes Debuggers ist das Ansehen und Ändern von Variablenwerten. Der CXdb bietet hierfür eine Menge Möglichkeiten, um auch größere Datenmengen (z.B. Arrayinhalte) in verschiedensten Formaten darzustellen.

Übersicht über Variable Mit den Befehlen

```
info locals  bzw.  info args
```

erhält man eine Liste aller aktuellen lokalen Variablen bzw. aller Argumente der aktuellen Routine, jeweils mit dem Typ und entweder dem Wert (für skalare Typen) oder der Adresse (für Arrays, Strukturen usw.). Möchte man auch globale Variable sehen, kann man sich mit

```
info symbols
```

eine Liste aller Symbol-Deklarationen (Variablen und Routinen) ansehen. Neben den eigenen Definitionen findet man vor allem in C-Programmen noch eine lange Liste von Symbolen, die in den eingebundenen Include-Dateien definiert wurden. Ausführlichere Informationen zu einem Symbol oder einem beliebigen Ausdruck liefert

```
info expression EXPRESSION
```

Anzeigen von Variablenwerten Mit dem Kommando

```
print EXPRESSION
```

wobei EXPRESSION ein beliebiger Ausdruck der aktuellen Sprache (Fortran oder C) ist, kann man sich die Werte von Variablen oder ganzen Ausdrücken ansehen. Fließkomma-Variablen werden defaultmäßig im Format 10.4 angezeigt. Bei Strukturen erhält man die einzelnen Komponenten, jeweils mit Typ und Wert. Von Arrays werden nur die ersten Werte angezeigt, was insbesondere bei großen Arrays sinnvoll ist; die Default-Anzahl ist 20. Teile von Arrays kann man sich ansehen, indem man mit ".." Bereiche angibt, etwa A(2..4, 2..4) (Ausschnitt eines Fortran-Arrays) oder b[0] [1..N] (1. Zeile eines C-Arrays, falls N = Zeilenlänge). Auch hierbei werden nicht mehr als die Default-Zahl von Elementen ausgegeben. Schließlich kann man sich mit dem print-Befehl auch die Werte von Debugger-Variablen ansehen. Dies sind Variable, die nicht im Programm, sondern nur im CXdb eine Bedeutung haben. Man kann sie z.B. benutzen, um Zwischenergebnisse festzuhalten oder um CXdb-Objekte wie Haltepunkte oder Fenster zu bezeichnen. Zur Unterscheidung von Programm-Variablen beginnen Debugger-Variable mit "$". Es gibt im CXdb schon eine Reihe von vordefinierten Variablen, insbesondere für die Werte der Convex-Register. Näheres dazu im Online-Help unter "debugger variables". Um sich eine Variable in einem anderen als dem Default-Format anzusehen, kann man dem print-Befehl eine Format-Anweisung mitgeben. So erhält man mit " print/f12.8 foo" 8 Nachkommastellen der Float-Variablen foo, mit "print/x foo" wird sie als Hexadezimalzahl ausgegeben. Die Liste aller möglichen Formate findet man im Online-Help unter "print".
Die Defaultwerte für das print-Kommando ändert man mit dem Kommando:

```
set printopts OPTION [Wert]
```

Folgende Optionen sind möglich:

maxarray ANZAHL:	max. Anzahl von Arrayelementen, die mit einem print-Befehl ausgegeben werden Default: 20
precision M.N :	Format für Fließkomma-Ausgaben Default: 10.4
no padding :	Legt fest, ob führende Nullen ausgegeben werden , nützlich für übersichtliche Arrayausgabe Default: nopadding

Das Kommando

```
info formatting
```

zeigt die aktuellen Werte dieser Parameter an.

Ändern von Variablenwerte Das Verändern von Variableninhalten geschieht immer dann, wenn vom CXdb Ausdrücke ausgewertet werden, die Seiteneffekte haben. Insbesondere gibt es direkt zum Auswerten den Befehl

```
evaluate EXPRESSION
```

Z.B. kann man mit "evaluate I=I+1" den Wert von I erhöhen, oder mit "evaluate \$x=2$*N*NMAX$" einen Wert in der Debugger-Variablen \$x zwischenspeichern. Da auch print sein Argument auswertet, kann man es ebenfalls zum Verändern von Variablen benutzen. So erhöht das Kommando "print I=I+1" I um eins und gibt das Ergebnis aus.

Speicherbereiche Manchmal ist es nützlich, sich ganze Speicherbereiche auf einmal ansehen zu können. Ein häufiger Anwendungsfall sind Pointer in C: Ist nämlich eine Variable als Pointer statt als Array deklariert (z.B. in der Argumentliste eines Unterprogramms), kann man im CXdb die Array-Notation nicht verwenden, um Teilbereiche von Arrays darzustellen. Stattdessen kann man sich dann einen Speicherbereich ansehen mit

```
examine[/MEMORY-UNIT FORMAT]  START..ENDE
examine[/MEMORY-UNIT FORMAT]  START:ANZAHL
```

Dabei bezeichnen START und ENDE die Anfangs- und Endadresse des gewünschten Bereichs und ANZAHL die Zahl der auszugebenden Werte. MEMORY-UNIT gibt gemäß folgender Tabelle an, wieviele Bytes ein einzelner Wert hat:

Wert von MEMORY-UNIT	Größe in Byte
b	1
h	2
w	4
l	8
q	16

FORMAT bezeichnet mit einem Buchstaben wie bei print das Ausgabeformat. Um also etwa ein Array a von 12 Werten vom Typ "double" in C auszugeben, verwendet man "examine/lf

a:12". In FORTRAN müßte man statt des Arraynamens den "Pointer" loc(a) verwenden, um die Anfangsadresse zu erhalten. Eine andere Möglichkeit, den Speicher zu durchforsten, die insbesondere für größere Datenmengen geeignet ist, bietet das Examine-Fenster. Man erhält es im "ProcessWindows"-Menü des Source-Fensters unter "create Examine". Über das "DataView"-Menü kann man hier den Anfangsbereich und die Ausgabeform auswählen sowie nach einem Wert im Speicher suchen. Schließlich kann man noch Speicherbereiche kopieren oder mit konstanten Werten füllen. Informationen dazu findet man im Online-Help unter "fill" und "copy".

Sichtbarkeit von Variablen Manchmal möchte man sich auch Variable ansehen, die im Programm an der aktuellen Stelle nicht sichtbar sind, etwa weil sie lokal bzgl. einer anderen Routine sind oder weil sie von lokalen Variablen überschrieben werden. In diesem Fall muß man Variablennamen mit dem Gültigkeitsbereich ("Scope") versehen. Die Syntax unterscheidet sich hier für Fortran und C, da sich auch die entsprechenden "Scope"-Regeln sehr unterscheiden. Eine Fortran-Variable kann man allgemein ansprechen in der Form

```
ROUTINEN-NAME`VARIABLEN-NAME
```

Befindet sich die Programm-Ausführung etwa in der Routine getvec in Zeile 21, so erhält man mit "print i" den Wert der lokalen Variablen i, bei der ersten Iteration also 1. Dagegen liefert "print getmat`i" den Wert 101, nämlich den Wert von i in getmat nach Beenden der Schleife.
In C lautet die allgemeine Syntax

```
FILE-NAME`ROUTINEN-NAME`[BLOCK-LISTE`]VARIABLEN-NAME ,
```

weil der Gültigkeitsbereich einer Variablen hier auch ein File oder aber nur einen Block umfaßen kann. Die Block-Liste ist nur nötig, falls man innerhalb von Blöcken, die kleiner sind als der Rumpf einer Routine, lokale Variable deklariert. Näheres dazu mit "help scope". Statt globale Namen für verdeckte Variable zu benutzen, kann man auch den Zustand des Stacks verändern, um aus der Tiefe der aufgerufenen Routinen wieder zurückzugehen. Als Beispiel nehmen wir an, daß sich das Programm linalg gerade in der Routine LUBKSB befinde, die von LUINV aufgerufen wurde, diese wiederum von LINALG. Im Menü "ProcessWindows" des Source-Fensters kann man durch Auswahl von "create Stack" ein Stack-Fenster öffnen, in dem die Hierarchie der aufgerufenen Routinen dargestellt wird. Nach Anklicken der gewünschten Routine erscheint ein Fenster, in dem die Werte aller Argumente und lokalen Variablen des entsprechenden Frames dargestellt sind. Wer ohne eine X-Umgebung arbeitet, kann sich die entsprechenden Informationen über Kommandos besorgen. Näheres dazu unter den Stichworten "backtrace" und "frame" des Online-Helps.

3.1.4 Haltepunkte

Die Möglichkeiten, in den Programmablauf einzugreifen, beschränken sich nicht auf das Durchlaufen in (verschieden großen) Schritten, sondern es gibt mit den verallgemeinerten Haltepunkten ("eventpoints") im CXdb ein sehr mächtiges Konzept, um gezielt bei bestimmten Ereignissen den Programmablauf anzuhalten, Variablenwerte auszugeben oder andere Aktionen auszuführen. Bevor dieses im Detail vorgestellt wird, sollen zunächst die wichtigsten Spezialfälle betrachtet werden.

Breakpoints Der häufigste Spezialfall des Eventpoints ist sicherlich der einfache Haltepunkt oder Breakpoint, der an vorbestimmter Stelle die Programmausführung anhält, wobei die Stelle durch die Zeilennummer angegeben wird. Dazu dient der Befehl

```
break line [FILENAME:]NR ,
```

wobei man mit der Angabe des Filenamens auch Zeilen in anderen als dem aktuellen File erreichen kann.
Befindet sich in der Zeile NR keine ausführbare Anweisung, wird man unter Angabe der nächsten möglichen Zeilennummer gefragt, ob man dort einen Breakpoint haben möchte. Ein gesetzter Breakpoint wird im Source-Fenster durch ein Symbol neben der Zeilennummer markiert, außerdem wird im Command-Fenster eine Meldung mit weiteren Informationen ausgegeben. Stößt das Programm bei der Ausführung auf diesen Breakpoint, wird die Ausführung vor der Zeile angehalten und eine entsprechende Meldung ausgegeben. Damit kann man gezielt bis in die Nähe der vermuteteten Fehlerquelle kommen und von dort z.B. in kleineren Schritten weitermachen.
Mit

```
info break
```

erhält man Informationen über alle gesetzte Breakpoints, darunter als erstes die Nummer, unter der ein Breakpoint angesprochen werden kann. Sie ist u.a. zum Löschen von Breakpoints erforderlich, das mit

```
remove event EVENT-NR [,EVENT-NR ...]
```

möglich ist.

Außer zu Beginn einer Zeile kann man einen Breakpoint auch an den Anfang einer Routine setzen mit

```
break routine NAME .
```

Das Programm bleibt dann nach dem Auswerten und der Übergabe der Argumente, aber vor der ersten ausführbaren Anweisung stehen.

Tracepoints und Watchpoints Ein Tracepoint verhält sich sehr ähnlich zu einem Breakpoint, nur daß die Programmausführung nicht angehalten, sondern nur eine Meldung über das Erreichen des Tracepoints ausgegeben wird. Damit kann man auf einfache Weise die Programmausführung überwachen. Die zugehörigen Kommandos sind

```
trace line/routine
info trace
remove event .
```

Ein Watchpoint überwacht im Gegensatz zu Break- und Tracepoints nicht eine spezielle Stelle im Programm, sondern im Speicher: Er hält die Programmausführung - mit einer entsprechenden Meldung - an, wenn sich der Wert einer Variablen oder eines beliebigen anderen Speicherbereiches ändert. Das ist z.B. dann nützlich, wenn sich der Wert einer Variablen "mysteriöserweise" ändert, ohne daß entsprechende Anweisungen im betrachtetetn Programmabschnitt vorkommen. Dieser Fall tritt vor allem in C gelegentlich auf, insbesondere bei falscher Pointerarithmetik oder bei String-Manipulationen mit Strings, für die zu wenig Speicherplatz bereitgestellt wurde. Allerdings führt ein Watchpoint zu einer erheblichen Verlangsamung des Programmablaufs; daher sollte man zunächst mit einem Breakpoint so dicht wie möglich an die zu überwachende Stelle herangehen und dann erst einen Watchpoint setzen (bzw. ihn dann erst aktivieren, s.u.).
Es gibt drei Möglichkeiten, den zu überwachenden Bereich anzugeben:

```
watch START
watch START..ENDE
watch START:ANZAHL .
```

Bei der ersten Version bezeichnet START einen Zeiger auf die zu überwachende Variable, etwa "watch &intvar" in C oder "watch LOC(I)" in Fortran. Das Ende des Bereichs wird durch den Typ der Variablen bestimmt. Stattdessen kann man auch explizit die Endadresse oder die Anzahl der Bytes des zu kontrollierenden Speicherbereichs angeben.
Mit "info watch" können Informationen zu allen Watchpoints angefordert werden, mit "remove event NR" wird ein Watchpoint wieder gelöscht.

Eventpoints Break-, Trace- und Watchpoints sind Spezialfälle des allgemeineren Eventpoints. Ein Eventpoint wartet auf ein Ereignis und führt bei dessen Eintreffen eine Menge von Aktionen, den sogenannten "Eventpoint Handler", aus. Je nachdem, auf welche Art von Ereignis gewartet werden soll, gibt es verschiedene Typen von Eventpoints, darunter

Typ	Ereignis
reached line	Zeile erreicht
reached routine	Routine erreicht
modify	Speicherbereichs ändert sich
relation	relationaler Ausdruck wird wahr
signal	Signal trifft ein

(eine vollständige Liste mit "help eventpoints"). Eventpoints können gesetzt werden mit dem Kommando

```
event TYP [PARAMETER ...],
```

wobei die Parameter gegebenenfalls das Ereignis näher spezifizieren (Zeilennummer, Adressbereich, Signalnummer, logischer Ausdruck, ...).
Zu jedem Typ von Eventpoints gibt es einen Standard-Handler, man kann aber für jeden Eventpoint auch selbst auf zwei Weisen einen Handler definieren. Um z.B. in der Routine LUDCMP jeweils das Pivot-Element anzuzeigen, könnte man folgenden Eventpoint setzen:

```
event reached line ludcmp.f:42   \
   {echo/n ''Pivotelement in Spalte ''; print I; print AAMAX; resume;}
```

Genausogut kann man den Handler – die Anweisungsfolge in { } – nachträglich definieren mit dem Kommando

```
set handler NR HANDLER ,
```

wobei NR die Nummer des bereits gesetzten Eventpoints ist. Einen gesetzten Handler kann man wieder löschen mit

```
clear handler NR .
```

In einem Handler können alle CXdb-Befehle vorkommen außer solchen, die die Programmausführung fortsetzen (continue, next, ...). Dafür wird das Kommando "resume" verwendet. Außerdem gibt es zur Ausgabe von Strings - wahlweise ohne abschließendes Newline - noch den Befehl "echo[/n]".
Nachdem man einen Eventpoint mühsam definiert hat, möchte man ihn nicht immer gleich löschen, wenn man ihn gerade nicht braucht. Dafür kann man einen Eventpoint auch abschalten mit

```
disable event NR
```

und später mit

```
enable event NR
```

wieder "scharf" machen.
Manchmal soll ein Eventhandler nicht gleich beim ersten Mal ansprechen, wenn das gewünschte Ereignis eintritt, sondern erst beim N. Mal (besonders bei Schleifen nützlich). Zu diesem Zweck führt jeder Eventpoint einen Zähler mit, den sogenannten Ignore-Counter, der jedesmal, wenn das Ereignis eintritt, um eins erniedrigt wird. Erst, wenn dieser Zähler 0 wird, wird der Handler ausgeführt. Man setzt ihn auf COUNT für den Eventpoint NR mit

```
set ignore COUNT NR .
```

Eine Liste aller Eventpoints mit zusätzlichen Informationen wie dem aktuellen Stand des Ignore-Zählers und dem Status (disabled/enabled) erhält man mit

```
info event .
```

3.1.5 Die CXdb-Umgebung

In diesem Abschnitt sollen einige Kommandos zusammengestellt werden, die das Arbeiten mit dem CXdb selbst und nicht den eigentlichen Debug-Prozeß betreffen.

Debuggen laufender Prozesse und Core-Files Statt einen Prozeß direkt unter Debugger-Kontrolle zu starten, kann man auch einen laufenden Prozeß oder ein Core-File mit dem CXdb untersuchen - vorausgesetzt, es wurde vorher mit der Option -cxdb übersetzt und gelinkt. Man ruft dazu den CXdb ganz normal mit "cxdb PROGRAMMNAME" im Directory auf, in dem sich das Programm befindet, damit der Bezug zu den Source-Files hergestellt werden kann.
Um einen bereits laufenden Prozeß, gegeben durch seine PID, zu debuggen, holt man ihn mit

```
attach PID .
```

Dies funktioniert auch für eigene Prozesse, die im Batchsystem laufen. Nun kann man alle normalen Debugger-Funktionen damit ausführen, insbesondere sich Variablen ansehen oder in Schritten weiterlaufen oder auch (Vorsicht!) Variablenwerte ändern. Weiß man genug, kann man den Prozeß wieder freigeben mit

```
detach .
```

Danach läuft er normal weiter. Zum einen kann man auf diese Weise Programme debuggen, deren Fehler erst nach langer Rechenzeit auftreten; man kann aber auch einfach mal nachsehen, wie weit das Programm gerade ist und ob es noch Sinnvolles rechnet.
Ein Core-File kann man sich mit dem Befehl

```
core CORE-FILE
```

ansehen, wobei CORE-FILE der Name des Core-Files ist (meist einfach "core"). Natürlich kann man es nicht zum Leben erwecken, also keine Kommandos wie next oder continue eingeben. Aber man kann sich die Werte aller Variable ansehen, auch solcher in anderen Stack-Frames. Übrigens: Damit ein Core-File überhaupt angelegt wird, muß natürlich die "coredumpsize" mit dem limit-Befehl entsprechend groß genug gewählt werden.

CXdb-Parameter Der CXdb kennt eine ganze Reihe von Parametern, mit denen einige Eigenschaften beeinflußt werden können oder die die Arbeit erleichtern. Ihre aktuellen Werte können mit dem Kommando

```
info cxdb
```

aufgelistet werden. Wir wollen uns nun einige davon ansehen. Um die Ein- und Ausgaben im Command-Fenster in Dateien mitschreiben zu können, gibt es die drei Variablen cmdout, cmderr und cmdlog, die jeweils der normalen Ausgabe, der Fehler-Ausgabe und der Eingabe entsprechen. Sie enthalten Listen der Files und Fenster, die einem Kanal zugeordnet sind. Standardmäßig enthalten alle drei nur das Command-Fenster. Mit Hilfe des "add"-Befehls können nun Log-Files hinzugefügt werden, etwa

```
add cmdout ~/LOGFILE ,
```

um die Ausgaben im File LOGFILE im Home-Directory zu protokollieren. Mit einem entsprechenden "remove"-Kommando können Files wieder aus der Liste entfernt werden, z.B.

```
remove cmdout ~/LOGFILE .
```

Während cmdout und cmderr nach dem "add"-Befehl gleich mitschreiben, muß man dies für die Eingabe (cmdlog) noch explizit starten durch

```
set logging .
```

Mit

```
clear logging
```

kann man es wieder deaktivieren, ohne den Wert für cmdlog zu verändern.
Arbeitet man mit größeren Programmen, befinden sich die Sourcen oft nicht nur in einem Directory. Um die benötigten Files finden zu können, unterhält der CXdb - ähnlich wie eine Shell - einen Such-Pfad, der zunächst nur das Direktory enthält, in dem der CXdb gestartet wurde (nämlich "."). Man kann diesen Pfad setzen, erweitern oder Eintrage entfernen mit

```
set path DIR1 [, DIR2 ...]
add path DIR
remove path DIR .
```

Da die Compiler allerdings den Pfad der Source-Datei ins Objectfile schreiben, findet der CXdb in der Regel die Sourcen automatisch.
Ebenfalls wie in vielen Shells gibt es im CXdb einen Alias-Mechanismus, mit dem man Abkürzungen für häufig benutzte Kommandos definieren kann, etwa mit

```
alias pa50 'print/f12.8 A(50..55, 50..55)'.
```

Man kann mit

```
remove alias NAME
```

eine Abkürzung wieder löschen, ansonsten gilt sie bis zum Ende der CXdb-Sitzung. Eine Liste aller Alias-Namen erhält man mit

```
info alias ,
```

darunter eine große Zahl schon vordefinierter Abkürzungen. Ein Alias muß am Anfang einer Befehlszeile stehen und kann keine Parameter enthalten. Flexibler ist der Macro-Mechanismus, über den das Online-Help unter "macro" Auskunft gibt. Der CXdb unterhält auch eine Liste der letzten 100 eingegebenen Befehle. Mit <Control>-p und <Control>-n kann man sie auf- und ablaufen und die alten Kommandos in die Kommandozeile holen, sie editieren und neu abschicken.

Kommandodateien Damit die ganzen Einstellungen, Aliasse u.ä. auch beim nächsten CXdb-Aufruf noch vorhanden sind, kann man eine Startup-Datei anlegen, in denen sie definiert werden. Diese Datei heißt .cxdbinit und befindet sich normalerweise im Home-Directory; sie kann beliebige cxdb-Kommandos enthalten. Möchte man für ein bestimmtes Programm einige Änderungen oder Ergänzungen, kann eine weitere .cxdbinit-Datei im Directory mit dem Programm anlegen. Es wird - falls vorhanden - zunächst die allgemeine Datei (~/.cxdbinit), dann die spezielle (./.cxdbinit) ausgeführt. Die Startup-Datei ist ein Beispiel für eine Kommandodatei, also eine Datei mit CXdb-Befehlen. Man kann beliebige solche Dateien anlegen und sie dann vom CXdb aus mit

```
source KOMMANDODATEI
```

ausführen. Oft erzeugt man eine solche Datei, indem man CXdb-Eingaben mitprotokolliert und das entstandene File ggf. nachträglich editiert. Kommando-Dateien sind wichtig, falls man den CXdb nicht interaktiv benutzen will, sondern eine Menge von Kommandos automatisch ablaufen lassen will ("Batch-Betrieb"). Man muß diese Datei, das zu debuggende Programm und ein File für die Ausgaben angeben und den CXdb aufrufen mit

```
cxdb -b -e PROGRAMM -f KOMMANDODATEI > AUSGABEDATEI &
```

Dabei werden auch die Startup-Dateien durchlaufen. Am Ende der KOMMANDODATEI sollte der Befehl "quit" stehen, sonst wird der CXdb nicht beendet und wartet auf weitere Eingaben.

3.1.6 Debuggen von optimiertem Code

Eine besonders interessante Eigenschaft des CXdb ist die Möglichkeit, optimierten Code debuggen zu können. Dabei sind zwei Hauptprobleme zu überwinden, die beim Optimieren entstehen:

1. Eine Source-Zeile kann mehrfach oder gar nicht im Object-Code dargestellt sein oder die Reihenfolge der Zeilen kann sich ändern.
2. Variablen im Source sind u.U. wegoptimiert worden und tauchen im Object-Code nicht mehr oder nur als Register auf; umgekehrt können neue Größen als Hilfsvariable eingeführt worden sein.

Die Grundlage zur Lösung des ersten Problems ist der Übergang von Source-Zeilen zu allgemeinen Source-Units. Insbesondere kleinere Einheiten (Expressions) oder größere Einheiten (Blocks, Routinen) sind oft besser im Object-Code wiederzufinden. Außerdem bietet das Markieren der aktuellen Source-Unit die Möglichkeit, beim schrittweisen Durchlaufen die geänderte Reihenfolge oder ein mehrfaches Auftauchen im Source-Fenster entsprechend anzuzeigen. Das zweite Problem wird dadurch gelöst, daß für jede Variable auch ein Gültigkeitsbereich im Object-Code - der eben von dem im Source-Code abweichen kann - und ein Ort (Speicherplatz oder Register) mitgeführt wird. Außerdem kennt der CXdb auch die vom Optimierer erzeugten Variablen ("synthetische Variable") und ihren Bezug zu Source-Variablen. Damit kann er – meistens jedenfalls – aus den synthetischen die Source-Variablen, die im Object-Code nicht mehr vorkommen, rekonstruieren. Trotz dieser Hilfen ist das Debuggen von optimierten Programmen mühsam und sollte nur in besonderen Fällen benutzt werden, z.B. wenn man den Optimierer als Fehlerursache im Verdacht hat – das kommt seltener vor als man denkt! – oder wenn man bereits länger laufende Prozesse debuggen will. I.f. sollen einige typische Schwierigkeiten, die beim Debuggen von skalar optimiertem und vektorisiertem Code auftreten, am Beispiel linalg vorgestellt werden. Das Debuggen parallelisierter Programme wird im Abschnitt 4.4.2 behandelt.

Skalare Optimierung Wir gehen vom Programm linalg in der Version 2.1 aus und übersetzen es mit -cxdb und -O1. Nach dem Start des Debuggers setzen wir die Schrittweite auf "expression", setzen einen Breakpoint an den Beginn der Routine GETVEC und lassen das Programm bis dahin laufen:

```
set step expression
break routine getvec
run
```

Wir sehen uns nun den Wert von I mit "print I" an und bekommen die erstaunliche Meldung "Variable's storage is not available.". Genauere Informationen über I erhalten wir mit

```
info expression I ,
```

nämlich u.a.:

```
used to create 1 synthesized variable(s):
1. <INDV>    ?i0 = (loc(R)+-8)+(8*(I-1))
```

Da keine Gültigkeitsbereiche ("liveness ranges") angegeben werden, ist die Variable I vollständig wegoptimiert worden; sie kommt im Object-Code nicht vor. Dafür wurde eine Variable ?i0 vom Optimierer erzeugt, die auf die angegebene Weise mit I zusammenhängt. Der Grund für diese Ersetzung ist klar: Statt bei jeder Iteration die Größe I zu erhöhen und dann die Stelle R(I) zu suchen, wird gleich ein Pointer in das Array hinein verwendet, der bei jeder Iteration um 8 Byte (Größe eines Elements von R) weiter zeigt. Weitere Informationen zu der synthetischen Variablen ?i0 erhält man mit

```
info expression \?i0 .
```

Als nächstes sehen wir uns mit

```
info line 20
```

die Source-Units an, die in Zeile 20 beginnen und bekommen u.a.:

```
     Id    Address Boundaries    Start       End     Kind
  2. (  3)        0:        0    20 x 13    20 x 17  <STMT>  I = 1
  3. (  5)  80001cfe:80001d02    20 x 20    20 x 20  <EXPR>  N
            80001cf6:80001cfa
            80001ce8:80001cf0
```

Zeile 2 zeigt, daß das Statement "I = 1" im Source-Code im Object-Code keine Entsprechnug hat, und wissen inzwischen auch, warum. Die nächste Zeile zeigt, daß der simple Ausdruck "N" im Objekt-Code an drei Stellen auftaucht. Mit

```
info expression N
```

sehen wir uns das genauer an:

```
object type: Fortran identifier
   location: @(@($ap+4)) <0x8008c6a8>
       size: 4 bytes
       type: INTEGER*4
      value: 100
       used to create 1 synthesized variable(s):
         1. <SEXP>    ?c1 = (loc(R)+-8)+(8*N)
       3 liveness ranges:
               Start      End         Location
         1.  0x80001cf0:0x80001cf6 - register s0
         2.  0x80001d02:0x80001d08 - register a5
         3.  0x80001ce4:0x80001d44 - @(@($ap+4))
```

Man erkennt, daß die Variable N nicht direkt in den Speicher geladen wird, sondern daß sie über den Argumentpointer **$ap** angesprochen und zusätzlich in Registern gehalten wird. Außerdem wird sie noch an anderer Stelle durch den Pointer **?c1** ersetzt, der auf das Ende des Arrays R zeigt. Um zu sehen, was passiert, wenn wir auf diese vielfältige Source-Unit "N" einen Breakpoint setzen, öffnen wir nun ein Disassemble-Fenster (Menü "ProcessWindows" im Source-Fenster) und setzen den Breakpoint auf die Source-Unit 5 (vgl. "info line 20 " -Ausgabe von oben):

```
break source 5 .
```

Tatsächlich erscheinen im Disassembler-Listing drei Breakpoints, von denen der erste an der gleichen Stelle sitzt wie der schon vorher an den Anfang der Routine GETVEC gesetzte. Diese Doppelbelegung wird durch einen "*" im Listing angezeigt.

Vektorisierung Um uns einige Effekte der Vektorisierung anzusehen, übersetzen wir linalg neu mit der Option -O2, starten den Debugger und gehen bis zum Beginn der Routine SETVEC mit

```
set step expression
break routine SETVEC
run .
```

Wir wollen nun das Array X überwachen und starten dazu vom Source-Fenster aus ein "Examine"-Fenster. Über dessen Menüs stellen wir die Adresse auf "X" und das Format auf "longword, float" und schalten die "Auto-Update"-Funktion ein, damit Änderungen automatisch angezeigt werden. Um den ersten Wert des X-Arrays zu setzen, gehen wir mit

```
next block 2
```

voran. Dies sollte zunächst den DO-Header überspringen und dann eine Zuweisung ausführen. Tatsächlich aber ist sofort das ganze X-Array (100 Werte) mit Zufallszahlen gefüllt. Das liegt daran, daß die Zuweisung der Arrays im Assembler durch einen einzigen Vektorbefehl ersetzt wird, der max. 128 Elemente auf einmal übertragen kann.
Daß die Routine aber doch noch etwas länger ist als ein paar Assemblerbefehle, wird deutlich, wenn wir uns mit "info expression I" ansehen, was aus I geworden ist:

```
Variable's storage is not available, line: 1 col: 18, I.

object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: <unknown>
       used to create 7 synthesized variable(s):
         1. <OSTR>    ?i0 = I
         2. <TRIP>    ?i1 = (-1*N)+(I-1)
         3. <TRIP>    ?i2 = N+((I-1)*-1)
         4. <ISTR>    ?i3 = ?i0+((I-1)%128)
         5. <INDV>    ?i4 = loc(Y)+((?i0-1)*8)
         6. <INDV>    ?i5 = loc(X)+((?i0-1)*8)
         7. <INDV>    ?i6 = ?i2
```

Wie zu erwarten, ist die Variable I selbst im Programm nicht mehr vorhanden, sondern durch einige synthetische Variable ersetzt, darunter die üblichen Pointer in die Arrays hinein (?i5, ?i6). Erstaunlich sind allerdings die anderen Variablen, insbesondere ?i3, die über eine Modulo-Operation (Operator %) aus I erzeugt wird. Dies sind die Auswirkungen des sogenannten "strip mining", eines Verfahrens, um die Arraylänge und die maximale Vektorlänge der Hardware in Einklang zu bringen: Die eine Schleife über I wird in zwei Schleifen zerlegt, von denen die innere über 128 Elemente geht und die äußere in 128-Blöcken weitergeht. Das "strip mining" wird noch ausführlich im Abschnitt 4.2.1 besprochen.

Allgemeine Tips Folgende Hinweise sollen helfen, beim Debuggen von optimierten Programmen die Übersicht zu behalten:

- Die normale Granularität sollte "expression" sein (set step expression).
- Breakpoints setzt man bezogen auf Source-Units statt auf Zeilen-Nummern.
- Schrittweises Durchgehen erfolgt entweder in "expressions" (ganz fein) oder in "blocks", "loops" und "routines" (grob), nicht in "statements".
- Wichtige Informationen liefern "info expression" und "info line".
- Gelegentlich ist Debuggen auf Assembler-Ebene hilfreich ("Disassemble-Fenster", "step/next instruction").

3.2 Der Convex-Performance-Analyzer CXpa

3.2.1 Die Oberfläche des CXpa

Einführung Der CXpa ist ein interaktiver Profiler, der die Fähigkeiten der bisherigen Profiler (prof, gprof, bprof) in sich vereint und einige zusätzliche Möglichkeiten bietet, insbesondere eine interaktive Oberfläche. Die Arbeitsschritte entsprechen i.w. denen herkömmlicher Profiler, nämlich Compilieren und Linken, Erzeugen eines Analysefiles und Auswerten der Analysedaten. Bis auf den ersten Schritt werden jedoch alle weiteren Kommandos direkt von der CXpa-Oberfläche aus abgesetzt. Insbesondere kann man ohne Neuübersetzung die zu überwachenden Stellen wechseln, das Programm laufen lassen und sich die neuen Daten ansehen.
Der CXpa wird aufgerufen mit

```
cpa FILENAME ,
```

(nicht cxpa !), wobei das anzugebende File normalerweise das ausführbare Programm ist. Nach einigen Meldungen erscheint der Prompt "(cpa)", an dem in normaler Text-Umgebung (ohne X11- oder sonstige Fenster) die Eingaben erfolgen. Man verläßt den CXpa wieder mit

```
quit .
```

In diesem Kapitel werden alle wesentlichen Funktionen des CXpa behandelt außer der Untersuchung von parallelen Programmen. Diese findet man im Abschnitt 4.4.3.

Tastatur-Funktionen Der CXpa kennt einige Tasten-Funktionen zur Erleichterung der Eingabe. Da sind zunächst die wichtigsten Funktionen zur Bewegung des Cursors und zum Löschen und Einfügen von Text, wie sie auch im Emacs vorhanden sind, z.B. Control-a, um an den Beginn der Eingabezeile zu kommen, oder Control-k, um von der Cursor-Position bis zum Zeilenende zu löschen. Recht hilfreich sind die Textergänzungs-Funktionen TAB und SPACE, die jeweils die Eingabe soweit vervollständigen, wie es eindeutig möglich ist. Bei Mehrdeutigkeit wird ein Warnton ausgegeben, bei TAB außerdem eine Liste der Möglichkeiten. Schließlich kann man mit ESC-p und ESC-s, in der Liste der bisherigen Kommandos auf- und abgehend, alte Befehle in die Eingabezeile holen. Eine Liste der Tastenfunktionen erhält man mit "help keys".

Online-Hilfe Mit "help" kann man sich eine Liste aller Kommandos ausgeben lassen, mit

```
help KOMMANDO
```

bekommt man zu KOMMANDO einen kurzen Erklärungstext. Schließlich kann man sich noch mit

```
help keys
```

die Liste aller Tastatur-Funktionen anzeigen lassen.

3.2.2 Erzeugung eines Analyse-Files

Compilieren und Linken Um ein Programm mit dem CXpa untersuchen zu können, muß es erst mit einer Profiler-Option übersetzt und gelinkt werden, damit entsprechende Zeitmeß-Routinen in den Code eingebaut werden. Dazu gibt es drei Optionen:

-pa	Routinen- und Schleifen-Profiling
-par	nur Schleifen-Profiling
-pab	Block-Profiling

Da das Untersuchen von Routinen und Schleifen die häufigste Anwendung ist, wird man im allgemeinen die Option -pa verwenden. Block-Profiling schließt die beiden anderen Möglichkeiten aus. Zwar erhält man bei Kombination von -pab mit einer der anderen Optionen keine Fehlermeldung, aber die gelieferten Meßwerte sind völlig falsch.
Man muß nicht das ganze Programm mit einer Profiler-Option übersetzen, sondern kann sich auf einzelne Files beschränken. Allerdings muß man dann darauf achten, daß alle Routinen, die direkt oder indirekt von einer präparierten Routine aufgerufen werden, ebenfalls entsprechend übersetzt wurden, sonst werden falsche Ergebnisse erzeugt.

Auswahl des Bereichs Ist ein ausführbares Programm für den CXpa vorbereitet, startet man mit "cpa FILENAME" den CXpa und muß nun auswählen, welche Art von Informationen man sammeln möchte. Dazu kann man sich zunächst mit dem "list"-Kommando im Source-Code ansehen, an welche Stellen man überhaupt Monitor-Punkte setzen kann. Zeilen, die den Beginn einer Routine, einer Schleife oder eines Blocks bilden, werden (falls das Programm entsprechend präpariert wurde) durch "r", "l" oder "b" markiert. Sind schon Monitor-Punkte ausgewählt, werden diese durch die entsprechenden Großbuchstaben im Listing angezeigt.
Das list-Kommando ist sehr vielseitig: Mit

```
list [VON [BIS]]
```

bzw.

```
list file FILENAME [VON] [BIS]
```

kann man sich den angegebenen Bereich des aktuellen bzw. des angegebenen Files ansehen, standardmäßig jeweils 10 Zeilen. Mit

```
list monitors ROUTINE
list monitors at VON BIS
```

werden nur die Zeilen der Routine oder des Bereichs angezeigt, die Monitor-Punkte enthalten.
Nun kann man mit dem "monitor"-Kommando Routinen, Schleifen oder Blöcke (je nach Compiler-Option) auswählen bzw. mit dem "deselect"-Kommando wieder abschalten. Mit jedem neuen "monitor"- bzw. "deselect"-Befehl werden Punkte hinzugefügt bzw. weggenommen, bis man die gewünschte Auswahl erreicht hat. Schleifen auszuwählen ist allerdings nur dann sinnvoll, wenn man mindestens mit der Optimierungsstufe **-O1** übersetzt hat, sonst sind die Meßergebnisse falsch!
Alle überhaupt im Programm vorhandenen Monitor-Punkte werden ausgewählt mit dem Kommando

```
monitor all ;
```

etwas spezieller bekommt man alle Routinen, alle Schleifen oder alle Blöcke mit

```
monitor TYPE all, TYPE = routine, loop, block
```

(oder pregion, s. 4.4.3). Einzelne Routinen kann man auswählen mit

```
monitor routine ROUTINE1 ... ,
```

alle Punkte oder alle Schleifen oder Blöcke in einer Routine mit

```
monitor ROUTINE
monitor loop in ROUTINE
monitor block in ROUTINE .
```

Ganz spezifisch kann man schließlich einzelne Punkte auswählen, indem man die Zeilennummer und ggf. den Source-File-Namen – falls nicht das aktuelle File gemeint ist – angibt:

```
monitor at [FILE:]NUMMER .
```

Zu jedem "monitor"-Kommando gibt es das zugehörige "deselect"-Kommando, mit dem man die entsprechenden Punkte wieder löschen kann. Auf diese Weise kann man z.B. mit einem globalen "monitor"-Befehl viele Punkte setzen und einzelne mit deselect wieder löschen, oder aber für einen zweiten Lauf, bei dem man an ganz anderen Werten interessiert ist, gleich viele Punkte auf einmal löschen.
Diese Vielfalt an Möglichkeiten sieht zwar beeindruckend aus, in der Praxis wird man aber meistens mit "monitor [TYPE] all" arbeiten und sich dann aus den Ergebnislisten die interessierenden Daten herauspicken.

Ausführen des Programms Nachdem ausgewählt wurde, welche Daten man haben möchte, kann man noch den Namen des Files angeben, in dem sie abgelegt werden. Defaultmäßig heißt das File mit den Analysedaten "cpa.pdf" (pdf = Performance Data File). Einen anderen Namen legt man fest mit

```
profile FILENAME .
```

Als nächstes wird mit

```
run [ARGUMENTE]
```

das Programm gestartet. Dabei sind ARGUMENTE mögliche Kommandozeilen-Parameter, die man normalerweise nach dem Programmnamen angibt. Beim Lauf werden nun die Zeiten für die ausgewählten Programmteile in das PDF-File geschrieben.
Möchte man mehrere verschiedene Datensätze erzeugen, kann man wieder mit "monitor" und "deselect" Punkte auswählen, mit "profile" einen anderen Filenamen angeben und mit "run" erneut starten. Zur Vereinfachung gibt es den Befehl

```
rerun  ,
```

mit dem der letzte run-Befehl (incl. der zugehörigen Argumente) wiederholt wird.
"run" und "rerun" erlauben noch, die Ein- und Ausgabe, wie bei der C-Shell üblich, mit "<", ">" und ">&" umzulenken.

3.2.3 Auswertung eines Analyse-Files

Allgemeines Der CXpa braucht für seine Aufgaben insgesamt drei Files: Das ausführbare Programm ("current executable"), ein Analysefile, in das neu erzeugte Analysedaten geschrieben werden ("default output PDF"), und ein File, aus dem Analysedaten gelesen werden ("current input PDF"). Mit

```
status
```

kann man sich ihre momentanen Namen anzeigen lassen. Hat man, wie oben beschrieben, Analysedaten mit einem "run" erzeugt, wird das Ausgabefile automatisch zum neuen Eingabefile und kann gleich ausgewertet werden. Man kann aber auch ein anderes Eingabefile wählen mit

```
analyze INPUT-PDF .
```

Schließlich kann man den CXpa auch später direkt für die Analyse alter Daten aufrufen mit

```
cpa INPUT-PDF .
```

Da dabei kein Programm angegeben wird, kann man weder "run"- noch "monitor"-Befehle eingeben, sondern nur das Eingabefile auswerten.
Der letzte Schritt, das Auswerten der PDF-Daten, geschieht mit den verschiedenen Formen des analyze-Befehls, die i.f. vorgestellt werden und die alle die Ausgabe-Umlenkung mit ">" erlauben. Je nach Kommando werden Meßwerte für Routinen, Schleifen oder Blöcke ausgegeben (oder für parallele Bereiche, s. Abschnitt 4.4.3). Mit einem einfachen

```
analyze
```

erhält man alle vorhandenen Daten auf einmal. Man beginnt i.d.R. mit der Routinen-Analyse, um zunächst einmal herauszufinden, wo die meiste CPU-Zeit verbraucht wird und sich daher Optimierungsanstrengungen lohnen. Genauere Informationen, wie sich die CPU-Zeit in einer Routine verteilen, gewinnt man durch die Schleifen-Analyse. Wenn man noch mehr ins Detail gehen will, kann man mit der Block-Analyse noch genauer sehen, welche Teile einer Routine wie oft durchlaufen wurden und wieviel CPU-Zeit dabei verbraucht wurde.

Die CPU-Zeiten werden in Sekunden angegeben oder, falls sie zu klein sind, in Millisekunden, was durch ein nachgestelltes "m" angezeigt wird. CXpa zieht automatisch die Zeiten ab, die zum Aufnehmen der Meßwerte benötigt wurden, so daß die Werte i. W. die tatsächlichen Verhältnisse (ohne störende Messung) darstellen.

Analysieren von Routinen Mit dem Kommando

```
analyze routine
```

erhält man drei Tabellen, die wir uns an unserem Beispielprogramm LINALG in der Version 2.1 und der Optimierungsstufe -O2 ansehen wollen:

Routine Performance Analysis
(sorted by CPU time (less children))

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
0.627	57.7%	0.627	57.7%	1	gaussj
0.188	17.3%	0.188	17.3%	2	testmt
0.127	11.7%	0.127	11.7%	101	lubksb
0.071	6.5%	0.071	6.5%	1	ludcmp
0.038	3.5%	0.038	3.5%	10100	drand_$n
0.029	2.6%	0.066	6.1%	1	getmat
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Die erste Tabelle zeigt für jede überwachte Routine die CPU-Zeiten, die alle Aufrufe dieser Routine ohne bzw. mit von ihr aufgerufenen Unterroutinen verbraucht haben sowie die Zahl der Aufrufe. Sortiert wird nach der CPU-Zeit der Routine selbst. Neben eigenen Routinen werden auch alle Bibliotheksroutinen aufgeführt, im Beispiel etwa die Routine "drand_$n" zum Erzeugen von Zufallszahlen. Bei der Routine "getmat" sieht man deutlich den Unterschied der ersten beiden Spalten: In "getmat" wird die Funktion "drand" aufgerufen, die selbst wieder "drand_$n" aufruft, von der der größte Teil der CPU-Zeit von getmat incl. Unterprogrammen stammt.

Routine Performance Analysis
(sorted by CPU time (less children))

CPU Time (less children)			CPU Time (plus children)			
Min.	Max.	Avg.	Min.	Max.	Avg.	Routine name
0.627	0.627	0.627	0.627	0.627	0.627	gaussj
0.092	0.096	0.094	0.092	0.096	0.094	testmt
0.813m	1.830m	1.259m	0.813m	1.830m	1.259m	lubksb
0.071	0.071	0.071	0.071	0.071	0.071	ludcmp
0.003m	0.021m	0.003m	0.003m	0.021m	0.003m	drand_$n
0.029	0.029	0.029	0.066	0.066	0.066	getmat
.	.	.	.	.	.	
.	.	.	.	.	.	

Während die erste Tabelle summarische Informationen über alle Aufrufe einer Funktion enthält, kann man an der zweiten ablesen, wie sich die Zeiten bei mehrfachem Aufruf einer Routine verteilen. Dazu wird für jede Routine der kleinste, größte und der Durchschnittswert der CPU-Zeit eines Aufrufs angegeben, jeweils ohne bzw. mit von ihr aufgerufenen Unterprogrammen. So sieht man z.B., daß die Aufrufe für die Routine "lubksb" für verschiedene Eingabe-Parameter recht stark streuen und könnte dies möglicherweise benutzen, um für verschiedene Parameterwerte unterschiedliche Optimierungen zu finden.

```
                Dynamic Call Graph
   (in topological order, cycles severed)

   .
   .
```

```
f_exit:
  parents:  (spontaneous call)
  children: fclose[3] for$clos2[3]

for$clos2:
  parents:  f_exit
  children: free[3] t_runc[2]
  .
  .
MAIN_:
  parents:  main
  children: for$do_lio[2] for$s_wsle[1] ernorm[1] testmt[1] gaussj[1]
            setvec[1] setmat[1] getvec[1] getmat[1]

getmat:
  parents:  MAIN_
  children: drand_$n[10000]

getvec:
  parents:  MAIN_
  children: drand_$n[100]

drand_$n:
  parents:  getvec getmat
  .
  .
```

Nach einer Liste aller Routinen, die zum Profilen vorbereitet waren, aber nicht aufgerufen wurden (vor allem eine Menge von Bibliotheksfunktionen) folgt der "dynamische Aufruf-Graph". Ein Aufruf-Graph ("Call Graph") ist eigentlich eine Graphik, die in Baumform darstellt, welche Routine von welcher aufgerufen wird. Man unterscheidet zwischen dem statischen und dem dynamischen Aufruf-Graphen. Ersterer kann aus dem Programmtext abgeleitet werden, letzterer dagegen wird erst durch einen Programmlauf bestimmt. Um sich den Unterschied klarzumachen, denke man sich eine Funktion, die nach einer IF-Bedingung aufgerufen wird: Der statische Graph enthält sie immer, der dynamische nur dann, wenn die Bedingung zur Laufzeit erfüllt war. Da jedoch der CXpa (noch ?) keine graphische Schnittstelle hat, wird der Graph in Form einer Liste ausgegeben: Von der Spitze ausgehend werden die Routinen mit allen von ihnen direkt aufgerufenen Unterroutinen aufgelistet. In Klammern wird jeweils angegeben, wie oft eine Routine aufgerufen wurde. Rekursionen, d.h. Schleifen im Graphen, werden bei der ersten beteiligten Routine mit "start of cycle" gekennzeichnet.

Im Beispiel kommen zunächst einige Start-Routinen, etwa f_exit, bis das eigentliche Hauptprogramm aufgerufen wird, das vom Fortran-Compiler immer "MAIN_" genannt wird. Man sieht auch, daß der Compiler die "DRAND"-Funktion in einen Aufruf von "drand_$n" verwandelt hat.

Analysieren von Schleifen Meßergebnisse für alle ausgewählten Schleifen einer Routine erhält man mit

```
analyze loop [ROUTINE] ,
```

ohne Routinennamen für alle Routinen. Sie werden in zwei Tabellen dargestellt. Da der Compiler Schleifen erst ab der Optimierungsstufe -O1 erkennt, machen Schleifenanalysen für niedrigere Optimierung keinen Sinn und liefern falsche Werte. Als Beispiel sehen wir uns die Ergebnisse für die Routine "lubksb" an:

Loop Performance Analysis
For lubksb.f:lubksb

Line	Times	Iteration Count			
Number	Exec.	Min	Max	Avg	Total CPU Time
32	101	100	100	100	0.049237
37	5049	1	99	33	0.018813
45	101	99	99	99	0.077001
48	9999	1	99	50	0.047125

Für jede Schleife, gegeben durch die Nummer der Zeile, in der sie beginnt, ist dargestellt, wie oft sie aufgerufen wird, wie oft sie mindestens, höchstens und im Mittel durchlaufen wurde und wieviel CPU-Zeit insgesamt dabei verbraucht wurde.

Loop Transformation Performance Analysis
For lubksb.f:lubksb

Line	Trans	Times	Iteration Count			
Number	Formation	Exec.	Min	Max	Avg	Total CPU Time
32	0.S	101	100	100	100	0.030424
37	1.SM	5049	1	99	33	0.018813
45	0.S	101	99	99	99	0.029876
48	1.SM	9999	1	99	50	0.047125

Die zweite Tabelle zeigt in der zweiten Spalte vor dem Doppelpunkt die Verschachtelungstiefe einer Schleife an, wobei äußere Schleifen die Tiefe 0 haben. Die folgenden Buchstaben bezeichnen die Umformungen, die der Vektorisierer an der Schleife vorgenommen hat, z.B. "S" für eine skalare Schleife (d.h. unverändert), "SM" für sogenanntes "strip mining". [10]
Die letzte Spalte enthält im Gegensatz zur vorigen Tabelle die CPU-Zeiten innerhalb einer Schleife fester Tiefe, d.h. ohne die Zeit, die in einer vollständig umschloßenen Schleife verbraucht wird. Alle anderen Einträge sind mit denen der ersten Tabelle identisch.

Analysieren von Blöcken Die Daten eines Laufs mit Messungen auf Block-Ebene bekommt man mit

```
analyze block [ROUTINE] ,
```

wieder für eine bestimmte Routine oder für alle. Für die Routine "testmt" erhält man beispielweise:

[10] Diese Umformungen werden ausführlich im Abschnitt 4.2.1 besprochen.

Basic Block Performance Analysis
(sorted by line number)
For testmt.f:testmt

Pc Value	Block Line #	Times Executed	
0x80002888	24	2	0.0%
0x800028f0	28	200	0.2%
0x800028aa	28	2	0.0%
0x8000293c	29	20000	16.6%
0x80002908	29	200	0.2%
0x80002972	31	20000	16.6%
0x80002958	31	20000	16.6%
0x800029a0	34	20000	16.6%
0x800029ba	35	200	0.2%
0x800029da	37	19800	16.4%
0x800029f0	39	20000	16.6%
0x80002a1e	41	200	0.2%
0x80002a50	46	2	0.0%
0x80002ad4	50	2	0.0%

Blocks Not Executed
For testmt.f:testmt

Pc Value	Block Line #
0x80002a6a	47

Für jeden Block wird seine Adresse und Zeilennummer angegeben, wie oft er durchlaufen und wieviel CPU-Zeit dabei verbraucht wurde. Ausserdem folgt eine Liste aller Blöcke, die gar nicht ausgeführt wurden, wie hier z.B. die Ausgabe im Fehlerfall in den Zeilen 47 und 48. Die Aufschlüsselung ist sehr fein, man kann sie fast als Histogramm neben das Listing legen. Einige Zeilen enthalten sogar mehr als einen Block, etwa Zeile 28: Die Initialisierung des Schleifenzählers bildet einen Block, der nur zweimal durchlaufen wird. ("testmt" wird zweimal aufgerufen.) Das Inkrementieren und die Prüfung, ob der Endwert erreicht wird, bilden den zweiten Block; er wird $2 * N$ mal ausgeführt.

3.2.4 Die CXpa-Umgebung

CXpa-Parameter Der CXpa unterhält einen Such-Pfad für Source-Files, den man sich mit

```
use
```

ansehen kann. Mit

```
use DIR1 ...
```

kann man ihn umsetzen, mit

```
use add DIR1 ...
```

kann man den bestehenden Pfad erweitern. Außerdem kann man den Pfad schon beim Aufruf des CXpa mit der Option "-u DIRECTORY" erweitern (ggf. mehrfach).

Schließlich gibt es im CXpa noch zwei verschiedene Default-Listing-Files, genannt “default source file” und “default listing file”. Die Unterscheidung ist allerdings wenig hilfreich und höchst verwirrend. Sollte bei einem “list”-Kommando einmal ein anderes als das gewünschte File erscheinen, sollte man den Filenamen einfach explizit angeben.

Kommando-Dateien Natürlich gibt es auch beim CXpa die Möglichkeit, Kommandos in Dateien zu schreiben, die dann vom CXpa aus mit

```
source KOMMANDO-DATEI
```

aufgerufen werden können. Sie können Kommentare enthalten, die von einem “#” bis zum Ende einer Zeile gehen. Eine Datei mit dem Namen “.cpainit” im aktuellen oder im Home-Directory wird als Kommando-Datei interpretiert und beim Start automatisch ausgeführt. Außerdem kann man dem CXpa gleich beim Aufruf eine Kommando-Datei mitgeben mit

```
cpa -x KOMMANDO-DATEI PROGRAMM   ,
```

was insbesondere dann nützlich ist, wenn wegen der langen Laufzeit des “run”-Kommandos der CXpa im Batch laufen soll. Die Kommando-Datei sollte dann zum Schluß einen “quit”-Befehl enthalten, sonst geht der CXpa in den interaktiven Mode über. Die folgende Beispieldatei ist für die meisten Fälle ausreichend:

```
monitor all
run [ARGUMENTE]
analyze
quit
```

3.3 Application Compiler

3.3.1 Arbeitsweise des Application-Compilers

Im Gegensatz zu bisherigen optimierenden Compilern, die Programme auf der Ebene von Blöcken, Schleifen und höchstens Routinen analysieren, untersucht der Application-Compiler ein vollständiges Programm. Dazu legt er eine umfangreiche Datenbasis im Directory PDB an ("Program DataBase"), die einen Umfang von etwa 1 - 1.5 kB pro Source-Zeile hat. Mit diesen Informationen versehen, kann er nun zum einen Fehler finden, die anderen Compilern verborgen bleiben. Dazu gehören Routinenaufrufe mit einer falschen Anzahl oder falschen Typen von Parametern, mehrfach oder gar nicht initialisierte globale Variable und in manchen Fällen auch Index-Überschreitungen bei Array-Zugriffen. Zum anderen werden Optimierungen automatisch möglich, die man vorher nur durch zusätzliche "Handarbeit" erreichen konnte. Diese wollen wir uns i.f. etwas genauer ansehen.
Eine Optimierung, die schon ab der Stufe -O0 durchgeführt wird, ist das Weiterreichen von Konstanten ("constant propagation"). Dabei wird eine Variable, die auf einen konstanten Wert gesetzt und i.f. nicht verändert wird, durch diesen Wert ersetzt. Dieser Prozeß kann beliebig iteriert werden, allerdings nicht über die Grenzen einer Routine hinaus. Der Application-Compiler dagegen reicht Konstanten beliebig tief durch Routinen-Aufrufe hindurch, sowohl nach oben als auch nach unten. Dies ermöglicht dem nachfolgenden Vektorisierer u.U. eine ganze Menge zusätzlicher Optimierungen, z.B. im folgenden Beispiel:

```
      PROGRAM FOO

      REAL*4 A(1000, 10)
      REAL*4 B(10, 1000)

      CALL TWICE(A, 1000, 10)
C
C     CALL TWICE(B, 10, 1000)
C
      END


      SUBROUTINE TWICE(A, M, N)

      REAL*4   A(M, N)
      INTEGER  M, N
      INTEGER  I, J

      DO J=1, M
        DO I=1, N
           A(I, J) = 2 * A(I, J)
        ENDDO
      ENDDO

      END
```

Normalerweise würde der Vektorisierer die Schleifen nicht verändern, da sie schon so geordnet sind, daß auf aufeinanderfolgende Speicherplätze auch direkt hintereinander zugegriffen wird. Durch das Weiterreichen der konstanten Arraygrößen wird jedoch klar, daß es viel effizienter ist, die lange

statt der kurzen Schleife zu vektorisieren und dafür etwas langsameren Speicherzugriff in Kauf zu nehmen. Daher werden die Schleifen vertauscht.
Ein großes Problem bei der Vektorisierung ist das mögliche Überlappen von Arrays, das sogenannte "Aliasing".[11] In C kann es wegen der Möglichkeit von Pointern an verschiedenen Stellen auftreten; daher ist der Optimierer gezwungen, den schlimmsten Fall anzunehmen und vektorisiert viele Schleifen nicht, obwohl es möglich wäre. In FORTRAN dagegen kommt Aliasing vor allem vor, wenn ein Array mehrmals als Parameter in einem Routinenaufruf benutzt wird. Da dies aber vom FORTRAN77-Standard explizit verboten wird, geht der Optimierer auch davon aus, daß Array-Parameter in Unterroutinen zu verschiedenen Speicherbereichen gehören und vektorisiert entsprechend. Hat man sich nicht an den Standard gehalten, führt dies daher normalerweise zu fehlerhaftem Programmverhalten. Der Application-Compiler verfolgt alle Zeiger und Arrays durch das ganze Programm und kann daher in einem Fall Fehler vermeiden, im anderen die Vektorisierungsmöglichkeiten sehr verbessern.
Eine weitere Optimierung ist das "Inlining" von Routinen, d.h. das Ersetzen eines Funktionsaufrufs durch den Code der Routine, natürlich unter Anpassung der Parameter. Dies kann man beim Fortran-Compiler manuell erreichen, beim C-Compiler gibt es eine solche Möglichkeit nicht. Der Vorteil besteht nicht nur darin, daß der Overhead für einen Funktionsaufruf entfällt, sondern es können sich dadurch ggf. auch neue Optimierungsmöglichkeiten ergeben. Nachteil ist die dadurch wachsende Programmgröße. Der Application-Compiler wägt Aufwand und Nutzen gegeneinander ab, wobei er für jede Routine ihre Größe und die Zahl der Aufrufe berücksichtigt, und führt dementsprechend ein Inlining geeigneter Routinen durch. Durch Kommandozeilen-Parameter oder Direktiven kann man die Kostenrechnung beeinflußen und selteneres oder häufigeres Inlining ereichen (s. Abschnitt 3.3.5).
Manchmal hängt es von den Parametern einer Routine ab, wie sie sich am besten optimieren läßt. Im obigen Beispielprogramm etwa würde man die Reihenfolge der beiden Schleifen der Routine TWICE von dem Verhältnis der Parameter M und N abhängig machen wollen. Kommen aber verschiedene Parameter vor, wie z.B. wenn man den zweiten Aufruf von TWICE entkommentiert, ist nicht klar, was der Compiler am besten tun soll. In einem solchen Fall legt der Application-Compiler eine oder mehrere Kopien einer Routine an, sogenannte "Clones", und optimiert jede anders.
Eine weitere wichtige Optimierung dient der Verringerung von Speicherbank-Konflikten: Hat ein Array (mindestens) eine Dimension, die ein Vielfaches der Anzahl der Speicherbänke beträgt, vergrößert der Application-Compiler diese Dimension um eins, ähnlich wie wir das im Beispiel-Programm linalg gemacht haben (vgl. 2.9.1).
Um seine Entscheidungen, z.B. über den Nutzen von Inling oder Cloning, zu verbessern, kann der Application-Compiler auch Profiling-Daten benutzen, allerdings z.Z. nur solche der Programme "prof" und "gprof". Eine Anpassung an den CXpa ist geplant.
Ein häufiges Verhalten ist, daß durch eine der neuen Optimierungen des Application-Compilers weitere Möglichkeiten zur Optimierung entstehen und so das Laufzeitverhalten eines Programms verbessert wird. In unserem Beispiel werden durch das Clonen von TWICE zwei Routinen erzeugt, die jeweils nur einmal mit konstanten Werten aufgerufen werden. Beiden Versionen können daher die Konstanten weitergegeben werden. Dies wiederum ermöglicht das optimale Anpassen der Schleifen an die gegebenen Werte.

3.3.2 Meldungen des Application Compilers

Der Application-Compiler erzeugt normalerweise während des Laufs eine Menge von Ausgaben: Zunächst erscheinen Meldungen, die über den Stand des Compilierungsvorgangs Auskunft geben, dazwischen kommen die Optimierungsreports für die einzelnen Routinen, den Schluß bildet der

[11] Beispiele für Aliasing s. Abschnitt 4.2.2.

sogenannte IPO-Report ("InterProcedural Optimization"), der aus zwei Tabellen besteht.
Die Optimierungsreports sind i.w. die gleichen, die die Compiler auch sonst erzeugen, allerdings sind sie um die neu hinzugekommenen Optimierungen ergänzt. Außerdem werden auch Routinen berücksichtigt, die geklont oder eingefügt ("inlined") wurden. Näheres zum Optimierungsreport findet man in Abschnitt 4.2.3. Der IPO-Report gibt in Tabellenform einen Überblick über die vom Application-Compiler durchgeführten Optimierungen und über die gefundenen Fehler. Für das Programm linalg in der Version 2.1 sieht die Optimierungstabelle folgendermaßen aus:

Optimizations performed

Procedure	Times Procedure Inlined	Times Procedure Cloned	Propagated Constants Used	Pointer Variables Renamed	Arrays Resized
LINALG					
LUINV			5		
GAUSSJ			15		
GETMAT			3		
GETVEC		1	1		
ERNORM			3		
SETVEC		2	1		
SETMAT			3		
TESTMT			4		
LUDCMP			11		
LUBKSB			6		
Totals		3	52		
build -show	inline	clones	constants	pointers	resizes

Die ersten drei Spalten geben an, wie oft welche Funktionen eingefügt oder geklont wurden oder wieviele weitergereichte Konstanten benutzt wurden. Die letzte Zeile gibt Informationen darüber an, wieviele Pointervariablen (insbesondere in C) umgenannt wurden, um ein "Aliasing" (s.o.) zu verhindern. Möchte man genauere Informationen zu den einzelnen Punkten, kann man mit der Option "-show TOPIC" des "build"-Kommandos weitere Informationen erhalten, wobei in der letzten Zeile angegeben ist, welcher Wert von TOPIC Details für die entsprechende Spalte liefert. So erhält man mit "build -show constants" u.a.:

```
Constants on entry to procedure ‘LUBKSB’: N=100, NP=100
Compilation: Optimization: Propagated IPO constant on entry to ‘LUBKSB’,
                           NP = 100
```

Es gibt darüberhinaus noch weitere Möglichkeiten für TOPIC, die wichtigste ist "all", womit man alle TOPICs bekommt. Der Application-Compiler schreibt unabhängig von der "-show" -Option alle Informationen in ein Logfile, das man sich später auch ohne weiteres Neucompilieren mit dem Kommando

```
build -display
```

ansehen kann. Die Fehlertabelle von linalg in der Version 1.1 sieht so aus:

Errors Detected

Procedure	Mis-Matched Args	Wrong Mis-Number Of Args	Matched Return Type	Scalar Invalid Aliases	Passed To Array	Invalid SubScript	Variables Not Initialized
LINALG	6						
LUINV	2						
GAUSSJ							
GETMAT							
GETVEC							
ERNORM							
SETVEC							
SETMAT							
TESTMT							
LUDCMP							
LUBKSB							
Totals	8						
build -check	types	types	types			arrays	init

Auch hier gibt es wieder die Möglichkeit, sich über eine build-Option genauere Informationen zu den Fehlern ausgeben zu lassen; die entsprechende Option lautet "-check TOPIC". Es ist wieder "all" als TOPIC möglich für eine vollständige Liste. Außerdem enthält die Ausgabe von "build -display" immer auch alle Fehler-Informationen, darunter z.B. im obigen Beispiel folgende Zeilen:

```
Warning: Argument number 1 of GAUSSJ has inconsistent type in ./main.f
   on line 57 and ./gaussj.f on line 1
Warning: Argument number 4 of GAUSSJ has inconsistent type in ./main.f
   on line 57 and ./gaussj.f on line 1
Call to GAUSSJ on line 57 of ./main.f:
  VOID ((REAL*8 *), (INTEGER*4 *), (INTEGER*4 *), (REAL*8 *),
        (INTEGER*4 *), (INTEGER*4 *))
Defn of GAUSSJ on line 1 of ./gaussj.f:
  VOID ((REAL*4 *), (INTEGER*4 *), (INTEGER*4 *), (REAL*4 *),
        (INTEGER*4 *), (INTEGER*4 *))
```

Möchte man den Application-Compiler nur zur Fehlersuche benutzen, sollte man ihn mit der Option "-check only" aufrufen, dann werden keine Optimierungs- und Übersetzungsschritte ausgeführt. Schließlich kann man sich mit der build-Option "-time" anzeigen lassen, wie lange der Application-Compiler für die einzelnen Phasen gebraucht hat:

Summary of Time Usage (HR:MIN:SEC)

Summary phase	00:00:00
Synthesis phase (pass 1)	00:00:01
Analysis phase	00:00:11
Synthesis phase (pass 2)	00:00:00
Compilation phase	00:00:16
Link phase	00:00:02
Total build time	00:00:33

Alle angegebenen Zeiten sind "real"-Zeiten, keine CPU-Zeiten.
Für X-Window-Benutzer gibt es seit der APC-Version 1.1 auch die Möglichkeit, sich mit dem Programm pdbview Informationen aus der Datenbasis PDB anzeigen zu lassen. Auf diese Weise kann man sich u.a. alle Meldungen ansehen, wahlweise zusammen mit dem dazugehörigen Source-Text, oder eine Übersicht über den Aufbau einer Routine, insbesondere der Loops, und den gesamten Aufruf-Graphen (diesmal wirklich als Graphen) anzeigen lassen. Es wird einfach mit

```
pdbview &
```

aufgerufen und arbeitet mit Menüsteuerung. Die Menüpunkte sind i.w. selbsterklärend; weitere Fragen lassen sich durch Ausprobieren oder mit der eingebauten "Help"-Funktion schnell beantworten.

3.3.3 Das "build"-File

Die Schnittstelle zum Application-Compiler ist das "build"-Kommando, das mit einem Buildfile zusammenarbeitet, ähnlich wie das "make"-Programm. Es kennt eine ganze Reihe von Optionen, darunter:

-c	löscht die Datenbasis, wichtig nach Fehlern
-f FILE	setzt einen anderen Namen für das Buildfile, Default: buildfile oder Buildfile
-path DIR	wählt ein anderes Verzeichnis für die Datenbasis, Default: ./PDB
-o NAME	setzt den Namen des Programms, Default: Name des aktuellen Verzeichnisses
-n	gibt die Files aus, die aufgrund von Änderungen seit dem letzten Übersetzen neu kompiliert werden würden, tut aber nichts (ähnlich zu make -n).

Warnung: Die aktuelle Version des Application-Compilers gibt zwar wirklich nur die Files aus, die sich geändert haben, kompiliert bei einem "build" dann aber trotzdem alles neu!
Weitere Optionen dienen zur Steuerung der Meldungen (s. 3.3.2) und zur Beeinflussung der Optimierungen (s. 3.3.5).
Ein Buildfile ist sehr einfach aufgebaut, denn im Gegensatz zum Makefile brauchen keine Abhängigkeiten der Source-Files untereinander angegeben zu werden, die findet der Application-Compiler automatisch. Folgende Zeilen können beliebig oft und in beliebiger Reihenfolge in einem Buildfile vorkommen:

- Kommentarzeile:

  ```
  # TEXT
  ```

  wird ignoriert.

- Leerzeile:

  wird überlesen.

- Directory-Zeile:

  ```
  DIRECTORY [COMPILER-OPTIONEN]
  ```

  alle Source-Files im DIRECTORY (oft einfach "."), die nicht explizit in eigenen Sourcefile-Zeilen aufgeführt sind, werden mit den COMPILER-OPTIONEN übersetzt. Die Optionen "-c", "-il" und "-is" dürfen nicht verwendet werden. Vorsicht: Befinden sich in einem aufgeführten Directory Source-Files, die nicht zum Programm gehören, kann es Probleme geben!

- Source-Zeile:

  ```
  SOURCEFILE [COMPILER-OPTIONEN]
  ```

  SOURCEFILE wird mit COMPILER-OPTIONEN übersetzt, die ggf. in Directory-Zeilen angegebene Optionen überschreiben.

- Link-Zeile:

  ```
  link [SPRACHE] [OBJECTFILES] [LIBRARIES] [LOADER-OPTIONEN]
  ```

  SPRACHE ist entweder "FORTRAN" oder "C", je nach verwendeter Sprache. Bei gemischter Fortran-C-Programmierung ist "FORTRAN" zu verwenden. Neben zusätzlich zu bindenden Objectfiles und Bibliotheken werden Linker-Optionen angegeben, etwa -pa, -cxpa, aber nicht -o (s.u.).

- Optionen-Zeile:

  ```
  options [OPTIONEN]
  ```

  Mit der Optionen-Zeile können Kommandozeilen-Optionen des "build"-Kommandos direkt ins Buildfile geschrieben werden, darunter auch die "-o"-Option zur Benennung des Programms.

- Bibliotheks-Zeile:

  ```
  apc_libraries APCLIB1 [APCLIB2 ...]
  ```

  Mit dieser Zeile können eigene, vom Application-Compiler erzeugte Bibliotheken eingebunden werden (s.u.).

- Optimierungs-Kontrollzeilen:

  Solche Zeilen beeinflußen das Optimierungsverhalten des Application-Compilers. Näheres dazu im Abschnitt 3.3.5.

3.3.4 Benutzen von Bibliotheken

Die Fähigkeiten des Application-Compilers zur Programm-Analyse versagen natürlich, wenn nicht der ganze Source-Code gegeben ist, weil man Objekt-Code aus anderen Quellen in sein Programm einfügt, etwa Bibliotheken, vorübersetzte Object-Files oder auch Assembler-Programme. Für diesen Fall gibt es die Möglichkeit, die Informationen, die der Application-Compiler über eine Routine benötigt, direkt in ein Objekt-File einzufügen. Dazu schreibt man die Eigenschaften der Routinen

in ein File, das sogenannte Psum-File ("Procedure SUMmary File"), und fügt sie dann mit Hilfe des Programms "annotate" in den Object-Code ein. Die Standard-C- und -Fortran-Bibliotheken sowie die Veclib sind schon von Convex entsprechend ausgestattet ("annotiert") worden, so daß man diese Routinen problemlos benutzen kann.
Dieses Verfahren ist aber nicht nur umständlich, sondern auch fehleranfällig: Eine falsche Anweisung im PSUM-File kann bewirken, daß durch eigentlich unzulässige Optimierungen fehlerhafte Programme entstehen. Besser läßt man diese Arbeit vom Application-Compiler ausführen. Dazu übersetzt man die Sourcen der Bibliothek wie ein gewöhnliches Programm, gibt aber beim "build"-Kommando (oder in der "options"-Zeile) zusätzlich die Option "-library" an. Statt eines ausführbaren Programms wird dann eine Bibliothek (mit der Endung .apclib) erzeugt, die die zur Optimierung nötigen Informationen enthält.
Möchte man Routinen aus solchen Bibliotheken in einem anderen Programm verwenden, gibt man sie einfach in einer speziellen Buildfile-Zeile an:

```
apc_libraries lib1.apclib lib2.apclib ...
```

Das Programm kann nun genauso gut optimiert werden, als würde man die entsprechenden Routinen direkt mit übersetzen, insbesondere ist ein Inlinig möglich.
Falls man eine nicht-annotierte Bibliothek verwenden will, muß man den Application-Compiler dazu überreden, indem man "build" mit der Option "-permit unannotated" aufruft. Der Compiler kann für entsprechende Routinen nur das Schlimmste annehmen (alle Parameter und alle globalen Variablen werden geändert), daher sind die Optimierungs- und Fehlerprüfmöglichkeiten stark eingeschränkt.

3.3.5 Manuelle Kontrollmöglichkeiten des Application-Compilers

Zwar arbeitet der Application-Compiler normalerweise automatisch, es kann aber Fälle geben, wo man mehr oder weniger Funktionen inlinen oder klonen möchte. Insbesondere wenn man a priori Informationen über die Häufigkeit eines Routinen-Aufrufs hat, kann es sinnvoll sein, in den Optimierungsprozeß einzugreifen.
Um global die Häufigkeit des Inlining zu beeinflußen, kann man die Kostengrenze verändern, die der Application-Compiler bei der Auswahl von Funktionen zum Inlinen benutzt. Dazu gibt es eine Option des "build"-Kommandos:

```
build -inline DEGREE ,
```

wobei DEGREE die Werte "none", "low", "medium" und "high" annehmen kann. Der Standardwert ist "medium". In unserem Beispielprogramm "linalg" wird bei "low" die Funktion GETVEC, die nur einmal aufgerufen wird, nicht mehr eingefügt, wie sonst bei "medium". Die Option "high" bewirkt hier keine Veränderungen.
Man kann auch angeben, daß bestimmte Funktionen immer oder nie eingebaut werden sollen, indem man entweder im Buildfile eine Zeile

```
inline(FUNKTION1 [, ...])
```

bzw.

```
no_inline(FUNKTION1 [, ...])
```

vorsieht, oder direkt in den Sourcecode die Compilerdirektive INLINE bzw. NO_INLINE einfügt:

```
Fortran:
C$DIR INLINE
```

nach der SUBROUTINE- bzw. FUNCTION-Zeile

```
C:
/*$dir inline*/
```

nach der Deklaration von lokalen Variablen.
Mit der Direktive NO_INLINE_CALL bzw. INLINE_CALL im Source-Code direkt vor einem Funktionsaufruf kann man auch ganz spezifisch für diesen Aufruf ein Inlinen verhindern oder erzwingen. Auch das Erzeugen von Routinen-Klons kann man steuern: Mit der Build-Option "-clone none" verhindert man das Klonen völlig, mit "-clone all" wird für jede Routine, in die eine Konstante hineingezogen werden kann, ein eigener Klon erzeugt. Für einzelne Routinen kann man, wie beim Inlinen, entweder mit einer Buildfile-Zeile

```
clone(FUNKTION1 [,...])
no_clone(FUNKTION2 [,...])
```

oder mit einer Compiler-Direktive CLONE / NO_CLONE im Source-Code hinter der Deklaration das Klonen beeinflußen. Darüberhinaus gibt es noch die Möglichkeit, die Direktive CLONE in der Form

```
C$DIR CLONE(ARG1 [,...]) bzw. /*$dir clone(ARG1 [,...])*/
```

zu verwenden, wobei die Parameter Argumente der Routine sind. Dies bewirkt, daß immer dann ein Klon angelegt wird, wenn für eines oder mehrere der angegebenen Argumente eine Konstante in die Funktion hineingezogen werden kann.
Außerdem gibt es noch die Möglichkeit, das Verhalten des Application-Compilers indirekt zu steuern, indem man ihn mit zusätzlichen Informationen versieht. Um beispielweise abzuschätzen, wie oft eine Routine aufgerufen wird, nimmt er bei einer IF-Anweisung an, daß beide Zweige gleich wahrscheinlich sind (falls die Bedingung nicht schon zur Compilezeit ausgewertet werden kann). Wenn man dagegen schon weiß, daß die Zweige sehr unterschiedlich oft ausgeführt werden, kann man dies durch eine Compiler-Direktive mitteilen:

```
condition_true(NN)    (Syntax fuer C bzw. Fortran analog zu oben)
```

wobei NN, eine ganze Zahl zwischen 1 und 99, die Wahrscheinlichkeit dafür angibt, daß die Bedingung wahr ist. Die Direktive muß in der Zeile direkt vor der IF-Anweisung stehen. Eine ähnliche Direktive ist

```
estimated_trips(NN)  ,
```

die vor einer Schleife stehen kann und angibt, wie oft die Schleife im Mittel ausgeführt wird. Bei beiden Direktiven ist schon eine grobe Schätzung hilfreich; nützlich sind auch die Werte, die der CXpa liefert.

Schließlich gibt es noch die Möglichkeit, dem Application-Compiler auf die Sprünge zu helfen, wenn er trotz seiner globalen Analysen fälschlicherweise annimmt, daß für einige Variable ein Alias vorliegt (d.h. eine Überlappung von Arraybereichen, s.o.). Dazu fügt man in das Buildfile Zeilen der Art

```
no_alias(VAR1, VAR2)
```

ein, wobei VAR1 und VAR2 die beiden Variablen mit dem Pseudo-Alias sind. Sind es Funktionsargumente, so werden sie angegeben als "funktionsname:variable" , wobei in FORTRAN alles groß zu schreiben ist. Eine Fortran-COMMON-Block-Variable bezeichnet man mit "/COMMONBLOCK/VARIAB bzw. "//VARIABLE" für den unbenannten COMMON-Block. Mit

```
no_alias(BLOCKNAME)
```

kann man pauschal angeben, daß kein Alias zwischen einer Variable aus dem Common-Block BLOCKNAME und irgendeiner anderen Variablen vorliegt.

4 Vektorisieren und Parallelisieren

In diesem Kapitel wollen wir uns genauer ansehen, was der Compiler beim Vektorisieren und Parallelisieren macht und wie man ihn dabei unterstützen kann. Dazu sind Kenntnisse des grundsätzlichen Aufbaus eines Vektorrechners nützlich, die im ersten Abschnitt vermittelt werden sollen.

4.1 Architektur der Convex C3800

Die C3800-Rechner der Firma Convex sind eine Familie von Multiprozessor-Rechnern mit ein bis acht Prozessoren; die C3840 der TUHH ist ein Vier-Prozessor-Modell. Jeder Prozessor ist mit Hardware zur Vektorverarbeitung ausgestattet, d.h. er hat spezielle Vektorregister und Vektorkommandos. Im einzelnen sind dies folgende Register:

- Vektorregister V0, V1, .. V7
  128 x 64 bit
  Ein Vektorregister enthält einen Satz von 128 64bit-Operanden. Es dient zum Zwischenspeichern und als Akkumulator von Ergebnissen von Vektor-Operationen.
- Vektor-Länge-Register VL
  7 bit
  VL enthält die Anzahl (0 .. 127) der Elemente eines Vektors.
- Vektor-"Stride"-Register VS
  32 bit
  VS gibt in Byte den Abstand im Speicher zwischen aufeinanderfolgenden Elementen eines Vektors an.
- Vektor-"Mask"-Register VM
  128 bit

- VM enthält für jedes Element eines Vektors ein Bit, das angibt, ob das Element an einer Vektoroperation teilnehmen soll.

In der Assembler-Sprache der C3800 gibt es u.a. folgende Arten von Vektor-Operationen:

- Lade-/Speicher-Operationen, arithmetische und logische Verknüpfungen ($=, -, *, /$, Quadratwurzel, UND-, ODER- und XOR-Verknüpfung),
- Reduktionen, d.h. Bildung eines Skalars aus einem Vektor, durch Addition, Multiplikation, Maximum, Minimum oder eine der drei logischen Operationen,
- Vergleichsoperationen $<, <=, =$ (das Ergebnis steht im VM-Register),
- Masken-Operationen (alle obigen, aber unter Verwendung des VM-Registers)
- Gather/Scatter-Operationen, d.h. Verteilen der Elemente eines Vektors über einen Indexvektor

Die ersten vier Gruppen gab es in ähnlicher Weise schon auf den ersten Vektorrechnern der Firma Cray, andere kamen hinzu, als deutlich wurde, daß sich damit große Klassen von Anwendungen

beschleunigen lassen würden. So erlauben es die Masken-Operationen, gewisse Schleifen auch dann zu vektorisieren, wenn sie IF-Bedingungen enthalten. Die Gather/Scatter-Operationen bieten eine schnelle Methode zur indirekten Adressierung ganzer Vektoren, wie sie vor allem bei Berechnungen mit dünn besetzten Matrizen auftreten.
Als Beispiel, wie diese Operationen benutzt werden, betrachten wir die Bildung des Skalarprodukts:
In Fortran:

```
INTEGER  A(100), B(100), SUMME, I

SUMME = 0
DO I=1, 100
   SUMME = SUMME + A(I)*B(I)
ENDDO
```

Im Assembler der C3800:

```
ld.w   #0, s0          ; lade s0 mit 0
ld.w   #100, VL        ; Vektorlaenge = 100
ld.w   #4, VS          ; Abstand der Elemente = 4 Byte
ld.w   A, v0           ; lade Vektor A nach v0
ld.w   B, v1           ;  und B nach v1
mul.w  v0, v1, v2      ; multipliziere, Ergebnis nach v2
sum.w  v2              ; addiere Elemente aus v2, Ergebnis ist in s2
st.w   s2, SUMME       ; speichere s2 nach SUMME
```

Auch wenn die Addition von bis zu 128 Elementen nur ein einziger Assemblerbefehl ist, werden doch nicht alle Elemente gleichzeitig addiert, sondern es wird sogenanntes "Pipelining" verwendet, d.h. es wird mit der Verarbeitung der nächsten Elemente schon begonnen, bevor das letzte fertig ist. Benötigt eine Addition z.B. 8 Prozessortakte (Zahl willkürlich), so kann man durch Pipelining erreichen, daß mit jedem Takt eine weitere Addition beginnt, und so schließlich pro Takt doch eine Addition durchgeführt wird, zuzüglich der "Startup"-Zeit von 7 Takten. Da außerdem noch Zeit für das Setzen von VL- und VS-Register gebraucht wird, rentiert sich das Verfahren erst ab einer gewissen Vektorlänge, d.h. Gesamtzahl von Additionen.
Eine weiteres Verfahren, um noch mehr Geschwindigkeit herauszuholen, ist das "Chaining", d.h. das direkte Weiterleiten von Ergebnissen einer Pipeline in die nächste, ohne daß auf die komplette Beendigung der ersten Operation gewartet werden müßte. Damit dies funktioniert, werden die arithmetischen Einheiten in Funktionsgruppen eingeteilt, zwischen denen ein solches Weiterleiten möglich ist. So ist es beispielsweise auf der C3800 möglich, eine Vektor-Multiplikation und eine Addition zu "chainen", nicht aber zwei Additionen.
Falls alle diese Möglichkeiten optimal ausgenutzt werden und auch der Speicherzugriff die Verarbeitung nicht hemmt (was ebenfalls einiger Voraussetzungen bedarf, s.u.), kann man mit einem C38-Prozessor eine maximale Verarbeitungsgeschwindigkeit von 120 MFLOPS (Millionen Fließpunkt-Operationen pro Sekunde) erzielen, bei der C3840 also bei optimaler Parallelisierung (s.u.) 480 MFLOPS. Zum Vergleich: die C240 schafft 200 MFLOPS, die C1 20 MFLOPS.[12]
Eine Besonderheit der C38 ist noch, daß bei Verwendung von einfacher Genauigkeit die Geschwindigkeit der Grundoperationen sich verdoppelt, so daß dann Spitzenwerte von 960 MFLOPS zu erzielen sind (jedenfalls theoretisch).

[12] Mit dieser Spitzen-Leistung ("Peak Performance") verhält es sich wie mit der Lichtgeschwindigkeit in der Physik: Normalerweise liegt man weit darunter, und je mehr man sich ihr nähern will, umso mehr Energie braucht man, ohne sie je erreichen zu können.

Neben der Ausnutzung der Prozessor-Eigenschaften ist auch die Organisation der Daten im Speicher wichtig, wenn man hohe Geschwindigkeiten erzielen will. Da das Übertragen von Daten aus dem Hauptspeicher in Register der CPU länger als einen Takt dauert, muß man wieder auf Pipelining zurückgreifen, wenn man Vektoren verarbeiten will. Dazu ist der Speicher in eine Anzahl von Bänken aufgeteilt, aus denen man gleichzeitig Daten laden kann, ohne daß sie sich gegenseitig behindern. Aufeinanderfolgende Speicherelemente werden dabei auf verschiedene Bänke verteilt. Dieses Verfahren heißt "Interleaving", die Anzahl der Bänke "Interleave-Faktor". Auf der C3840 der TUHH beträgt er 64. An einem Beispiel wollen wir uns das genauer ansehen:
Nehmen wir an, ein einzelner Speicherzugriff brauche 8 Takte und der Speicher sei in 8 Bänke eingeteilt. Wir wollen ein Array A von 16 Elementen aus dem Hauptspeicher in Register laden, das sich dann folgendermaßen auf die Bänke verteilt:

Bank	0	1	2	3	4	5	6	7
	A(0)	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)
	A(8)	A(9)	A(10)	A(11)	A(12)	A(13)	A(14)	A(15)

Mit jedem Takt kann mit dem Laden eines neuen Elementes begonnen werden, nach 7 Takten beginnt das Laden von A(7), nach 8 Takten ist der Ladevorgang von A(0) abgeschlossen und das Laden von A(8) aus der gleichen Bank kann sich nahtlos anschliessen.
Im Beispiel kann also tatsächlich mit jedem Takt ein Element geladen werden, so daß die Vektorverarbeitung an dieser Stelle nicht gebremst wird. Allerdings können verschiedene Faktoren dieses schöne Bild verändern:
Angenommen, wir wollten nur jedes zweite Element laden, dann würde nach vier Takten wieder auf Bank 0 zugegriffen. Diese ist aber noch 4 Takte lang mit A(0) beschäftigt, so daß der Ladevorgang so lange unterbrochen wird. Noch schlimmer kommt es, wenn wir etwa nur jedes 64. Element laden wollen, denn dann wird immer auf die gleiche Bank zugegriffen. Solche großen Abstände treten aber in der Praxis auf, etwa wenn man Spalten und Zeilen von Arrays braucht. Was dies für die Programmierung bedeutet, wollen wir uns in Abschnitt 4.3.4 ansehen.
Ein weiteres Problem ist der Zugriff auf kleine Daten-Einheiten: Die Größe eines Speicherelements beträgt 64 bit (8 Byte). Zwar kann man auf der C38 auch auf kleinere Einheiten bis herunter zum Byte direkt zugreifen, aber das Schreiben dauert für Bytes und Halbworte (2 Byte) dann 20 Takte statt 8. Allerdings geschieht der Zugriff auf Fließkomma-Zahlen einfacher Genauigkeit (4 Byte) mit maximaler Geschwindigkeit.
Um noch mehr Rechenleistung für ein Programm zu bekommen, ist es möglich, es auf mehreren CPUs parallel abarbeiten zu lassen. Wie man in Fortran oder C diese Möglichkeit nutzen kann, ist Gegenstand des Abschnitts 4.4. Hier wollen wir kurz die zugrundeliegenden Architektur-Merkmale vorstellen:
Die CPUs einer C3 greifen unabhängig voneinander auf den gesamten Speicher zu, so daß ein Austausch von Daten durch das Verschicken von Nachrichten, wie es z.B. auf einem Transputer-Cluster geschieht, auf dieser Ebene nicht nötig ist ("Shared Memory"-Maschine im Unterschied zur "Local Memory"-Architektur). Darüberhinaus gibt es zur Synchronisation und zum schnelleren Austausch von Daten 8 Sätze von je 128 Kommunikationsregistern zu 64 Bit und einem zusätzlichen "Lock"-Bit sowie zwei Index-Register, von denen eines, das "Communication Index Register" CIR, die Zuordnung einer CPU zu einem Registersatz steuert und das andere, "Thread Identifier" TID genannt, Speicherbereiche für einzelne CPUs aufteilt, die an einem Programm arbeiten.
Ein Ausführungsstrang eines Programms, der zu einer CPU gehört, wird "Thread" genannt. Es können maximal so viele Threads wie CPUs gemeinsam an einem Programm arbeiten. Sie gehören zu einem Prozeß und teilen sich dessen Resourcen, meistens auch den Speicher, können aber auch private Speicherbereiche benutzen. Ein Programm beginnt zunächst mit einem einzelnen Thread auf einer CPU und kann sich dann in mehrere parallele Threads aufteilen. Dazu gibt es zwei verschiedene, von der Architektur unterstützte Möglichkeiten:

1. Das Programm setzt die Anforderung ab, so viele Threads zu erzeugen wie Prozessoren zur Verfügung stehen. Diese arbeiten dann ihre Anteile am Problem ab und verabschieden sich mit einem "join". Dadurch wird die CPU wieder für andere Prozesse freigegeben, außer beim letzten Thread, der nach seinem "join" automatisch mit dem Programm weitermacht ("symmetrische Parallelverarbeitung").

2. Das Programm gibt an, wieviele Threads erzeugt werden sollen. Die weitere Aufsplittung oder das Zusammenkommen mehrerer Stränge kann und muß von den einzelnen Threads gesteuert werden. Dazu gibt es verschiedene Möglichkeiten der Kommunikation und Synchronisation zwischen Threads ("asymmetrische Parallelverarbeitung").

Bei beiden Verfahren kann nicht angegeben werden, welcher Thread auf welcher CPU abläuft, sondern wenn eine CPU Zeit hat, sucht sie sich den nächsten, bisher unbearbeiteten Thread. Dies kann auch dazu führen, daß etwa ein Programm von vier Threads nur auf drei Prozessoren läuft, von denen einer zwei nacheinander bearbeitet, weil der vierte Prozessor zu beschäftigt ist. Dieses Verfahren nennt Convex ASAP ("Automatic Self-Allocating Processors"), es hat den Vorteil, daß bei unterschiedlich langer Dauer einzelner Threads kein Prozessor auf die anderen warten muß, und erhöht bei genügender Last den Gesamt-Durchsatz der Maschine.
Allerdings ist dadurch das Laufzeit-Verhalten und der aktuelle Parallelisierungsgrad stark lastabhängig und damit unvorhersagbar. Daher gibt es für Prozesse die Möglichkeit des "fixed Scheduling": Wann immer ein solcher Prozeß läuft, werden alle CPUs für ihn freigehalten, so daß die mögliche Parallelität auch verwirklicht wird. Dies bedeutet auf der andern Seite natürlich, daß einige Prozessoren gelegentlich - in der Praxis eher: des öfteren - leer stehen. Daher wird man es selten, vor allem bei Messungen des Laufzeit-Verhaltens mit Profilern, einsetzen.

4.2 Automatische Vektorisierung

Da die hohe Rechenleistung der Convex nur bei ausgiebiger Verwendung der Vektor-Einheiten genutzt werden kann, ist die Vektorisierung, d.h. die Umwandlung des Codes in eine Form, in der Vektor-Befehle eingesetzt werden, der wichtigste Schritt bei der Optimierung. Zum Glück wird einem diese Aufgabe weitgehend von den Compilern abgenommen, die ab der Optimierungsstufe -O2 neben den skalaren Optimierungen der Stufen -O0 und -O1 umfangreiche Umformungen vornehmen, um möglichst viele Schleifen vektorisieren zu können.[13] In diesem Abschnitt wollen wir uns ansehen, welche Programm-Konstruktionen der Compiler automatisch in Vektorcode umwandeln kann bzw. welche die Vektorisierung verhindern.

4.2.1 Code-Umstellungen durch den Compiler

Die für die Vektorisierung grundlegende Fähigkeit des Compilers ist das Erkennen von Schleifen. Dabei sind nicht nur DO-Schleifen in Fortran oder for-Schleifen in C mögliche Kandidaten, sondern auch while- oder until-Schleifen, ja sogar mit GOTOs selbstprogrammierte Schleifen können vom Compiler erkannt werden. Wichtig ist dafür das Vorhandensein einer Schleifen-Variablen, die bei jeder Iteration um einen konstanten Betrag geändert wird. Manchmal gelingt aber sogar die Vektorisierung in komplizierteren Fällen, wie das Beispiel am Ende dieses Abschnitts eindrucksvoll beweist.

[13] Umgekehrt gilt aber auch: Programme, die nur mit -O1 oder niedriger optimiert wurden, enthalten keine Vektor-Kommandos und gehören daher nicht auf die C3840 (eine mögliche Ausnahme: skalare Programme, die den riesigen Speicher benötigen)!

Die einfachste Umformung ist das sogenannte "Strip Mining". Dabei wird eine Schleife, die u.U. mehr als 128 Iterationen lang ist, in eine innere Schleife mit 128 Iterationen (beim letzten Mal evtl. weniger) und eine äußere über die entsprechende Anzahl von 128er Blöcken aufgeteilt:[14]

vorher:

```
      DO I = 1, N
         BLA(I)
      ENDDO
```

nachher:

```
      N_BLOCK = N / 128
      N_REST = MOD(N, 128)
C
C 128er Bloecke
C
      DO BL = 0, N_BLOCK-1
        DO I = 1, 128
           BLA(128*BL + I)
        ENDDO
      ENDDO
C
C Rest
C
      DO I = 1, N_REST
         BLA(128*N_BLOCK + I)
      ENDDO
```

Eine Schleife, die neben normalen Operationen weitere Schleifen enthält, kann vektorisiert werden, indem sie in Teile gleicher Tiefe zerhackt wird ("loop distribution"), z.B.:

vorher:

```
      DO I = 1, N
         BLA1(I)
         DO J = 1, M
            BLA2(I, J)
         ENDDO
      ENDDO
```

[14]Die Programmfragmente sollen die Struktur der Umformung verdeutlichen; sie sind nicht unbedingt mit dem vom Compiler erzeugten Code identisch. BLA(I) etc. steht dabei nicht für einen Funktionsaufruf, sondern für eine Menge von Array-Operationen.

nachher:

```
DO I = 1, N
   BLA1(I)
ENDDO

DO I = 1, N
   DO J = 1, M
      BLA2(I, J)
   ENDDO
ENDDO
```

Eine Operation, die häufig vorkommt, ist das Vertauschen von innerer und äußerer Schleife, z.B. um den Speicherzugriff zu optimieren oder die Schleife mit der größeren Vektorlänge nach innen zu bringen. Z.B. würde bei

```
DO I = 1, N
  DO J = 1, M
     A(I,J) = 2* B(I,J)
  ENDDO
ENDDO
```

ohne weitere Kenntnis von N und M die I-Schleife nach innen getauscht, da Fortran-Arrays spaltenweise im Speicher liegen (d.h. es folgen im Speicher A(1,1), A(2,1), A(3,1),...) und die Vektorzugriffe dann auf direkt hintereinanderliegende Elemente erfolgen. Wäre dagegen N=10, M=1000 dem Compiler bekannt, würde er die Schleifen nicht tauschen und die längere Schleife vektorisieren (vgl. Abschnitt 3.3.1).

Schleifen, in denen IF-Abfragen vorkommen, sind schwierig zu vektorisieren. Da solche Schleifen aber oft vorkommen, gibt es spezielle Hardware, um damit fertig zu werden (Masken-Register, s. Abschnitt 4.1). Außerdem hat der Compiler verschiedene Strategien zur Verfügung, z.B. das Herausziehen aus der Schleife wie im folgenden Beispiel:

vorher:

```
DO I = 1, N
   IF (SEIN_ODER_NICHT_SEIN) THEN
      BLA1(I)
   ELSE
      BLA2(I)
   ENDIF
ENDDO
```

nachher:

```
IF (SEIN_ODER_NICHT_SEIN) THEN
   DO I = 1, N
      BLA1(I)
   ENDDO
ELSE
   DO I = 1, N
      BLA2(I)
   ENDDO
ENDIF
```

oder das Herauslösen von Anfangs- oder Endabfragen ("Loop Peeling"), etwa:

vorher:

```
DO I = 1, N
   IF (I .EQ. 1) THEN
      A(I) = B(N)
   ELSE
      A(I) = B(I-1)
   ENDIF
ENDDO
```

nachher:

```
A(1) = B(N)
DO I = 2, N
   A(I) = B(I-1)
ENDDO
```

Schließlich gibt es gewisse Konstruktionen, die immer wieder vorkommen, z.B:

```
MAX = A(1)
DO I = 2, N
   IF (A(I) .GT. MAX) THEN
      MAX = A(I)
   ENDIF
ENDDO
```

Solche Fälle erkennt der Compiler durch Vergleich mit einem Satz von Standardmustern.

Schließlich zeigt das folgende Beispiel, zu welch erstaunlichen Leistungen der Compiler fähig ist:

```
K = 0
DO I = 1, 100
   IF (TEST(I) .EQ. .TRUE.) THEN
      K = K + 1
      A(I) = B(K)
   ENDIF
ENDDO
```

Der erzeugte Maschinencode macht folgendes: Zunächst werden die Werte von I bestimmt, für die der TEST wahr ist und ihre Anzahl gezählt. Dann wird der entsprechende Teil von B geladen und nach A expandiert, wobei der Vektor der I-Werte als Index-Vektor wirkt (scatter-Operation, s. Abschnitt 4.1). Schließlich wird der Vektor A abgespeichert.

4.2.2 Hindernisse für die Vektorisierung

Natürlich gibt es Schleifen, die grundsätzlich nicht vektorisiert werden können, etwa wenn gar keine Array-Operationen vorkommen. Es gibt aber einige oft vorkommende Konstruktionen, die ein automatisches Vektorisieren auch in solchen Fällen verhindern, in denen es bei etwas anderer Programmierung möglich wäre. Beispiele für solche Vektorisierungs-Hemmnisse wollen wir uns nun ansehen.
Ein Routinen-Aufruf, eine Ein-/Ausgabe-Anweisung oder ein "switch"-Statement in einer Schleife verhindern die Vektorisierung sofort, ebenso eine Schleife mit einer Induktionsvariable vom Typ unsigned.
Wird in einer Schleife während einer Iteration auf ein Array-Element zugegriffen, das in einer anderen Iteration berechnet wurde, sprechen wir von "Rückgriff" ("Recurrence"). Ein solcher Rückgriff kann u.U. die Vektorisierung verhindern, wie etwa im einfachen Beispiel

```
DO I = 2, 100
   A(I) = A(I - 1) + B(I)
ENDDO
```

Da der Wert von A(2) vom Wert von A(1) abhängt, würde eine Vektorisierung, d.h. eine quasi gleichzeitige Berechnung von A(1) und A(2), zum falschen Ergenis führen. Dies wäre nicht der Fall, wenn A(I) von A(I+1) abhängen würde, da dann die notwendige Reihenfolge der Schritte auch bei der Vektorisierung erhalten bleibt. Es gibt allerdings Fälle, in denen der Compiler nicht entscheiden kann, ob ein Rückgriff vorliegt, etwa bei

```
DO I = 2, 99
   A(I) = A(I + K) + B(I)
ENDDO
```

Hier hängt es vom Vorzeichen von K ab, ob vektorisiert werden kann oder nicht. Kann der Compiler das nicht entscheiden (z.B. weil K als Parameter übergeben wird), wird er nicht vektorisieren. Man spricht dann von einem "anscheinenden Rückgriff" ("apparent recurrence").
Allerdings benutzt der Compiler einige Tricks, um aus einem Rückgriff das Beste zu machen. So kann er die Berechnung von Zwischenergebnisse in eigene Schleifen ziehen, die dann vektorisieren, oder er kann durch Skalare entstehende Rückgriffe manchmal dadurch auflösen, daß er den Skalar zu einem Vektor aufbläht. Außerdem gibt es eine große Klasse von Rückgriffen, die nichts anderes sind als Vektor-Reduktionen (Summe, Maximum etc. eines Vektors). Diese erkennt der Compiler in der Regel und vektorisiert sie entsprechend.
Ein großes Problem für die Vektorisierung ist auch das sogenannte "Aliasing", nämlich das Zugreifen auf einen Speicherbereich unter verschiedenen Namen. Dies tritt in der Regel bei der Verwendung von Zeigern auf, ist damit vor allem bei der Programmierung in C wichtig. [15] Wir sehen uns dazu folgendes Beispiel an:

[15] Für Aliasing in Fortran vgl. die Bemerkung in Abschnitt 3.3.1

```
void bla(float *a, float *b, int n)
{
   int i;

   for (i=0; i<n; i++)
   {
      a[i] = b[i];
   }
}
```

An sich sollte man meinen, daß es kaum schönere Vektor-Schleifen gibt. Was aber, wenn diese Routine folgendermaßen aufgerufen wird:

```
float x[1000];

/* versieh x mit Werten */
initialize_array(x, 1000);

bla(x+100, x+99, 100);
```

Durch die Überlappung der Arrays a und b ist ein Rückgriff entstanden, der eine Vektorisierung verhindert. Da der Compiler bei der Optimierung der Routine bla vom schlimmsten Fall ausgehen muß, darf er die Schleife nicht vektorisieren. Nun wird aber in C oft mit Pointern gearbeitet, so daß dies eine starke Einschränkung der Vektorisierung zur Folge hat.
Um mögliche Überlappungen verhindern zu können, geht der Compiler im ANSI-Mode folgendermaßen vor: Er sieht alle Pointer auf einen Typ (Pointer auf int, float etc.) als Zeiger in ein großes Array an, das alle globalen Variablen, alle lokalen statischen Variablen und alle lokalen Variablen, deren Adresse benutzt wird, enthält, die vom entsprechenden Typ sind. Zwei beliebige Zeiger in diesen Bereich werden als potentielle Aliasse angesehen und verhindern dadurch u.U. eine Vektorisierung. Folgendes Beispiel zeigt, wie weitreichende Folgen dieses Verfahren haben kann:

```
void bla(int *int_array)
{
   int i;
   int *int_ptr;

   int_ptr = &i;

   for (i=0; i<4711; i++)
   {
      int_array[i] = 42;
   }
}
```

Diese harmlos wirkende Schleife wird nicht vektorisiert: Da von der Variablen i der Adress-Operator genommen wird, gehört auch sie in den Bereich des imaginären Arrays aller int-Pointer, könnte also auch vom int_array erreicht werden. Die Wert-Zuweisung im int_array ist somit potentiell eine Änderung an i, das seinen Wert mithin innerhalb der Schleife ändern kann und somit keine echte Schleifenvariable darstellt. Noch schlimmer wird es, wenn man nicht mit ANSI-C arbeitet: Da die

Typen von Pointern im alten K&R-C nicht streng unterschieden werden, gibt es überhaupt nur ein einziges gedachtes Array für alle Pointer, nicht mehrere für die verschiedenen Typen. Die Anzahl der potentiellen Überlappungen ist daher noch größer, es werden noch weniger Schleifen vektorisiert. Problematisch wird es, wenn man mit ANSI-C arbeitet, aber die Pointer-Typisierung nicht strikt beachtet, wie etwa in

```
int   *int_ptr;
float *float_ptr;

float_ptr = int_ptr;
```

Der Compiler gibt zwar eine Warnung aus ("operands of = point to incompatible types."), akzeptiert es aber ansonsten. Auf diese Weise kann Aliasing auftreten, ohne daß der Compiler es merkt! Zum Glück wird das Aliasing-Problem durch den Application-Compiler deutlich entschärft. Man kann aber auch ohne diesen durch den Programmierstil eine bessere Vektorisierung seiner C-Programme erreichen (s. Abschnitt 4.3.1).

4.2.3 Der Optimierungs-Bericht

Über die Umformungen, die der Compiler zur Vektorisierung (und Parallelisierung bei -O3) vorgenommen hat, sowie über Optimierungs-Hindernisse gibt der Optimierungs-Bericht Auskunft. Er besteht aus zwei Teilen, dem Schleifen- und dem Array-Bericht. Letzterer ist in den wenigsten Fällen von Nutzen, defaultmäßig erscheint auch nur der Schleifen-Bericht. Für die Routine LUINV des Beispielprogramms LINALG wollen wir ihn uns genauer ansehen:

Optimization for Procedure LUINV

Line Num.		Iter. Var.	Reordering Transformation	Optimizing/Special Transformation	Exec. Mode
31		I	Dist		
31	-1	I	FULL VECTOR Inter		
31	-2	I	FULL VECTOR		
32	-1	J	Scalar		
41		I	Scalar		

Line Num.		Iter. Var.	Analysis
31	-1	I	Interchanged to innermost
41		I	Insufficient vector code

Der Schleifen-Bericht ist zweigeteilt: Zunächst kommt eine Liste aller entdeckten Schleifen, zusammen mit den an ihnen durchgeführten Transformationen, danach ggf. zusätzliche Informationen. Wird eine Schleife in mehrere Teile zerhackt ("distributed"), erscheint eine Zeile für jeden Teil; sie werden durch Nummern hinter der Zeilennummer voneinander unterschieden.
In unserem Beispiel entnehmen wir der ersten Zeile des Berichts, daß die I-Schleife in zwei Teile aufgeteilt wird: Im ersten wird die ganze Matrix auf 0 gesetzt, im zweiten die Diagonale auf 1. Die zweite Zeile mit ihrer unten stehenden Erläuterung gibt an, daß die I-Schleife nach innen getauscht wurde und dann vollständig vektorisiert. Die J-Schleife wird dann nicht vektorisiert (4. Zeile). Das

Setzen der Diagonale vektorisiert ebenfalls vollständig (3. Zeile). Daß die Schleife in Zeile 41 skalar ist, verwundert uns nicht: Sie enthält ja den Aufruf eines Unterprogramms.
Der Application-Compiler bringt ja einige neue Optimierungen, die auch entsprechend im Optimierungs-Bericht erwähnt werden, z.B. über das Klonen und Inlinen von Routinen. Der entsprechende Teil für die Routine LUINV sieht jetzt so aus:

Optimization for Procedure LUINV

Line Num.	Name	Reordering Transformation	Optimizing/Special Transformation	Exec. Mode
31	I	Dist		
31 -1	I	FULL VECTOR Inter No s		
31 -2	I	FULL VECTOR No strip		
32 -1	J	Scalar		
41	I	Scalar		
42	LUBKSB	No inline		

Line Num.	Name	Analysis
31 -1	I	Interchanged to innermost
41	I	Insufficient vector code
42	LUBKSB	Argument #1 requiers dynamic binding

Neu gegenüber dem normalen Compiler ist der Zusatz "No strip" bei den Schleifen 31-1 (leicht abgehackt) und 31-2. Da die Länge der Schleifen aus dem Hauptprogramm als N=100 bekannt ist, erübrigt sich ein Zerhacken der Schleife in 128-er Teile ("strip mining"). Außerdem wird explizit erwähnt, daß der Aufruf von LUBKSB in Zeile 42 nicht durch die Routine ersetzt wurde, weil "dynamisches Binden" erforderlich ist. Dynamisches Binden tritt auf, wenn ein Array im Aufrufer anders dimensioniert ist als in der aufgerufenen Routine, hier etwa das Array B.[16] Beim Inlinen entsteht durch die nötige Umstrukturierung ein zusätzlicher Overhead, daher wird es normalerweise in solchen Fällen nicht durchgeführt. Möchte man es dennoch, kann man es mit der build-Option "-permit dynamic_binding" erzwingen. Man sollte dann allerdings das sich ergebende Laufzeit-Verhalten untersuchen.

4.3 Manuelle Verbesserung der Vektorisierung

In diesem Abschnitt soll gezeigt werden, wie man durch geeigneten Programmierstil sowie durch Beeinflußung des Compilers über Optionen oder Direktiven den Vektorisierer unterstützen kann und besser vektorisierte Programme erhält. Allerdings bleibt im Einzelfall viel Raum für eigene Kreativität, wenn man wirklich das Letzte aus der Maschine herausholen will.

4.3.1 Vermeiden von Aliassen

Das größte Problem (jedenfalls in C) sind die (potentiellen) Aliasse, die der Compiler an vielen Stellen annehmen muß und daher nicht vektorisieren kann. Am einfachsten ist es natürlich, den

[16] Warum das Argument #1 sein soll, weiß ich auch nicht. Vielleicht zählt der Application Compiler rückwärts ?

Application Compiler zu benutzen, denn dieser verfolgt durch das ganze Programm hindurch, wohin welche Pointer zeigen, und löst somit viele scheinbare Aliasse auf. Gelingt dies in einigen Fällen nicht, so kann man durch das Einhalten einiger einfacher Regeln oft weiterkommen: Zunächst sollte man versuchen, das interne Alias-Array so klein wie möglich zu halten, indem man wenig mit Adress-Operatoren arbeitet und auf Werte von globalen Variablen in Schleifen zugreift, indem man sie vorher lokalen Variablen zuweist und diese benutzt. Folgendes Beispiel, das zunächst aus zwei Gründen nicht vektorisiert, wird dadurch problemlos optimiert:

vorher:

```
int offset = 2;

void bla(int a[], int *n)
{
   int i;

   for (i = 0; i < *n; i += offset)
   {
      a[i] = 3030;
   }
}
```

nachher:

```
int offset = 2;

void fein(int a[], int n)
{
   int i;
   int lok_off = offset;

   for (i = 0; i < n; i += lok_off)
   {
      a[i] = 3030;
   }
}
```

Für viele Fälle hilft noch folgende Möglichkeit über eine Convex-Erweiterung: In ANSI-C besteht kein großer Unterschied zwischen einer Deklaration der Form

```
void foo1(float *a, float *b);
```

und

```
void foo1(float a[], float b[]);
```

solange man die Pointer a und b nicht ändern will. Dem Convex-Compiler kann man nun aber durch die Option "-alias array_args" mitteilen, daß keine Überlappungen zwischen Array-Argumenten bestehen, die in der zweiten Form deklariert sind, sowie zwischen diesen und globalen Variablen.

Mit anderen Worten: Solche Arrays werden nicht als Pointer betrachtet, die irgendwohin zeigen können, sondern als ordentliche Arrays, die ihren eigenen Speicherbereich haben. Auf diese einfache Weise lassen sich ein Großteil hartnäckiger Alias-Probleme lösen. Allerdings ist man dann selbst dafür verantwortlich, daß auch wirklich keine Aliasse zwischen den Arrays auftreten. Solange man den entsprechenden Routinen auch nur "echte" Arrays übergibt und keine Pointer, sollte das aber keine Probleme machen.

4.3.2 Vereinfachung von Schleifen

Eine weitere Ursache für schlechte Vektorisierung sind zu große oder zu komplizierte Schleifen. Obwohl der Compiler auch Schleifen mit IFs in vielen Fällen vektorisieren kann, nimmt doch die Geschwindigkeit gegenüber Schleifen ohne IF ab. Man sollte daher immer versuchen, IFs aus Schleifen zu ziehen und sie, falls möglich, zu zerhacken. Die Auswirkungen dieser Strategie sehen wir an folgendem Beispiel:

```
PROGRAMM BEISPIEL

REAL*4    A(100,100)

CALL MAKE_ID1(A,100)
CALL MAKE_ID2(A,100)

END


SUBROUTINE  MAKE_ID1(A,N)

REAL*4  A(N,N)
INTEGER N
INTEGER I, J

DO I = 1, N
   DO J = 1, N
      IF (I .EQ. J) THEN
         A(I, J) = 1.0
      ELSE
         A(I, J) = 0.0
      ENDIF
   ENDDO
ENDDO

END

SUBROUTINE  MAKE_ID2(A,N)

REAL*4  A(N,N)
INTEGER N
INTEGER I, J
```

```
DO I = 1, N
   DO J = 1, N
      A(I, J) = 0.0
   ENDDO
   A(I, I) = 1.0
ENDDO

END
```

Obwohl in MAKE_ID2 die Diagonal-Elemente zweimal besetzt werden, zeigt eine Laufzeit-Analyse mit dem CXpa, daß diese Routine fast 6x schneller läuft! Zum Vergleich die Zeiten jeweils für -O1 und -O2:

Routine	-O1	-O2
MAKE_ID1	3.354m	0.571m
MAKE_ID2	1.737m	0.106m

Normalerweise würde man sich wohl mit dem Vektorisierungsgewinn von MAKE_ID1 zufrieden geben - und dabei eine ganze Menge verschenken!

4.3.3 Compiler-Direktiven

Ein weiteres Optimierungs-Problem sind scheinbare Rückgriffe, die vor allem dann auftreten, wenn Array-Offsets oder Schrittweiten in Schleifen als Funktions-Parameter übergeben werden. Da das Vorzeichen beim Compilieren nicht bekannt ist, können manchmal Rückgriffe nicht ausgeschloßen werden, obwohl sie im Programm nicht wirklich vorkommen. Ähnliche, wenn auch nicht so drastische Effekte gibt es, wenn die Länge einer Schleife ein Parameter ist: Mangels besseren Wissens wird vom Compiler immer vektorisiert und auch immer eine äußere Schleife für das Strip Mining angelegt. Wäre die Zahl der Iterationen bekannt, könnte man das Strip Mining u.U. unterlassen, andere Schleifen gar nicht vektorisieren, weil sie zu kurz sind und ggf. die Reihenfolge geschachtelter Schleifen ändern. Für alle diese Fälle gibt es die Möglichkeit, mit Hilfe von Compiler-Direktiven zusätzliche Informationen in den Code einzubauen. Da sich das Problem bei Verwendung des Application Compilers durch das Weiterreichen von Konstanten aber nicht mehr so sehr stellt, sollen hier nur kurz die wichtigsten Direktiven kurz erwähnt werden. Eine Direktive BLA_DIREKTIVE erscheint im Source-Code als

```
C$DIR BLA_DIREKTIVE            (in Fortran)
```

bzw.

```
#pragma _CNX bla_direktive     (in C).
```

```
FORCE_VEKTOR, PREFER_VECTOR, NO_VECTOR, SCALAR
```

vor einer Schleife gibt an, ob sie immer, nur wenn es sicher ist oder nie vektorisiert werden soll. Bei SCALAR wird sie auch nicht parallelisiert. Ähnliche Anweisungen gibt es für die Parallelisierung.

```
NO_RECURRENCE
```

Die folgende Schleife enthält keinen Rückgriff, auch wenn es so aussieht.

```
MAX_TRIPS(Anzahl)
```

In der folgenden Schleife wird höchstens "Anzahl" mal iteriert.

```
UNROLL
```

Die nächste Schleife ist sie kurz, daß sie in Einzelanweisungen aufgelöst ("unrolled") werden soll.

```
PEEL_ALL, PEEL, NO_PEEL
```

Aus der nächsten Schleife sollen immer bzw. nur, wenn es nicht zu aufwendig wird bzw. nie spezielle Anfangs- und Endwert-Behandlungen herausgezogen werden.

4.3.4 Optimierung von Speicherzugriffen

Neben der Vektorisierung ist die Optimierung von Speicherzugriffen eine Möglichkeit, sein Programm merklich zu beschleunigen. Insbesondere sollte man versuchen, Bankkonflikte zu vermeiden. Dazu reicht es oft schon, wenn man die Dimensionen seiner Arrays ungerade wählt, vor allem aber keine Zweierpotenzen benutzt. Folgendes Beispiel zeigt die Auswirkungen:

```
      PROGRAM BENCH

C        LOESUNG EINES LINEAREN GLEICHUNGSSYSTEMS: AX=B

      INTEGER   N
      PARAMETER (N=128)

      REAL * 4  A(N,N), B(N)
      INTEGER   IPVT(N)
      REAL * 4  CPUTIME, RAND
      REAL * 4  OP, T1, T2
      INTEGER   I, J, INFO


C     DEFINITION DER MATRIX A UND DER RECHTEN SEITE B

      DO  I = 1, N
         B(I) = 100.0 * RAND(0)
         DO J = 1, N
            A(I,J) = 10.0 * RAND(0)
         ENDDO
      ENDDO

C     LU-ZERLEGUNG DER MATRIX A OHNE KONDITIONSABSCHAETZUNG

      T1 = CPUTIME(0.0)
```

```
CALL SGEFA(A,N,N,IPVT,INFO)

IF(INFO .EQ. 0) THEN
   CALL SGESL(A,N,N,IPVT,B,0)
   T2 = CPUTIME(T1)
   OP = 0.67*N**3 + 2*N**2
   PRINT*, 'MFLOPS: ', OP/(T2*1.0E+06), ', DIM N: ', N
ELSE
   PRINT*, 'MATRIX NUMERISCH SINGULAER'
END IF

END
```

Die folgende Tabelle zeigt die MFLOPS-Ergebnisse für verschiedene Arraygrößen (alle Zahlen Mittelwerte über jeweils fünf Läufe):

N	P(N-1) [MFLOPS]	P(N) [MFLOPS]	P(N+1) [MFLOPS]	PM(N) [MFLOPS]	DP(N) [%]
64	69.2	31.5	65.2	67.2	53
128	123.8	61.4	130.3	127.1	52
256	139.5	83.2	146.3	142.9	42
512	154.7	112.4	159.2	157.0	28
1024	173.0	144.7	174.5	173.8	17

Dabei ist

$$\mathrm{PM}(N) = 0.5 * (\mathrm{P}(N-1) + \mathrm{P}(N+1))$$

und

$$\mathrm{DP}(N) = (\mathrm{PM}(N) - \mathrm{P}(N))/\mathrm{PM}(N) * 100\%.$$

Man sieht deutlich die Auswirkung der Bankkonflikte bei Zweierpotenzen als Array-Dimensionen. Ihr Einfluß nimmt mit steigendem N ab, wie DP(N) zeigt. Dies liegt zum einen daran, daß die Zahl der Speicherzugriffe relativ zur Zahl der arithmetischen Operationen abnimmt. Zum anderen hat man mit wachsender Problemgröße mehr Möglichkeiten, durch Umstellen der Reihenfolge von Operationen die Speicherwartezeit sinnvoll mit etwas anderem als Warten zu verbringen. Bei den hochoptimierten VECLIB-Routinen kann man sicher sein, daß diese Möglichkeiten sehr weitgehend ausgenutzt werden.
Im Abschnitt 4.1 wurde schon darauf hingewiesen, daß Rechnungen mit einfacher statt doppelter Genauigkeit doppelt so schnell sind. Ändert man im Programm BENCH die Arrays auf REAL8 und die VECLIB-Aufrufe in die entsprechenden für doppelte Genauigkeit (DGEFA und DGESL), erhält man z.B. bei N=129 einen Wert von 73.1 MFLOPS, also tatsächlich nur wenig mehr als die Hälfte.

4.3.5 Benutzung von Bibliotheken

Wie wir bereits gesehen haben, führt die Verwendung von herstellereigenen, an die Maschine angepaßten Routinen für numerische Grundoperationen in der Regel zu erheblichen Geschwindigkeitsgewinnen. Andererseits sind solche Programme natürlich nicht mehr zwischen Rechnern verschiedener Hersteller transportabel. Um zumindest zwischen Convex- und Cray-Rechnern kompatibel zu sein

– ein in der Praxis recht häufiger Fall –, gibt es auf der Convex die Bibliothek SCILIB, die die sogenannte "Scientific Library" von Cray in an die Convex angepaßter Weise implementiert. Routinen der Cray-Bibliothek, die in der VECLIB schon vorkommen, wurden nicht zusätzlich in die SCILIB integriert, so daß man meistens beide Bibliotheken mit "-lscilib -lveclib" hinzulinken muß. Die Veclib- und Scilib-Bibliotheken (und Lapack, s. u.) sind von Convex auch auf HP 900/700-Maschienen portiert. Sie wurden speziell angepaßt und sind sehr viel schneller als vergleichbare NAG-Routinen. Auf dem Convex/HP-Metacluster des RZ sind sie vorhanden.
Um das Kompatibilitätsproblem in allgemeiner Form zu lösen, wurde eine Bibliothek von grundlegenden Routinen definiert, die sogenannte BLAS ("Basic Linear Algebra Subprograms"), die von jedem Vektorrechner-Hersteller optimal implementiert werden sollte und die die Basis für eigene Numerik-Routinen bilden kann. Diese Idee wurde relativ rasch verwirklicht; bei Convex ist die BLAS ein Teil der VECLIB. Sie enthält elementare Operationen zwischen Vektoren ("BLAS Level 1"), zwischen Matrizen und Vektoren ("BLAS Level 2") und zwischen Matrizen ("BLAS Level 3"). Die Hersteller von numerischen Bibliotheken (NAG, IMSL) haben inzwischen ihre Routinen auf BLAS-Routinen als unterstem Level aufgebaut, so daß sie ebenfalls von der schnellen BLAS-Implementierung profitieren.[17]
Wie sehr sich die Verwendung sogar der einfachen BLAS-Routinen auszahlt, zeigt der Vergleich: Für die Multiplikation einer NxK-Matrix mit einer KxN-Matrix ergaben sich folgende Zeiten:

N	K	eigene Routine	BLAS (DGEMM)
1000	1000	21.401	19.452
1000	100	2.214	1.860
1000	10	0.303	0.204
1000	1	0.093	0.025

Die BLAS-Routine DGEMM ist für jede Kombination von Matrix-Dimensionen optimal, während die eigene Routine nur für große K gut abschneidet. Aber sogar dann ist die BLAS-Routine noch überlegen. Fazit: Selbst bei solch einfachen Problemen wie der Matrix-Multiplikation sollte man die VECLIB-Routinen eigenen Programmen vorziehen!
Aus einer anderen Richtung kommen die Bibliotheken LINPACK, EISPACK und LAPACK: Bei LINPACK und EISPACK war das Ziel, grundlegende Routinen aus der Linearen Algebra mit Sourcen zur Verfügung zu stellen. Allerdings sind diese Routinen noch nicht für Vektorrechner gedacht, dafür sind sie schon zu alt. Die Idee, diese Bibliotheken zu "modernisieren" und Routinen zu schreiben, die an Vektor-Architekturen angepaßt sind und auf die BLAS aufbauen, führte zur Entstehung der LAPACK-Bibliothek, deren Sourcen ebenfalls frei verfügbar sind. Eine Anpassung an Parallelrechner mit verteiltem Speicher ist in Vorbereitung. Die LINPACK- und EISPACK-Routinen sind Bestandteil der VECLIB, es wurden allerdings nicht alle Routinen an die Convex angepaßt. Die LAPACK wird inzwischen von Convex zusammen mit der VECLIB ausgeliefert.

[17] Dies ist auch der Grund, warum man zur NAG auch immer die VECLIB mit einbinden muß.

4.4 Parallelisierung

4.4.1 Automatische Parallelisierung

Bei der Compiler-Option -O3 wird zusätzlich eine "automatische Parallelisierung" durchgeführt, d.h. es wird versucht, das Programm in mehrere Threads zu zerlegen, die unabhängig voneinander von mehreren CPUs gleichzeitig bearbeitet werden können. Im Gegensatz zur Vektorisierung, die schon so weit geht, daß man oft auch ohne manuelle Nachbearbeitung zufriedenstellende Resultate erzielen kann, ist die Parallelisierung eher in den Anfängen.[18] Die einzige Konstruktion, die der Compiler selbsttätig parallelisiert, ist eine äußere Schleife (innere werden vektorisiert, s.o.). Welche Schleife die äußere ist, hängt dabei von vorhergehenden Optimierungen ab; beispielsweise kann sie durch Vertauschungen nach außen gekommen oder durch "Strip Mining", also Zerhacken von Schleifen in 128-er Blöcke, überhaupt erst entstanden sein. Es werden so viele Threads erzeugt wie Prozessoren vorhanden sind - auf der C3840 also vier -, die alle den gleichen Code ausführen (symmetrische Parallelisierung, s. Abschnitt 4.1). Auf wievielen Prozessoren die Threads wirklich abgearbeitet werden, ist aufgrund des ASAP-Verfahrens stark von der momentanen Systemlast abhängig (vgl. Abschnitt 4.1). Ein Thread bearbeitet immer die nächste freie Iteration der Schleife (einen "Chore"), bis die ganze Schleife abgearbeitet ist. Welches der nächste Chore ist, wird in einem Kommunikations-Register festgehalten, so daß die Synchronisation der Threads gewährleistet ist. Da die einzelnen Chores u.U. verschieden lange laufen und die CPUs unterschiedlich ausgelastet sein können, ist auch die Verteilung der Chores auf die Threads i.a. bei jedem Programmlauf anders.

Als Beispiel betrachten wir folgenden Ausschnitt aus der Routine LUINV des Programms linalg:

```
0028    C
0029    C SET UP IDENTITY MATRIX
0030    C
0031          DO 11 I = 1, N
0032             DO 12 J = 1, N
0033                B(I, J) = 0.0
0034     12      CONTINUE
0035             B(I, I) = 1.0
0036     11   CONTINUE
```

Der zugehörige Ausschnitt aus dem Optimierungs-Bericht ist:

Line Num.		Iter. Var.	Reordering Transformation	Optimizing/Special Transformation	Exec. Mode
31		I	Dist		
31	-1	I	FULL VECTOR Inter		
31	-2	I	PARA/VECTOR		
32	-1	J	PARALLEL		

Line Num.		Iter. Var.	Analysis
31	-1	I	Interchanged to innermost
31	-2	I	Parallel outer strip mine loop

[18] Der Application-Compiler soll hier weitergehen: Er ist geplant als "automatischer Parallelisierer und Vektorisierer" für ein zukünftiges Viel-Prozessor-System.

Folgende Umformungen wurden vom Compiler vorgenommen:

1. Die I-Schleife wurde in zwei Teile zerhackt ("distributed"):
   31-1 : Doppelschleife B(I,J) = 0.0
   31-2 : Einfachschleife B(I,I) = 1.0
2. In 31-1 wurde die I-Schleife nach innen getauscht und voll vektorisiert, die (jetzt äußere) J-Schleife wurde voll parallelisiert.
3. In 31-2 wurde die I-Schleife in 128-er Blöcke zerhackt ("strip mining"), der innere Teil wurde vektorisiert, der äußere Teil über die Blöcke parallelisisert (deshalb "PARA/VEKTOR").

Natürlich gibt es wieder einige Konstruktionen, die das Parallelisieren einer äußeren Schleife verhindern. I.w. handelt es sich um dieselben Faktoren wie beim Vektorisieren, u.a. Unterprogramm-Aufrufe oder I/O-Anweisungen in der Schleife oder Rückgriffe, d.h. Abhängigkeiten von Array-Elementen von in der Schleife vorher berechneten Elementen. Neu ist, daß auch nach vorne gehende Abhängigkeiten in einer Schleife das Parallelisieren behindern, wie etwa im folgenden Beispiel:

```
DO I=1, N-1
   A(I) = A(I+1) + 31.93
ENDDO
```

Diese Schleife kann zwar vektorisiert, aber nicht parallelisiert werden!

4.4.2 Debuggen parallelisierter Programme

Mit Hilfe des CXdb kann man auch parallelisierte Programme debuggen. Natürlich sind die Schwierigkeiten, die beim Debuggen vektorisierter Programme auftreten, hier genauso vorhanden. Zusätzlich kann man sich nun die Aufteilung des Programmablaufs auf einzelne Threads ansehen und sie einzeln oder zusammen steuern.
Um reproduzierbare Ergebnisse zu bekommen, sollte man als erstes das automatische Verteilen der Threads abschalten und vor dem Starten eines Prozesses im CXdb auf "fixed scheduling" gehen mit

```
set default fixed sched .
```

Dadurch stehen innerhalb einer zugeteilten Zeitscheibe immer alle CPUs zur Verfügung, so daß jedem Thread auch wirklich ein Prozessor entspricht.
Läßt man das Programm bis in einen parallelen Bereich hineinlaufen und dann anhalten (durch Breakpoints oder schrittweise), kann man Kommandos in einzelnen Threads ausführen, sie einzeln durchsteppen oder ihre eigenen Variablen ansehen, indem man vor die entsprechenden CXdb-Kommandos die Thread-Nummer (bei uns 0, 1, 2, 3) angibt in der Form

```
:tNR CXdb-Kommando,
```

z.B. ":t1 step", oder ":t1,3 step" für Threads 1 und 3 auf einmal. Im Source-Fenster wird durch Voranstellen der Threadnummer in der Form "1 >" vor einer Zeile angezeigt, wo die einzelnen Threads stehen. Befinden sich mehrere Threads an der gleichen Stelle, erscheint dort stattdessen nur die Zahl der Threads (als n@).
Weitere Informationen über die einzelnen Threads erhält man mit

```
info threads .
```

Schließlich gibt es die speziellen Eventpoints

```
event spawn  bzw.  event join ,
```

die die Programm-Ausführung anhalten sollen, wenn Threads erzeugt bzw. beendet werden.

4.4.3 Profilen parallelisierter Programme

Ein großes Problem beim Parallelisieren ist, daß der Erfolg nicht direkt sichtbar wird: Da die normalerweise ausgegebenen CPU-Zeiten immer die Summe über alle CPUs sind, steigt die Zeit bei Verwendung von "-O3" durch den Parallelisierungs-Overhead immer an! Um zu sehen, wie gut ein Programm parallelisiert wird, muß man es mit dem CXpa untersuchen. Dazu übersetzt und linkt man es wieder mit der Option "-pa" und startet den CXpa dann mit

```
cpa -f linalg  .
```

Die Option "-f" bewirkt, daß der CXpa mit "fixed scheduling" arbeitet, also - wie schon beim CXdb möglich - innerhalb einer zugeteilten Zeitscheibe alle CPUs zur Verfügung stehen.

Im CXpa können jetzt alle parallelen Schleifen mit

```
monitor pregion all
```

ausgewählt werden. Statt "all" ist mit "in ROUTINE" oder "at LINE" wieder eine Auswahl der zu überwachenden parallelen Bereiche möglich. Nach

```
run
analyze pregion [ROUTINE ...]
```

erhält man die Meß-Ergebnisse in Form von zwei Tabellen.

Als Beispiel betrachten wir folgenden Teil der Routine TESTMT aus dem Programm linalg:

```
0024           MAXVAL = 0.0
0025    C
0026    C  BERECHNE REST = A*AINV - ID
0027    C
0028           DO 11 I = 1, N
0029              DO 12 J = 1, N
0030                 REST = 0.0
0031                 DO 13 K = 1, N
0032                    REST = REST + A(I, K)*AINV(K, J)
0033     13          CONTINUE
0034                 IF (I .EQ. J) THEN
0035                    REST = DABS(REST - 1.0)
```

```
0036                ELSE
0037                      REST = DABS(REST)
0038                END IF
0039                MAXVAL = MAX(MAXVAL, REST)
0040      12     CONTINUE
0041      11   CONTINUE
```

Beim Übersetzen mit "-O3" bekommen wir folgenden Optimierungs-Bericht:

Optimization for Procedure TESTMT

Line Num.	Iter. Var.	Reordering Transformation	Optimizing/Special Transformation	Exec. Mode
28	I	PARALLEL		
29	J	Scalar		
31	K	FULL VECTOR	Reduction	

Line Num.	Iter. Var.	Analysis
28	I	Unable to distribute
29	J	Unable to distribute

Die äußere I-Schleife wird also parallelisiert, während die J-Schleife skalar bleibt. Nur die innere K-Schleife wird vektorisiert. Nach entsprechender Instrumentierung erhalten wir vom CXpa folgendes Ergebnis:

Parallel Region Performance Analysis
For testmt.f:testmt

Line Num	Num Exec	Cpu Time	Wall Clock Time	Process Virtual Time	CPU/PVT
28	2	0.153986	0.395657	0.039482	3.90016

Line Num	Cpu Time	Wall Clock Time	Chore Count
28	0.039302	0.395461	51
	0.039135	0.395470	51
	0.038609	0.394882	50
	0.036940	0.393540	48

Die erste Tabelle zeigt, daß die I-Schleife (Zeile 28) zweimal durchlaufen wurde (TESTMT wird zweimal aufgerufen). Die CPU-Zeit für die Schleife ist wieder die Summe über alle CPUs, also näherungsweise gleich der CPU-Zeit der Schleife ohne Parallelisierung. Die "Wall Clock Time" ist die "real"-Zeit, also stark von der momentanen Auslastung abhängig. Die zur Bestimmung der Parallelität wichtige Zeit ist die "Process Virtual Time" (PVT). Sie wird durch einen Timer aufgenommen, der die "real"-Zeit "innerhalb eines Prozesses" mißt, d.h. er läuft nur weiter, wenn der Prozeß RUNNING ist, wobei die Zahl der CPUs, die gerade benutzt werden, keine Rolle spielt. Allerdings berücksichtigt die PVT nur User-, keine System-Anteile (insbesondere also keine I/O-Zeiten) und enthält den Instrumentierungs-Overhead vom CXpa, der nur insgesamt, aber nicht für jede CPU einzeln abgezogen werden kann.

Bis auf diese Ungenauigkeiten, die insbesondere bei CPU-intensiven Schleifen gering sind, gibt die letzte Spalte "CPU/PVT" die Geschwindigkeitserhöhung durch die Parallelisierung an. Ein Wert von 3.9 bei vier Prozessoren ist natürlich nahezu ideal; er entspricht einer Parallelisierungs-Effektivität von

$$(\mathrm{CPU/PVT})/\mathrm{NCPU} * 100\% = 3.9/4 * 100\% = 97.5\%.$$

Eine Möglichkeit, die Genauigkeit etwas zu erhöhen, besteht darin, die CPU-Zeit für die nicht-parallelisierte Schleife explizit zu messen (also: -O2, CXpa für die zu untersuchende Schleife). Wer auf ganz genaue Ergebnisse angewiesen ist, etwa für offizielle Benchmarks, dem bleibt nichts anderes übrig, als die Programmlaufzeiten im Single-User-Betrieb zu messen. Für die meisten Belange sollte aber der einfache CPU/PVT-Wert des CXpa ausreichen.
Die zweite Tabelle gibt an, wie stark sich die Arbeit auf die einzelnen CPUs verteilt hat. Die Beispiel-Schleife wurde zweimal mit jeweils 100 Iterationen durchlaufen, d.h. die Gesamtzahl der Chores beträgt 200. Wie man sieht, haben diese sich sehr gut auf die CPUs verteilt, sowohl nach der Zahl der Chores als auch nach der CPU-Zeit. Eine ungleiche Verteilung der Arbeit ist oft die Ursache für einen schlechten Parallelisierungsgrad. Dieses Problem tritt vor allem auf, wenn die Zahl der Chores klein ist (etwa 5 Chores auf 4 Prozessoren) oder wenn eine äußere "Strip-Mining"-Schleife mit zu kleiner Vektorlänge aufgeteilt wird (z.B. 400 Vektor-Elemente bei 4 Prozessoren: drei bekommen jeweils 128 Elemente, einer die restlichen 16). In solchen Fällen ist u.U. über Compiler-Direktiven eine bessere Aufteilung zu erreichen (s. Abschnitt 4.4.4).
Leider haben meine Messungen ergeben, daß die Concurrency-Werte des CXpa in manchen Fällen völlig falsch sind (zu großer Overhead?). Man braucht also gelegentlich eine Methode, um unabhängig den Parallelisierungsgrad zu messen. Dazu muß man die System-Routine cvxprusage benutzen, am besten, indem man sich eine kleine Hilfsroutine schreibt, wie etwa die folgende:

```
/*
 *  Routine zur Messungen von parallelen Zeiten
 */

#include <sys/time.h>
#include <sys/resource.h>


void paruse(long long *nsamples, long long *nthreads)
{

   /*
    * gibt die Gesamtzahl der Samples und der Threads bis zum Aufruf-Zeitpunkt
    * zurueck
    */

   struct cvxprusage   usage;

   cvxprusage(&usage);

   *nsamples = usage.pru_usamples;
   *nthreads = usage.pru_utotal;

}
```

Damit bestimmt man den Parallelisierungsgrad auf folgende Weise:

```
long long nsamp1, nsamp2, nthr1, nthr2;

paruse(&nsamp1, &nthr1);
compute();
paruse(&nsamp2, &nthr2);
concurrency = (nthr2 - nthr1)/(double)(nsamp2 - nsamp1);
```

Möchte man paruse in einem FORTRAN-Programm benutzen, muß man es in paruse_ umnennen und aufrufen mit

```
INTEGER*8 NSAMP, NTHR
CALL PARUSE(NSAMP, NTHR)    .
```

Um das Programm - ebenso wie bei CXpa-Messungen - unter "fixed scheduling" ablaufen zu lassen, startet man es auf folgende Weise:

```
mpa -f PROGRAMMNAME [ARGUMENTE]    .
```

4.4.4 Compiler-Direktiven zur Parallelisierung

Das Beispielprogramm linalg zeigt, daß ein globales Übersetzen aller Routinen mit der Option "-O3" nicht sinnvoll ist: Der Parallelisierungsgewinn liegt für viele Schleifen unter 1, d.h. sie arbeiten mit vier CPUs langsamer als mit einer! Man sollte die Option "-O3" also zunächst nur bei den Routinen einsetzen, die dadurch insgesamt schneller werden, und alle anderen nur vektorisieren. Weitergehende Schritte erfordern die Verwendung von Compiler-Direktiven, die mit "C$DIR DIREKTIVE" in Fortran- bzw. mit "#pragma_CNX direktive" in C-Programme eingebaut werden.
Möchte man auch solche Routinen parallelisieren, bei denen nur einige Schleifen davon profitieren, kann man den Compiler durch die Direktive

```
NO_PARALLEL
```

vor einer schlecht parallelisierenden Schleife anweisen, diese nicht zu parallelisieren.
Es gibt eine ganze Reihe von Direktiven, mit denen man den Compiler dazu bringen kann, eine Schleife zu parallelisieren:

- `FORCE_PARALLEL`
  Die nächste Schleife wird parallelisiert, auch wenn Abhängigkeiten zwischen den Iterationen bestehen!
- `FORCE_PARALLEL_EXT`
  Wie FORCE_PARALLEL, aber das Vertauschen der nächsten Schleifen bleibt erlaubt, so daß die tatsächlich parallelisierte Schleife auch die innere sein kann.
- `PREFER_PARALLEL, PREFER_PARALLEL_EXT`
  analog zu oben, aber wenn der Compiler Abhängigkeiten entdeckt, wird nicht parallelisiert.
- `SYNCH_PARALLEL`
  Die nächste Schleife wird parallelisiert. Bestehende Abhängigkeiten werden durch den Einbau geeigneter Synchronisationsanweisungen berücksichtigt.

Natürlich sind diese Anweisungen - vor allem FORCE_PARALLEL – mit Vorsicht zu benutzen; man sollte sich hinterher immer davon überzeugen, daß das Programm für verschiedene Eingabewerte noch richtige Ergebnisse produziert. Da aber das genaue Verhalten eines parallelisierten Programms lastabhängig ist, kann ein Fehler u.U. lange verborgen bleiben.
Ein typischer Anwendungsfall für diese Direktiven sind Schleifen, in denen Routinen aufgerufen werden. Solche Schleifen werden vom Compiler grundsätzlich nicht parallelisiert, da Seiteneffekte der Funktion, z.B. über globale Variable, möglicherweise eine Synchronisation der Iterationen erfordern, ohne daß der Compiler dies feststellen kann. Wenn man weiß, daß die Routine in der Schleife keine Seiteneffekte hat, kann man die Parallelisierung der Schleife durch eine der obigen Direktiven erzwingen. Außerdem muß man die Funktion mit der Compiler-Option "-re" ("reentrant") übersetzen, damit bei gleichzeitiger mehrfacher Ausführung der Funktion jedes Exemplar seinen eigenen Datenbereich bekommt. Ohne diese Option würde der Compiler nur einen Text- und Stackbereich für die Funktion anlegen, so daß alle parallelen Versionen auf dieselben lokalen Variablen zugreifen, was natürlich zu einem vollständigen Durcheinander führen könnte.
Man kann auch über die Parallelisierung von Schleifen hinausgehen, indem man eigenständige "Tasks" definiert, also Programmteile, die bei der Ausführung verschiedenen Threads zugeordnet werden können. Dazu dienen die Anweisungen

```
BEGIN_TASKS, NEXT_TASK, END_TASKS .
```

Schließlich gibt es Direktiven, mit denen man die Synchronisation von Threads explizit steuern kann, indem man kritische Bereiche mit Hilfe von Sperr-Variablen ("Locks") sichert. Näheres dazu kann man dem "FORTRAN/C Optimization Guide" von Convex entnehmen.

5 Fallstudien

In diesem Abschnitt werden die vorgestellten Tools und Methoden an zwei Beispielprogrammen ausprobiert. Zwar sind beide Programme kürzer als die meisten Anwendungen, lösen aber doch jeweils ein bestimmtes Problem, incl. vollständiger Ein- und Ausgabe. Sie sind insofern typisch, als der größte Teil des Codes für die Optimierung völlig irrelevant ist. Bei beiden Programmen wurde von Anfang an auf die Einhaltung von Standards (FORTRAN-77 bzw. ANSI-C), auf konzeptionell saubere Programmierung und gründliche Tests Wert gelegt. Inwieweit dies realisiert werden konnte, bleibt dem Leser zur Beurteilung und zum Test überlassen. Die Sourcen aller Versionen sowie benötigte Make- und Datenfiles befinden sich unter `/tuhh/info/tu_beispiele/convex_kurs` in den Verzeichnissen poisson bzw. nkoerper.

Die Beschreibung der einzelnen Optimierungs-Schritte verläuft in folgender Weise: Für jede Programm-Version wird zunächst der Unterschied zur vorherigen beschrieben. Danach folgen Ergebnisse der Laufzeitmessung mit dem time-Kommando (Summe aus User- und System-Anteil), meistens für verschiedene Optimierungsstufen, sowie ggf. weitere wichtige Daten, etwa Profiler-Ergebnisse oder relevante Teile aus dem Optimierungsbericht. Schließlich werden diese Daten analysiert und Bemerkungen zum weiteren Vorgehen gemacht.

Die Darstellung orientiert sich am tatsächlichen Vorgehen bei der Optimierung, wobei auch einige der Fehlversuche eingeschlossen wurden. Natürlich sind viele andere Reihenfolgen oder auch Vorgehensweisen denkbar; auch wird kein Anspruch darauf erhoben, alle Möglichkeiten ausgeschöpft zu haben. Vielmehr soll ein Einblick in die Vorgehensweise bei der Programm-Optimierung vermittelt werden.

5.1 POISSON - zweidimensionale Potentialprobleme in FORTRAN

Das Programm POISSON dient zur Bestimmung des elektrostatischen Potentials bei vorgegebener Ladungsverteilung in zwei Dimensionen. Es basiert auf dem SOR-Verfahren ("Simultaneous Over-Relaxation"), das z.B. in den "Numerical Recipes in C" von Press, Flannery, Teukolsky und Vetterling beschrieben ist. Die dort beschriebene SOR-Routine wird – leicht abgewandelt – in POISSON benutzt.

Das Hauptprogramm steht in main.f, es arbeitet in folgender Weise:

Zunächst wird mit GETSRC die Größe der Ladungs-Matrix und die Ladungsverteilung selbst bestimmt. Zur Kontrolle und zur Weiterverwendung bei der späteren bildlichen Darstellung ("Visualisierung") wird sie mit WRITEQ in ein File ausgegeben. Da die verwendete SOR-Routine wesentlich allgemeinere Gleichungen (nämlich beliebige elliptische partielle Differentialgleichungen) lösen kann, werden zunächst in der Routine POISSN die allgemeinen Koeffizienten speziell für die Poisson-Gleichung gesetzt. Mit UINIT wird eine willkührliche Anfangslösung (hier: Potential U == 0) als Startwert für die Iterationen vorgegeben. Außerdem benötigt die SOR-Routine noch den Spektralradius RJAC, einen Parameter, der von der speziellen Form der Gleichung abhängt und der für die Poisson-Gleichung mit der Funktion SPECR bestimmt wird. Schließlich wird das SOR-Verfahren durchgeführt und die Lösung mit WRITEU ausgegeben.

Version 1.1

Änderungen:

- keine, Ausgangsversion

Meßergebnisse:

time(-no): 469.6s
time(-O0): 339.8s
time(-O1): 242.0s
time(-O2): 214.9s

CXpa:

	CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
-O1:	233.170	84.9%	233.170	84.9%	1	sor
	0.967	0.4%	23.761	8.6%	1	writeu
	0.798	0.3%	17.731	6.5%	1	writeq
	.	.	.	.		
-O2:	198.702	82.7%	198.702	82.7%	1	sor
	0.954	0.4%	23.338	9.7%	1	writeu
	0.790	0.3%	18.091	7.5%	1	writeq

Optimierungsbericht für SOR:

Line Num.	Iter. Var.	Reordering Transformation	Optimizing/Special Transformation	Exec. Mode
34	J	FULL VECTOR Inter	Reduction	
35	L	Scalar		
48	J	Scalar		
49	L	72% VECTOR	Reduction	
43	IT	Scalar	Peel	
48	J	Scalar		
49	L	72% VECTOR	Reduction	

Line Num.	Iter. Var.	Analysis
34	J	Interchanged to innermost
49	L	Assign to U recurrence on 55
43	IT	Loop or inner loop has exit
49	L	Assign to U recurrence on 55

Line Num.	Col. Num.	Test Transformation	Analysis
62	17	TEST REMOVED	Peeled first iteration of loop on 43

Analyse:

- skalare Optimierung bis zur Stufe -O1 bringt eine Menge
- schlechter Vektorisierungsgrad
- SOR-Routine Hauptverbraucher, muß besser vektorisiert werden

- entscheidene Schleife in Z.48/49 (Durchgang durch das Gitter) vektorisiert schlecht wegen komplizierter IF-Bedingung in Z.50

Bemerkungen:

- Die IF-Bedingung in Z.50 bewirkt ein Durchgehen durch das Gitter im Schachbrett-Muster: Für gerades bzw. ungerades IT werden nur jeweils die "weißen" bzw. "schwarzen" Felder verändert. Der neue Wert eines "weißen" Feldes hängt nur von Werten auf "schwarzen" Feldern ab.
  Im Bild:

J				
3	X		X	
2		X		X
1	X	O	X	
0		X		X
	0	1	2	3 L

(X: J+L ungerade)
(O: zu veränderndes Feld)

  Trotz der Meldung im Optimierungsbericht liegt also keine echte "Recurrence" in dieser Schleife vor.

Version 1.2

Änderungen:

- Compiler-Direktive "NO_RECURRENCE" vor die Schleife in Z.49

Meßergebnisse:

time(-O2): 86.4s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
73.161	63.9%	73.161	63.9%	1	sor
1.026	0.9%	23.356	20.4%	1	writeu
0.790	0.7%	17.877	15.6%	1	writeq

Analyse:

- Vektorisierungsgrad deutlich verbessert
- SOR bleibt Hauptverbraucher, aber auch relativ großer CPU-Anteil der Ausgabe-Routinen

Bemerkungen:

- Wie wir in Abschnitt 4.3.2 gesehen haben, führt eine IF-Bedingung in einer Schleife auch bei vollständiger Vektorisierung zu Geschwindigkeitsverlusten. Die komplizierte IF-Bedingung der "Schachbrett"-Schleifen muß also vereinfacht werden.
- Die Routine WRITEQ war nur zur Kontrolle eingebaut worden, sie kann jetzt auch entfallen.

Version 1.3

Änderungen:

- IF-Bedingung in Z.51 durch Aufteilen der Schleife in mehrere Teile weggefallen
- Routine WRITEQ gestrichen

Meßergebnisse:

time(-O2): 54.7s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
47.953	66.9%	47.953	66.9%	1	sor
1.132	1.6%	23.741	33.1%	1	writeu

Analyse:

- SOR deutlich schneller
- relativer Anteil von WRITEU sehr gestiegen

Bemerkungen:

- Durch die geänderte Reihenfolge der Berechnungen hat sich in der Ergebnisdatei ein Wert minimal geändert.

Version 1.4

Änderungen:

- In WRITEU implizite DO-Liste statt expliziter Schleife

Meßergebnisse:

time(-O2): 51.3s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
48.284	81.3%	48.284	81.3%	1	sor
0.460	0.8%	11.059	18.6%	1	writeu

Analyse:

- Array-Ausgabe durch implizite DO-Liste schneller

Bemerkungen:

- Die Schachbrett-Struktur in SOR legt zur Erzeugung längerer Vektoren ein Durchlaufen des Arrays längs der Diagonal-Richtungen nahe. Dazu verhilft folgender Trick: Die Differentialgleichung wird nicht direkt, sondern nach einer Drehung der x-y-Ebene um 45° diskretisiert. Dies führt (neben einem Faktor $\sqrt{2}$) dazu, daß beim Updaten statt der direkten jetzt die diagonalen Nachbarn benötigt werden. Vorteil: Eine ganze Spalte kann upgedatet werden, ohne daß innerhalb der Spalte Rückgriffe auftreten. Die Numerik wird dabei eher besser. Der so veränderte Algorithmus heißt "XSOR".

Version 1.5

Änderungen:

- Algorithmus von SOR in XSOR geändert

Meßergebnisse:

time(-O2): 38.4s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
28.210	70.8%	28.210	70.8%	1	sor
0.377	0.9%	11.612	29.1%	1	writeu

Analyse:

- XSOR ist deutlich schneller als SOR

Bemerkungen:

- Wir erhalten im Rahmen unserer Genauigkeit etwas abweichende Ergebnisse; man müßte also beide bei höherer Genauigkeit vergleichen. In unserem Fall überzeugen wir uns nur qualitativ (durch Visualisieren mit Unigraph) davon, daß die Ergebnisse im wesentlichen übereinstimmen.
- Um die Speicherzugriffe zu beschleunigen, betrachten wir verschiedene Werte für die physikalische Arraygröße NMAX:

NMAX	Zeit	Bemerkung
300	38.4	Ausgangswert
301	37.4	ungerade Dimension ist meist besser
320	214.5	$320 = 5 * 64$ (= Interleave): max. Bankkonflikte
321	28.0	1 größer als $n *$ Interleave: Optimaler Wert!

Version 1.6

Änderungen:

- phys. Arraygröße in NMAX=321 geändert

Meßergebnisse:

time(-O2): 28.0s

Analyse:

- Verringerung der Bankkonflikte bringt deutliche Beschleunigung

Bemerkungen:

- Die allgemeine SOR-Routine kann durch direkte Anpassung an die Poisson-Gleichung noch stark vereinfacht werden.

Version 1.7

Änderungen:

- Koeffizienten-Arrays A - E in SOR durch Konstanten ersetzt
- POISSN gestrichen

Meßergebnisse:

time(-O2): 16.7s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
13.833	55.4%	13.833	55.4%	1	sor
0.379	1.5%	11.132	44.6%	1	writeu

Analyse:

- Zeit für SOR mehr als halbiert
- Ausgabe mit WRITEU hat nun großen relativen Anteil

Bemerkungen:

- Um den I/O-Anteil zu drücken, soll die Array-Ausgabe unformatiert erfolgen. Da die Daten hinterher mit Unigraph ausgewertet werden sollen, das nur REAL-Daten binär einlesen kann, wird mit der REAL-Funktion explizit nach REAL gewandelt.

Version 1.8

Änderungen:

- Ausgabe in WRITEU unformatiert

Meßergebnisse:

time(-O2): 13.8s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
13.057	92.4%	13.057	92.4%	1	sor
0.331	2.3%	1.065	7.5%	1	writeu

Analyse:

- Anteil von WRITEU hat drastisch abgenommen

Bemerkungen:

- Bei genauerer Betrachtung der Zeiten wird deutlich, daß etwas nicht stimmen kann: Der Anteil von writeu in Version 1.7 betrug fast die Hälfte der CPU-Zeit von 16.7s, nämlich etwa 7.4s. In Version 1.8 soll der Anteil von WRITEU auf nahezu vernachlässigbare 7.5% gesunken sein. Das bedeutet aber, daß die Gesamtzeit auch um etwa 7s zurückgehen müßte. Tatsächlich ist sie aber nur um 2.9s gesunken! Wie sich nach längerem Testen, insbesondere mit expliziten Aufrufen von CPUTIME in main.f, herausstellte, liegt hier ein Bug des CXpa (Version 1.3) vor: Die Zeiten für FORTRAN-Writes sind immer wesentlich zu hoch!

Version 1.9

Änderungen:

- CPUTIME-Aufrufe eingefügt, um die Zeiten für WRITEU und SOR explizit zu messen

Meßergebnisse:

Zeit	:	13.8s
SOR	:	13.4s
WRITEU:		0.3s

zum Vergleich analoge Messung mit Version 1.7:

Zeit	:	16.7s
SOR	:	13.6s
WRITEU:		3.1s

Analyse:

- Wie man sieht, hat der Anteil von writeu tatsächlich drastisch abgenommen, wenn er auch nie so groß war, wie uns die CXpa-Ergebnisse glauben ließen.

Bemerkungen:

- Neben der höheren Ausgabegeschwindigkeit besteht ein weiterer Vorteil des unformatierten Schreibens darin, daß auch das Einlesen der Daten in Unigraph jetzt wesentlich schneller geht. Übrigens hat der vermeintliche Nachteil (Wandlung nach $REAL * 4$ für Unigraph) sich zeitlich nicht ausgewirkt: Ein direktes Herausschreiben der $REAL * 8$-Daten ist im Gegenteil sogar langsamer – wegen der doppelt so großen Datenmenge!
- Beim Arbeiten mit unformatierten Daten ist noch darauf zu achten, daß die Formate i.a. maschinenspezifisch sind, so daß eine Auswertung ohne Wandlung nur auf Maschienen möglich ist, die das gleiche Datenformat verwenden. Allerdings benutzen fast alle gängigen Workstations und die Convex für Fließkommazahlen das von der IEEE genormte Format, so daß normalerweise keine Probleme auftreten sollten.

Parallelisierung

Änderungen:

- Version 1.8, sor.f mit -O3 übersetzt

Meßergebnisse:

time(-O3): 8.9s real 19.8s user 0.2s system

CXpa:

Parallel Region Performance Analysis
For sor.f:sor

Line Num	Num Exec	Cpu Time	Wall Clock Time	Process Virtual Time	CPU/PVT
34	1	0.002797	0.001849	0.001756	1.59282
48	296	0.023810	0.039057	0.021411	1.11205
48	260184	20.956636	84.758220	19.041363	1.10058

Analyse:

- Parallelisierungsgewinn nach time-Ergebnis: > 2.2
  gemessen bei einem weiteren Job im System
- Parallelisierungsgewinn nach CXpa-Messung: ~ 1.1

Bemerkungen:

- Schleife in Z.48 taucht zweimal auf, weil wegen des Tests in Z.59 der Wert IT=1 der Hauptschleife (Z.42) herausgezogen wurde
- Die Meßwerte des CXpa sind nicht konsistent mit der einfachen time-Messung!
- Eigene Messungen mit der Routine PARUSE (s. 4.4.3) ergeben einen Wert von 2.9 für die Parallelität (concurrency") der SOR-Routine, in Übereinstimmung mit dem time-Wert.

Zusammenfassung der Optimierungen:

Version	time(-O2)/s	Änderungen
1.1	214.9	Ausgangsversion
1.2	86.4	Compiler-Direktive in sor.f eingefügt
1.3	54.7	Schleifen in sor vereinfacht, writeq gestrichen
1.4	51.3	Ausgabe durch implizite DO-Liste in writeu.f
1.5	38.4	sor in xsor geändert
1.6	28.0	Arraygröße von 300 auf 321 gesetzt
1.7	16.7	Koeffizientenmatrizen in sor durch Konstanten ersetzt
1.8	13.8	Ausgabe in writeu.f binär

Zum Vergleich: Zeiten der verschiedenen Versionen auf einer HP9000/720:

Version	Optionen		Zeit
1.1	-A		1399.3s
	-A	+O3	666.6s
1.2			
1.3	-A	+O3	650.4s
1.4			
1.5	-A	+O3	469.6s
1.6	-A	+O3	464.0s
1.7	-A	+O3	167.5s
1.8	-A	+O3	165.5s

5.2 NKOERPER – Bahnkurven von Massepunkten unter Gravitationskraft in C

Das Programm NKOERPER löst die Bewegungsgleichungen von Teilchen, die sich unter dem Einfluß der wechselseitigen Gravitationskräfte bewegen. Es benutzt dazu ein Runge-Kutta-Verfahren mit Adaption der Schrittweite. Die entsprechenden Routinen runge_kutta und integriere sind Adaptionen von Programmen aus dem Buch "Theoretische Physik mit dem Personal Computer" von Schmid, Spitz und Lösch.

Das Hauptprogramm main ist folgendermaßen aufgebaut:

Mit io_init werden alle Eingabe-Parameter wie Zahl der Teilchen, ihre Massen, Anfangsorte und -geschwindigkeiten eingelesen und über den globalen IO-Block zurückgegeben.[19] Die Routine dvector besorgt den Speicherplatz für Orts- und Geschwindigkeits-Vektoren, die dann auf die Anfangsbedingungen gesetzt werden. Nach diesen Vorbereitungen startet die Rechnung: Mit integriere werden Ort und Geschwindigkeit der Teilchen für den gegenwärtigen Zeitpunkt bestimmt und mit write_vector ausgegeben. set_vector setzt die Vektoren für die nächste Iteration und die Zeit wird um einen Schritt erhöht. Nach der Rechnung werden noch mit io_close abschließende Arbeiten erledigt (Files geschloßen).

"integriere" implementiert die Schrittweiten-Anpassung und ruft für die eigentliche Integration die Routine runge_kutta auf, die allgemeine Systeme von Differentialgleichungen erster Ordnung lösen kann. Das hier benötigte spezielle System

```
[xdot]a  =  [v]a                                   (a: Teilchenindex)
[vdot]a  =  sum (b!=a) [m]b /([r]ab ** 3)  *  ([x]b - [x]a)
```

wird durch "kepler_dgl" definiert.

Version 2.1

Änderungen:

- keine, Ausgangsversion

Meßergebnisse:

time(-no): 438.2s
time(-O0): 364.9s
time(-O1): 314.7s
time(-O2): 331.7s

CXpa(-O1):

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
306.987	56.0%	546.519	99.7%	3924	kepler_dgl
239.526	43.7%	539.526	43.7%	39632727	mthd_pown
1.307	0.2%	547.845	99.9%	981	runge_kutta

[19] Diese Konstruktion ist analog zu COMMON-Blöcken in FORTRAN, unterliegt aber einer strengen Typprüfung.

Analyse:

- NKOERPER vektorisiert nicht
- Zeit geht in kepler_dgl und sein "Kind" mthd_pown

Bemerkungen:

- mthd_pown ist die Potenz-Funktion aus der Standard-Bibliothek. Da nur die Potenz 3/2 auftritt, kann sie durch die Wurzel-Funktion ersetzt werden, die auf der C3 in Hardware realisiert ist.

Version 2.2

Änderungen:

- pow-Funktion in kepler_dgl durch sqrt ersetzt

Meßergebnisse:

time(-O1): 168.2s
time(-O2): 179.3s

CXpa(-O1):

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
166.538	99.0%	166.546	99.0%	3924	kepler_dgl

Analyse:

- großer Zeitanteil der Potenz-Routine völlig weggefallen
- kepler_dgl vektorisiert nicht

Bemerkungen:

- Die Schleifen in kepler_dgl sind zu groß und zu kompliziert, sie werden in Teile aufgespalten.
- Man kann die Symmetrie der Kräfte (F_ab = F_ba, actio = reactio) mit folgendem Trick ausnutzen: Statt des Vektors rinv3[b] in Z.92, der nur für einen Wert von a (ein Teilchen) gilt, wird ein Array rinv3[a][b] berechnet, wobei dann die Symmetrie eingehen kann.
- Zur einfachen dynamischen Erzeugung eines Arrays wird eine Funktion darray analog zu dvector benötigt. Damit auf das Array mit der üblichen Index-Notation a[i][j] zugegriffen werden kann, muß man einen Vektor a von Zeigern auf die Zeilen des Arrays erzeugen und entsprechend initialisieren.

Version 2.3

Änderungen:

- Hilfsarray rinv3[a][b] eingeführt
- Symmetrie ausgenutzt durch Umstrukturierung der Schleifen
- neue Routine darray in vektor_utl.c

Meßergebnisse:

time(-O1): 128.3s
time(-O2): 142.2s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
126.037	98.7%	126.082	98.7%	3924	kepler_dgl (-O1)
140.455	98.9%	140.463	98.9%	3924	kepler_dgl (-O2)

Optimierungsbericht für kepler_dgl:

Line Num.	Iter. Var.	Reordering Transformation	Optimizing / Special Transformation	Exec. Mode
7	a	Scalar		
72	b	Scalar		
77	i	FULL VECTOR	Reduction	
87	a	Scalar		
89	b	Scalar		
96	a	Scalar		
98	i	Scalar		
103	b	Scalar		

Line Num.	Iter. Var.	Analysis
70	a	Inner loop has varying trip count
72	b	Unable to distribute
87	a	Inner loop has varying trip count
89	b	Insufficient vector code
96	a	Unable to distribute
98	i	Unable to distribute
103	b	Insufficient vector code

Analyse:

- Skalarzeit wesentlich verbessert
- einzige vektorisierte Schleife: Z.77

Bemerkungen:

- Schleife in Z.77 hat nur Vektorlänge d=2, sollte also besser nicht vektorisieren. Da d aber erst zur Laufzeit bestimmt wird, kann dies kein Compiler erkennen.
- Application-Compiler bringt gar nichts.
- kepler_dgl enthält zwei Teile: “x’ = v“und “v’ = F“. Der erste Teil kann in einer eigenen Schleife über alle Teilchen bestimmt werden.
- Um die Extrabehandlung der Diagonalen (a==b) in Z.105 - Z.108 mit dem continue loszuwerden, setzen wir bei der Berechnung von rinv3 rinv3[a][a] = 0.

Version 2.4

Änderungen:

- Schleifenstruktur in kepler_dgl vereinfacht

Meßergebnisse:

time(-O1): 122.2s
time(-O2): 120.9s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
120.066	98.7%	120.074	98.7%	3924	kepler_dgl (-O1)
119.051	98.7%	119.059	98.7%	3924	kepler_dgl (-O2)

Analyse:

- keine Vektorisierung auch einfacher Schleifen in kepler_dgl, da das dynamische Array rinv3 mit Zeilenpointern aufgebaut ist.

Bemerkungen:

- Der Zugriff auf Arrays über Zeilenzeiger verhindert, daß Spalten als Vektoren erkannt werden können. Um Vektorisierung überhaupt möglich zu machen, müssen die dynamischen Arrays also anders erzeugt werden. Dazu wird ein Array als eine Datenstruktur mit einem langen Vektor und Dimensionen definiert und über ein spezielles INDEX-Makro darauf zugegriffen. Die Elemente werden dann statt mit der Notation a[i][j] durch INDEX(a,i,j) angesprochen.

Version 2.5

Änderungen:

- Typ dmatrix und Makro INDEX in nkoerper.h definiert
- darray in vektor_utl.c angepaßt
- kepler_dgl angepaßt

Meßergebnisse:

time(-O1): 101.6s
time(-O2): 107.7s

Optimierungsbericht für kepler_dgl:

Line Num.	Iter. Var.	Reordering Transformation	Optimizing / Special Transformation	Exec. Mode
71	a	Scalar		
73	b	Scalar		
77	i	FULL VECTOR	Reduction	
87	a	Scalar		
89	b	Scalar		
96	a	FULL VECTOR		
103	i	Scalar		
109	a	Scalar		
111	i	Scalar		
114	b	Scalar		

Line Num.	Iter. Var.	Analysis
71	a	Inner loop has varying trip count
73	b	Unable to distribute
87	a	Inner loop has varying trip count
89	b	Insufficient vector code
103	i	Insufficient vector code
109	a	Unable to distribute
111	i	Unable to distribute
114	b	Insufficient vector code

Loop Performance Analysis
For kepler_dgl.c:kepler_dgl

Line Number	Times Exec.	Iteration Min	Max	Avg	Count Total CPU Time
71	3924	101	101	101	109.110868
73	392400	1	100	50	108.079641
77	19816200	2	2	2	44.060601
87	3924	101	101	101	10.615843
89	392400	1	100	50	9.681291
96	3924	101	101	101	0.019290
103	3924	202	202	202	0.223310
109	3924	101	101	101	61.225854
111	396324	2	2	2	60.099856
114	792648	101	101	101	57.622604

Analyse:

- Schleife Z.96 (rinv3[a][a] = 0) vektorisiert, d.h. grundsätzlich ist Vektorisierung mit der neuen Datenstruktur möglich
- übrige Schleifen enthalten alias-Probleme
- Hauptschleifen: Z.71-84 und Z.114

Bemerkungen:

- Die Schleifen Z.71-84 können durch Einführung eines Hilfsarrays r[a][b], das die Abstände der Teilchen enthält, vereinfacht werden.
- Die Schleifen über Teilchenindizes müssen explizit nach innen gebracht werden, die über Raumindizes nach außen, weil der Compiler nicht wissen kann, daß die Teilchenzahl wesentlich größer ist als die Dimension des Raums.

Version 2.6

Änderungen:

- Vereinfachung und Aufteilung der Schleifenstruktur in Z.71-84

Meßergebnisse:

time(-O1): 100.5s
time(-O2): 88.4s

Optimierungsbericht für kepler_dgl:

Line Num.	Iter. Var.	Reordering Transformation	Optimizing / Special Transformation	Exec. Mode
74	a	Scalar		
76	b	FULL VECTOR		
82	a	Scalar		
84	i	Scalar		
86	b	Scalar		
93	a	Scalar		
95	b	Scalar		
103	a	Scalar		
105	b	Scalar		
112	a	FULL VECTOR		
120	i	Scalar		
126	a	Scalar		
128	i	Scalar		
131	b	Scalar		

Analyse:

- kaum Vektorisierung wegen Alias-Problem (z.B. zwischen r und x)

Bemerkungen:

- Auch der Application-Compiler schafft es nicht, die Aliasse aufzulösen. Anscheinend versagt die Pointer-Analyse bei dynamisch erzeugten Arrays.

Version 2.7

Änderungen:

- no_recurrence-Direktive eingefügt in Z.86, 96, 107, 123

Meßergebnisse:

time(-O2): 65.4s

Optimierungsbericht für kepler_dgl:

Line Num.	Iter. Var.	Reordering Transformation	Optimizing / Special Transformation	Exec. Mode
74	a	Scalar		
76	b	FULL VECTOR		
82	a	Scalar		
84	i	Scalar		
87	b	FULL VECTOR		
94	a	Scalar		
97	b	FULL VECTOR		
105	a	Scalar		
108	b	FULL VECTOR		
115	a	FULL VECTOR		
124	i	FULL VECTOR		
130	a	Scalar		
132	i	Scalar		
135	b	Scalar		

Analyse:

- Schleifen in Z.74-124 voll vektorisiert

Bemerkungen:

- Nun müssen noch die Schleifen Z.130-135 zerlegt werden.

Version 2.8

Änderungen:

- extra Schleife für fv=0 eingeführt
- Schleife über Teilchenindex a nach innen getauscht
- no_recurrence-Direktive vor Schleife über a

Meßergebnisse:

time(-O1): 93.8s
time(-O2): 22.3s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
20.708	93.0%	20.723	93.0%	3924	kepler_dgl

Analyse:

- volle Vektorisierung aller möglichen Schleifen in kepler_dgl
- Vektorisierungs-Beschleunigung 4.2

Bemerkungen:

- Da kepler_dgl auch bei guter Vektorisierung immer noch den größten Teil der Zeit verbraucht, bleibt zur weiteren Beschleunigung nur die Möglichkeit, den Algorithmus zum Lösen der DGL mit variabler Schrittweite so zu verbessern, daß bei gleicher Genauigkeit weniger oft kepler_dgl aufgerufen wird. Dazu wollen wir uns die entsprechenden NAG-Routinen vornehmen.

Version 3.1

Änderungen:

- integriere nur noch Interface zur NAG-Routine D02BAF
- in kepler_dgl die Variablen x und v zu einem Vektor (x,v) zusammengefaßt

Meßergebnisse:

time(-O2): 2.7s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
2.435	92.7%	2.437	92.8%	495	kepler_dgl

Analyse:

- kepler_dgl wird statt 3924 mal nur 495 mal aufgerufen

Bemerkungen:

- Um das Ergebnis beurteilen zu können, müssen auch die Genauigkeiten beider Verfahren verglichen werden. Die Konstante eps hat laut NAG-Manual eine ähnliche Bedeutung wie in Version 2.1 - 2.8, allerdings ist nicht klar, was dies für die Genauigkeit des Ergebnisses bedeutet. Im Rahmen der Ausgabe-Genauigkeit (4 Dezimalen) sind beide Verfahren identisch, so daß sich die Frage stellt, welche Genauigkeit überhaupt nötig ist und ob man nicht auch dadurch enorm CPU-Zeit sparen kann, daß man die Genauigkeit heruntersetzt. Dies erfordert aber weitergehende Untersuchungen, die hier nicht angestellt werden.
- Dem NAG-Manual kann man entnehmen, daß die Routine D02BAF im Normalfall, insbesondere bei nicht zu zeitaufwendiger rechter Seite anzuwenden ist. Ansonsten wird D02CAF ("Adams-Verfahren") empfohlen. Als Änderung ist nur ein Austausch des Routinennamnes und der Größe des Workspace nötig.

Version 3.2

Änderungen:

- NAG-Routine D02BAF durch D02CAF ersetzt

Meßergebnisse:

time(-O2): 0.9s

CXpa:

CPU Time (less children)		CPU Time (plus children)		Times Called	Routine Name
0.787	79.8%	0.788	79.9%	142	kepler_dgl

Analyse:

- Zeitgewinn durch drastisch reduzierte Zahl der Aufrufe von kepler_dgl

Bemerkungen:

- Beschleunigung gegenüber Version 2.1: ≈ 350 !

Zusammenfaßung der Optimierungen:

Version	time(-O2)/s	Änderungen
2.1	331.7	Ausgangsversion
2.2	179.3	pow-Funktion in kepler_dgl durch sqrt ersetzt
2.3	142.2	Symmetrie, Vereinfachung der Schleifen in kepler_dgl
2.4	120.9	Schleifenstruktur in kepler_dgl weiter vereinfacht
2.5	107.7	Typ dmatrix und Makro INDEX eingeführt
2.6	88.4	Vereinfachung der Schleifenstruktur in Z.71-84
2.7	65.4	no_recurrence-Direktiven
2.8	22.3	Vereinfachung der Schleifen in Z.130-135, Direktive
3.1	2.7	integriere durch NAG-Routine D02BAF ersetzt
3.2	0.9	NAG-Routine D02CAF statt D02BAF