**FORSCHUNGSZENTRUM JÜLICH GmbH**
**Zentralinstitut für Angewandte Mathematik**
**D-52425 Jülich, Tel. (02461) 61-6402**

Interner Bericht

# Datenlokalität in Prozessoren mit hierarchischem Speichersystem am Beispiel des DEC Alpha-Chip

*Andreas Krumme*

KFA-ZAM-IB-9503

Februar 1995
(Stand 13.02.95)

Erschienen in: Proceedings des PARS-Workshop, 19./20. September 1994, Dresden,
PARS-Mitteilungen Nr. 13, November 1994

# Datenlokalität in Prozessoren mit hierarchischem Speichersystem am Beispiel des DEC Alpha-Chip

Andreas Krumme

Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich (KFA)
D-52425 Jülich
e-mail: a.krumme@kfa-juelich.de

**Zusammenfassung.** Für die Lösung von komplexen, sehr rechenintensiven Aufgabenstellungen aus dem technisch-naturwissenschaftlichen Bereich werden immer häufiger hochparallele Mehrprozessorsysteme eingesetzt. Diese Rechner bestehen aus RISC-Prozessoren, die nur dann ihre volle Leistung nutzen können, wenn die Daten im Cache-Speicher des Prozessors gehalten werden und nicht bei jedem Zugriff aus dem Speicher geladen werden müssen. Für die effektive Nutzung der hohen theoretischen Peak-Performance dieser modernen superskalaren und superpipelined RISC-Prozessoren spielt die gemeinsame Betrachtung von Prozessor, Speicher und Compiler eine Schlüsselrolle. Zu dem gleichen Ergebnis kommt auch der vorliegende Artikel, in dem stellvertretend für einige neuartige RISC-Prozessor-Architekturen der DEC Alpha-Chip AA-21064 als Repräsentant untersucht wird. Auf diesem Prozessortyp basiert auch der massiv-parallele Rechner T3D der Firma CRAY Research Inc.

## 1 Einleitung

Der DEC Alpha-Chip ist der erste 64-Bit RISC-Prozessor, den der Hersteller Digital Equipment in eigener Produktion herstellt. Seiner Entwicklung ging die Überlegung voraus, daß in nächster Zeit die bisherigen 32-Bit-Architekturen nicht mehr in der Lage sein werden, die immer größer werdenden Speicher zu adressieren. Neben der realisierten 64-Bit-Architektur ist insbesondere die hohe Taktrate von bis zu 150 MHz ein charakteristisches Merkmal. Aufgrund der parallelen Anordnung der internen Funktionseinheiten kann theoretisch pro Takt eine Fließpunktoperation durchgeführt werden, was eine Peak-Performance von maximal 150 MFLOPS entspricht. Da jedoch die Zugriffszeit der verwendeten Speicherbausteine keine entsprechende Verbesserung aufweist, muß untersucht werden, welche Rechenleistung tatsächlich erreichbar ist.

## 2 Aufbau des DEC Alpha-Chip AA-1064

Die Alpha-AXP-Architektur [2] ist eine 64-Bit RISC Load/Store-Architektur: Alle Datenbewegungen zwischen dem Speichersystem und den Registern werden ausschließlich durch Lade- und Speicherinstruktionen ausgeführt, alle Operationen werden auf den Registerinhalten durchgeführt. Der DEC Alpha-Chip AA-21064 [6] ist die erste Implementation der Alpha-AXP-Architektur. Er besteht, wie in Abbildung 1 dargestellt, aus folgenden Komponenten:

- **Das Leitwerk des Prozessors (IBOX)** enthält eine doppelt ausgeführte statische Leitwerk-Pipeline, die das Laden, Dekodieren und die Weitergabe von Instruktionen an die entsprechenden Funktionseinheiten (EBOX, FBOX und ABOX) durchführt. Die Ressourcen-Konfliktlogik ermittelt den aktuellen Belegungszustand der jeweils anzusprechenden dynamischen Pipeline der Funktionseinheiten. Die Pipeline-Kontrolleinheit dient zur Steuerung der Leitwerk-Pipeline und verzögert im Konfliktfall die Instruktionsausgabe. Zur schnelleren Zuordnung von virtuellen und physikalischen Instruktionsadressen ist ein Puffer vorgesehen. Der Programmzähler beinhaltet die Adresse der aktuell bearbeiteten Instruktion. Die Prefetching-Einheit steuert das Laden von Instruktionen in die erste Stufe der Leitwerk-Pipeline.

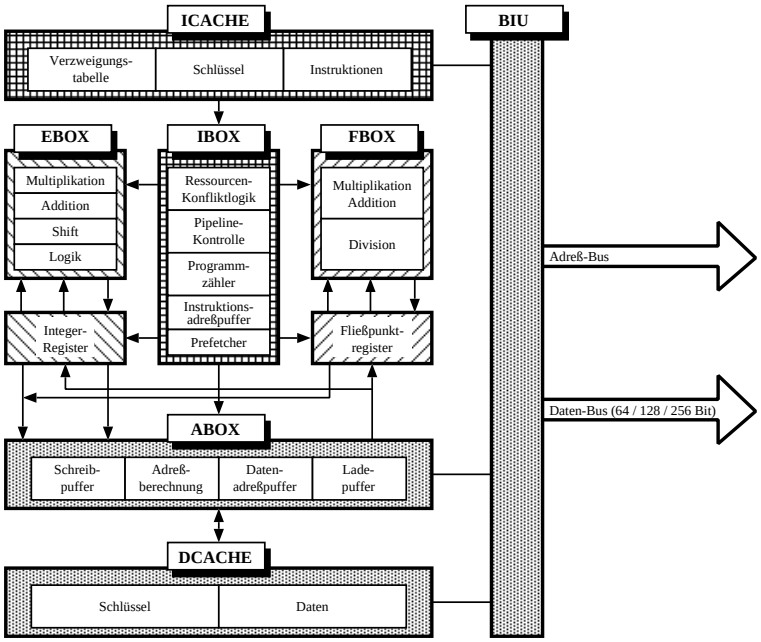



**Abb. 1.** Komponenten des DEC Alpha-Chip AA-21064

- **Der interne Instruktions-Cache-Speicher (ICACHE)** ist ein 8 KByte großer, einfach-assoziativer, physikalisch adressierter Cache-Speicher für die Zwischenspeicherung der zuletzt durchgeführten Instruktionen.
- **Die Integer-Funktionseinheit (EBOX)** führt arithmetische und logische Integer-Operationen aus. Alle Instruktionen, außer der Division und Multiplikation, werden in einer dreistufigen dynamischen Pipeline ausgeführt und verarbeiten ausschließlich 64-Bit-Daten aus den 32 Integer-Registern. Die Multiplikation ist als Assembler-Befehl vorgesehen, wird aber nicht in der Pipeline abgearbeitet. Die Integer-Division wird durch ein Unterprogramm realisiert, welches vom Compiler zur Übersetzungszeit eingebunden wird.
- **Die Fließpunkt-Funktionseinheit (FBOX)** enthält eine sechsstufige dynamische Pipeline zur Durchführung von Fließpunktaddition und -multiplikation. Divisionsinstruktionen werden nicht in der Pipeline abgearbeitet und beanspruchen einen hohen Zeitaufwand zur Durchführung. Alle Fließpunktoperationen verarbeiten ausschließlich 64-Bit-Daten aus den 32 Fließpunktregistern.
- **Der Fließpunkt-Registersatz** verfügt über zwei Schreib- und drei Lesepfade, zwei davon zur EBOX. Es sind keine vier Lesepfade wie bei den Integer-Registern notwendig, da kein Instruktionspaar existiert, welches auf vier Fließpunktregister lesend zugreift.
- **Der Integer-Registersatz** verfügt über zwei Schreibpfade und vier Lesepfade zur ABOX und EBOX.
- **Der interne Daten-Cache-Speicher (DCACHE)** ist ein 8 KByte großer, einfach-assoziativer Cache-Speicher für die Zwischenspeicherung der zuletzt referenzierten Daten. Er ist physikalisch adressiert und nach der *write-through* Strategie ausgelegt.
- **Die Adreß-Funktionseinheit (ABOX)** führt in einer dreistufigen Pipeline die Adreßberechnung und den Datenaustausch zwischen den Registern und dem übrigen Speichersystem aus. Die in Abbildung 1 angeführten Komponenten Lade- und Schreibpuffer dienen als Zwischenspeicher zur Erweiterung der Datentransfer-Bandbreite. Im Datenadreßpuffer sind die zuletzt genutzten virtuellen und zugehörige physikalischen Speicheradressen abgelegt. Die *External Bus Interface Unit* (BIU) bildet die Schnittstelle zum externen Speichersystem.

Die Performance-Analysen wurden auf den Rechnern

- DEC 3000, Model 400S AXP (Server) und
- DEC 3000, Model 300 AXP
- CRAY T3D, 1 Prozessor

durchgeführt, deren Daten in Tabelle 1 aufgelistet sind. Die Unterschiede zwischen den Systemen bestehen im wesentlichen in der unterschiedlichen Taktfrequenz, der Speichergröße und der Breite der Datenwege.

| **Modellbezeichnung** | **DEC 3000 Model 400S AXP** | **DEC 3000 Model 300 AXP** | **CRAY T3D** |
|---|---|---|---|
| **CPU** | DECchip 21064 RISC Microprocessor | | |
| **Taktfrequenz** | 133 MHz (7.5 ns) | 150 MHz (6.6 ns) | 150 MHz (6.6 ns) |
| **CPU Datenbusbreite** | 128 Bit | 64 Bit | 64 Bit |
| **Instruktionsinitiierung** | Maximal 2 Instruktionen pro Takt | | |
| **Maschinenwortlänge** | 64 Bit | | |
| **Hauptspeicher** | 128 MByte | 32 MByte | 64 MByte |
| **ICache-Speicher** | 8 KByte, einfach-assoziativ | | |
| **DCache-Speicher** | 8 KByte, einfach-assoziativ, write-through | | |
| **Externer Cache-Speicher** | 512 KByte einfach-assoziativ write-back | 256 KByte einfach-assoziativ write-back | nicht vorhanden |
| **Speicherbusbreite** | 256 Bit | 64 Bit | 128 Bit |
| **Speicherlatenzzeit** | | | |
| int. DCache-Speicher | 3 Takte | 3 Takte | 3 Takte |
| ext. Cache-Speicher | 8 Takte | 8 Takte | |
| Hauptspeicher | 24-30 Takte | 30-67 Takte | CRAY Prop. |

**Tabelle 1:** Hardware-Daten der untersuchten Prozessor-Realisierungen

## 3 Optimierung

Um die hohe theoretische Leistungsfähigkeit von superskalaren und superpipelined Prozessoren zu erreichen, muß das übersetzte Programm sowohl den internen Parallelismus des Prozessors als auch das hierarchische Speichersystem effektiv ausnutzen. Existierende parallelisierende Fortran-Compiler benutzen in der Regel zwei verschiedene Programmrepräsentationen: Auf Hochsprachenebene werden Optimierungen für das Speichersystem, auf der Instruktionsebene Optimierungen für eine bessere Ausnutzung des prozessorinternen Parallelismus durchgeführt. Die Übersetzung des Programms erfolgt dann in zwei Stufen. Dies hat den Nachteil, daß die notwendigen Programmanalysen zweifach durchgeführt werden müssen und die Optimierungen nur eingeschränkt aufeinander abgestimmt werden können. Zur Zeit wird daran gearbeitet, die Optimierungen auf einer gemeinsamen Abstraktionsebene durchzuführen [9]. Die Optimierungen des zur Verfügung stehenden DEC Fortran-Compilers (Version 3.3) [3] sind in erster Linie auf die Instruktionsebene ausgerichtet, was besonders bei Algorithmen mit hoher Datenlokalität zu Performance-Werten führt, die weit unterhalb der erreichbaren Performance liegen.

### 3.1 Optimierung auf Instruktionsebene

Die Untersuchung des DEC Fortran-Compilers ergibt, daß neben den Optimierungen, die auch bei skalaren Prozessoren zu einer Performance-Steigerung führen (*common subexpression elimination, dead code elimination etc.*), insbesondere der effektiven Ausnutzung des Parallelismus auf Instruktionsebene (*Instruction-Level*

*Parallelism, ILP*) ein besonderes Gewicht zufällt. Das Instruktions-Scheduling zielt darauf ab, eine möglichst effiziente Instruktionsreihenfolge zu generieren [8]. Zur Verdeutlichung dient der folgende Programm-Code[1].

```
DO J = 1, N
   DO I = 1, 4*N
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

```
DO J = 1, N
   DO I = 1, 4*N-3, 4
      A(I)   = A(I)   + B(J)
      A(I+1) = A(I+1) + B(J)
      A(I+2) = A(I+2) + B(J)
      A(I+3) = A(I+3) + B(J)
   ENDDO
ENDDO
```

Den Quelltext auf der linken Seite übersetzt der benutzte DEC Fortran-Compiler bei Angabe der Optimierungsstufe **-O2** in einen Basisblock, der inklusive Verzweigungsbefehl sieben Assembler-Instruktionen umfaßt. Unter der Voraussetzung, daß alle referenzierten Daten im internen Cache-Speicher enthalten sind, benötigt der DEC Alpha-Chip zur Durchführung einer Iteration der Schleife neun Takte. Auf der rechten Seite des Beispiels ist die vierfach entfaltete Schleife dargestellt, die vom Compiler bei Angabe der Optimierungsstufe **-O3** bzw. **-O4** generiert wird. Der entsprechende Basisblock (4 Iterationen) enthält insgesamt 26 Assembler-Instruktionen die innerhalb von 15 Takten ausgeführt werden. Dies entspricht einer Leistungssteigerung von $\frac{9 \cdot 4}{15} = 2,4$.

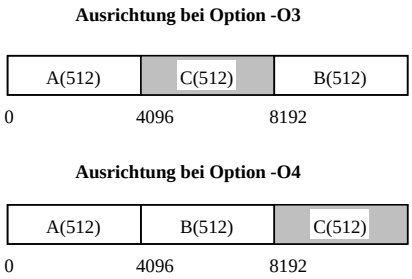
Die Vergrößerung des Basisblocks wird durch die Entfaltung der inneren Schleife eines Schleifennestes erreicht. Eine stärkere Entfaltung würde in dem obigen Beispiel zu einer weiteren Leistungssteigerung führen. Dies kann jedoch nur durch explizite Programmierung des Anwenders erfolgen, da der Compiler lediglich eine vierfache Entfaltung der inneren Schleife ausführt. Dabei resultiert eine zweifach entfaltete programmierte Schleife bei Angabe der Optimierungsstufen **-O3** bzw. **-O4** in einer achtfach entfalteten Schleife. Die obere Grenze der sinnvollen Entfaltung ist durch die Anzahl der zur Verfügung stehenden Fließpunktregister bestimmt. Hierauf wird im Abschnitt Anwendungen näher eingegangen. Die Auswirkungen von Lesefehlern im Cache-Speicher auf die Rechenzeit soll an dem obigen Beispiel exemplarisch betrachtet werden. Unter der Annahme, daß alle von `A(I)` referenzierten Daten aus dem Hauptspeicher geladen werden müssen, erhöht sich die Berechnungszeit um den Betrag der Speicherlatenzzeit. Im Fall der Workstation Model 300 bedeutet dies ein Zuwachs von mindestens 30 Takten. Vergleicht man wiederum die Ausführungszeit der obigen Programmstücke, so läßt sich eine Verminderung der Leistungssteigerung auf einen Wert von 1.4 errechnen. Hieraus ergibt sich, daß die Verminderung der Cache-Speicherlesefehler, neben der Ausnutzung des internen Parallelismus, die zweite wichtige Voraussetzung für das Erreichen höherer Performance-Werte darstellt. Die ungewollte Verdrängung von Cache-Speicherinhalten kann durch die folgenden Maßnahmen vermindert werden:

- Effizientes Ausrichten der Felder zur Übersetzungszeit (*Padding*).
- Änderung der Schleifeniterationsfolge durch Schleifentransformationen.

Die Schleifentransformationen werden im nächsten Abschnitt am Beispiel der Matrixmultiplikation beschrieben.

Zur Erläuterung des ersten Punktes soll der Programm-Code in Abbildung 2 betrachtet werden. Aus dem oberen Teil der Abbildung geht hervor, daß die jeweils 4096 Byte großen Felder `A`, `C` und `B` hintereinander im Hauptspeicher abgelegt sind und die Ausdrücke `A(I)` und `B(I)` in derselben Iteration Feldelemente referenzieren, die 8192 Byte weit auseinander liegen, was exakt der Größe des internen Daten-Cache-Speichers entspricht. Bei der Ausführung dieses Programmstücks ist bei der Übersetzung mit niedrigeren Optimierungsoptionen als **-O4** eine starke Erhöhung der Lesefehler im internen Cache-Speicher zu beobachten, da die Referenzierungen von `A(I)` und `B(I)` dieselbe Cache-Speicherzeile adressieren und damit `B(I)` in jeder Iteration aus dem externen Speicher geladen werden muß. Bei der Benutzung der Optimierungsstufe **-O4**

[1] Bei Schleifengrenzen, die nicht ganzzahlig durch den Entfaltungsfaktor (in diesem Fall 4) teilbar sind, ist eine zusätzliche Clean-Up-Schleife einzuführen, in der die verbleibenden Iterationen ausgeführt werden.




```
REAL*8 A(512),C(512),B(512)
DO I=1, 512
  A(I)=A(I)+B(I)
ENDDO
```

**Abb. 2.** Verminderung von Lesefehlern im Cache-Speicher durch Änderung der Feldausrichtung im Hauptspeicher

werden die Felder `A`, `C` und `B` durch den Compiler effizient ausgerichtet, wie im unteren Teil der Abbildung 2 dargestellt. Werden COMMON-Blöcke verwendet und wird bei der Übersetzung der Parameter **-align dcommons** angegeben, so wird das erste Feld eines Blockes auf eine 32-Byte-Grenze ausgerichtet und die folgenden Felder dieses COMMON-Blockes sequentiell im Hauptspeicher abgelegt. Durch das Einfügen von zusätzlichen Datenfeldern oder der Vergrößerung der einzelnen Felddimensionen kann ein effizientes Ausrichten der Felder auch manuell durch den Benutzer erfolgen. Die Ausrichtung von COMMON-Blöcke wird vom Compiler in keinem Fall geändert.

### 3.2 Optimierung auf Hochsprachen-Ebene

Durch eine zu hohe Wartezeit bei dem Zugriff auf Daten aus dem Hauptspeicher wird der Parallelismus auf Instruktionsebene aufgehoben, und die erreichbare Rechenleistung ist nicht größer als im skalaren Fall. Eine Lösung zur Vermeidung dieses Datenengpasses besteht in der Reduktion der notwendigen Hauptspeicherzugriffe durch die Ausnutzung der Datenlokalität in den Cache-Speichern. Dies kann durch die Anwendung von Programmtransformationen auf Hochsprachenebene (Schleifenentfaltung und Schleifenblockung) erreicht werden, die im nächsten Abschnitt am Beispiel der Matrixmultiplikation beschrieben werden.

Unter der Datenlokalität in Rechnern mit verteiltem Speicher versteht man die Präsenz von Daten in den lokalen Speichern der einzelnen Prozessoren. Im folgenden wird als Datenlokalität die Anwesenheit der Daten in der betrachteten Speicherhierarchieebene (Register und interner Cache-Speicher) definiert. Die Datenlokalität im Cache-Speicher führt an sich noch nicht zur Reduktion von Hauptspeicherzugriffen. Sind beispielsweise mit einem Ladebefehl vier 64-Bit-Daten in eine Cache-Speicherzeile geladen worden, aber auf nur ein Datum wird zugegriffen, so besteht für die anderen drei Daten zwar Datenlokalität, sie wird jedoch nicht ausgenutzt. Erst mit der Ausnutzung der Datenlokalität sind Performance-Gewinne zu erzielen, da Hauptspeicherzugriffe eingespart werden. Das Ausmaß der möglichen Datenlokalität ist abhängig von der verwendeten Speicherarchitektur bzw. von der Größe der betrachteten Speicherhierarchieebene: Je größer beispielsweise der Cache-Speicher ist, desto mehr Daten können gleichzeitig lokal sein.

Das Ziel ist es nun, bereits zur Übersetzungszeit beurteilen zu können, welche Feldreferenzen auf dasselbe Feldelement zugreifen und durch welche Änderung der Iterationsfolge diese Referenzen zeitlich näher zusammengebracht werden können. Um festzustellen, welche Feldelemente zu welcher Zeit lokal sind, kann die Datenabhängigkeitsanalyse zur Erkennung der Wiederverwendung von Feldelementen benutzt werden. Es existieren mehrere Arten der Wiederverwendung, von denen die selbst-zeitliche und selbst-örtliche im folgenden erläutert werden.

- Es besteht **selbst-zeitliche Wiederverwendung**, wenn eine Feldreferenz auf dasselbe Feldelement in verschiedenen Iterationen zugreift.
- Es besteht **selbst-örtliche Wiederverwendung**, wenn eine Feldreferenz auf Feldelemente innerhalb einer Cache-Speicherzeile in verschiedenen Iterationen zugreift.

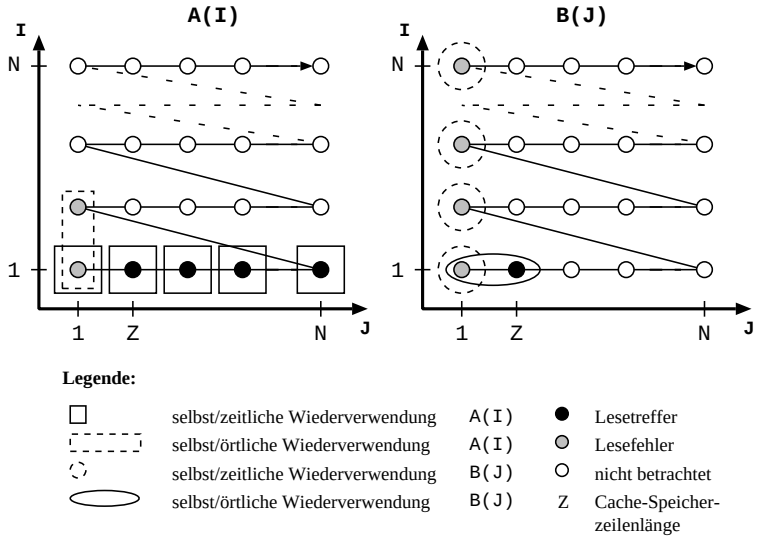



**Abb. 3.** Selbst-zeitliche und selbst-örtliche Wiederverwendung

In Abbildung 3 werden mit dem Ladebefehl `A(1)` in der Iteration (1,1) auch das Element `A(2)` in dieselbe Cache-Speicherzeile geladen[2]. Da jedoch in der inneren Schleife sämtliche Elemente des Feldes `B(J)` in den Cache-Speicher geladen werden und `N` als so groß angenommen wird, daß nicht alle Daten von `B(J)` in den internen Cache-Speicher passen, wird `A(2)` überschrieben sein, bevor die Datenlokalität ausgenutzt werden kann[3]. Im Gegensatz dazu kann die Datenlokalität des Datums `B(2)` in der Iteration (1,2) ausgenutzt und die Anzahl der notwendigen Ladezugriffe für `B(J)` um den Faktor 1/Z (Z=2) reduziert werden, wobei Z die Cache-Speicherzeilenlänge ist. Die bestehende Wiederverwendung der Feldreferenzen `A(I)` und `B(J)` ist in Abbildung 3 dargestellt. Wiederverwendung, die zu ausnutzbarer Datenlokalität führt, ist mit durchgezogenen Linien gekennzeichnet, Wiederverwendung von Feldelementen, die zu keiner ausnutzbaren Datenlokalität führt, ist gestrichelt eingetragen. In dem nächsten Abschnitt werden Schleifentransformationen vorgestellt, die die Iterationsfolge ändern und damit die ausgenutzte Datenlokalität der Feldelemente erhöhen.

## 4 Anwendungen

Anhand von zwei Beispielen, der Matrixmultiplikation und einem Simulationsalgorithmus, wird in diesem Abschnitt das Optimierungspotential von Programmen beschrieben, die mit der höchsten Optimierungsstufe des Fortran-Compilers übersetzt worden sind. Dabei stellt die Matrixmultiplikation zur Anwendung der Transformationen Schleifenvertauschen, -blocken und -entfalten ein geeignetes Beispiel dar, weil keine Datenabhängigkeiten die Permutation der Schleifen verhindern und eine hohe Wiederverwendung besteht, die wesentliche Performance-Verbesserungen gegenüber dem untransformierten Code erwarten läßt. Im Fall der realen Anwendung ist die Wiederverwendung geringer, dennoch kann eine erhebliche Performance-Steigerung durch Schleifenentfalten erzielt werden.

[2] Unter der Voraussetzung, daß `A(1)` im Hauptspeicher auf einer 32-Byte-Grenze ausgerichtet ist.

[3] Dabei wird angenommen, daß `A(1)` während der Iterationen (1,1),...,(1,`N`) in einem Register gehalten wird und somit kein Cache-Speicherzugriff bezüglich `A(I)` erfolgt. Ohne diese Annahme würde die Datenlokalität von `A(2)` ausgenutzt werden, da `A(1)` in Iteration (1,`N`) im Cache-Speicher resident ist und damit auch `A(2)` in der folgenden Iteration (2,1).

### 4.1 Matrixmultiplikation

Welche Performance-Steigerung bei der Matrixmultiplikation auf der untersuchten Alpha-Workstation erzielt werden kann, wird deutlich, wenn man den nichtoptimierten Code mit der DEC DGEMM-Routine[4] aus der Digital Extended Math Library für DEC OSF/1 AXP vergleicht.

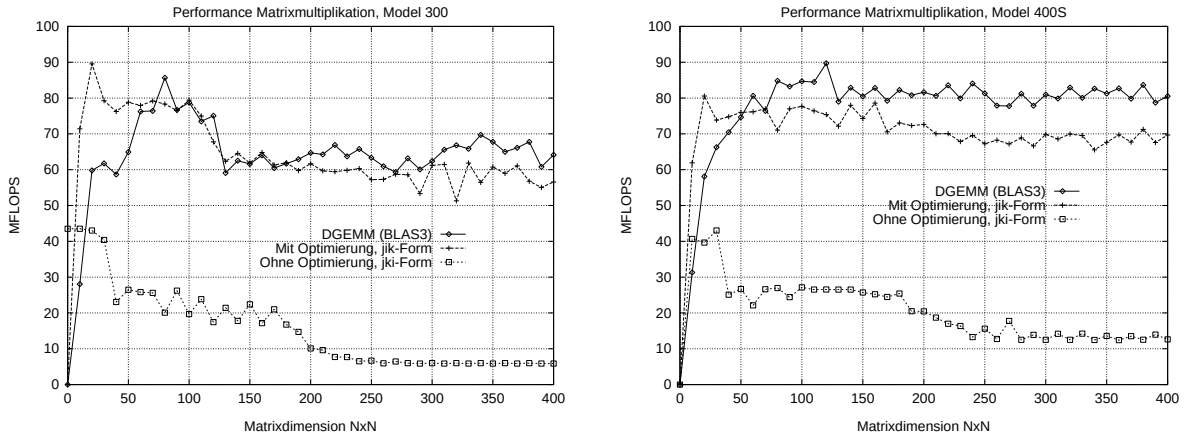



**Abb. 4.** Performance-Messung Matrixmultiplikation

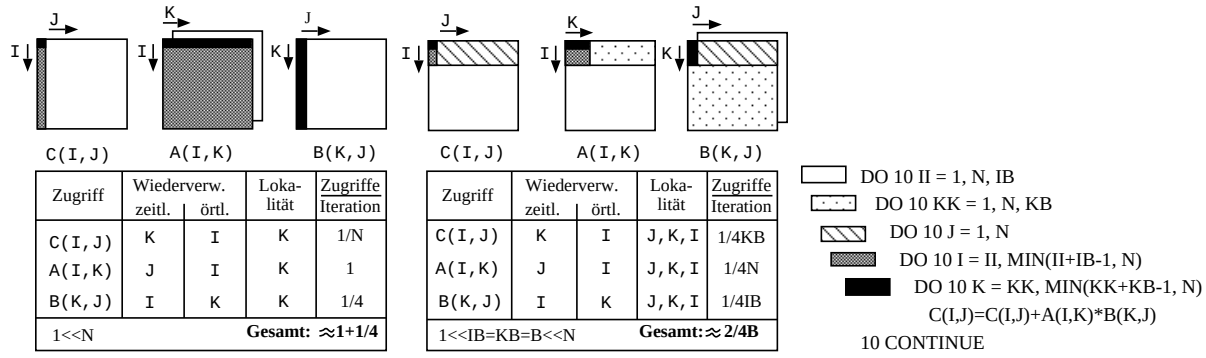
Aus Abbildung 4 geht hervor, daß der Mittelwert der erreichten Performance für quadratische Matrizen mit 64-Bit-Fließpunktwerten auf der Alpha-Workstation Model 300 für $\texttt{N} > 130$ bei ca. 60 MFLOPS liegt, was eine Leistungssteigerung gegenüber der nichttransformierten Version um den Faktor 6 bedeutet. Der Performance-Abfall bei der Dimension $\texttt{N} = 130$ kann auf die schlechtere Ausnutzung des internen Cache-Speichers zurückgeführt werden. Die Schwankungen in dem Verlauf der Kurven läßt sich auf das unterschiedliche Auslöschungsverhalten bei unterschiedlichen Feldgrößen aufgrund der anderen Abbildung auf den Cache-Speicher zurückführen. Da die Alpha-AXP Architektur bisher keine Instruktionen vorsieht, die eine Kontrolle der Cache-Speicherbelegung ermöglichen, ist die ungewollte Aufnahme von Daten in den Cache-Speicher nicht zu vermeiden. Dies führt zu einer erheblichen Reduktion der effektiv ausnutzbaren Cache-Speichergröße und damit zur Verringerung der erreichbaren Performance. Die nicht transformierte *JKI*-Version der Matrixmultiplikation zeigt für $\texttt{N} > 200$ einen konstanten Performance-Wert von ca. 8 MFLOPS. Der Verlauf der Kurve ist deshalb so glatt, weil für keine Problemgröße die zeitliche Wiederverwendung von `A(I,K)` und `C(I,J)` ausgenutzt wird und somit Interferenzen in beiden Cache-Speichern keine sichtbaren Auswirkungen haben. In dem Bereich bis $\texttt{N} = 200$ macht sich die unterschiedliche Ausnutzung der Datenlokalität bemerkbar.

Der Vergleich der Performance-Kurven für die beiden unterschiedlichen Workstation-Modelle ergibt, daß auch der optimierte Matrixmultiplikation-Code in der Performance von der Schnelligkeit des Datentransfers zwischen den einzelnen Speicherhierarchien begrenzt ist: Trotz der niedrigeren Taktrate von 133 MHz werden auf der Workstation Model 400S höhere Performance-Werte erzielt als auf dem mit 150 MHz getakteten Rechner. Desweiteren ist zu erkennen, daß auf beiden Rechnern der vom Benutzer optimierte Programm-Code hinsichtlich der Performance-Werte nahe an die DEC DGEMM-Routine heranreicht. Die Gründe für diese Performance-Verbesserung werden im folgenden näher erläutert.

In Abbildung 5 ist dargestellt, auf welche Weise das Blocken von Schleifen bei der Matrixmultiplikation zu einer Erhöhung der Datenlokalität führt und damit die Grundlage für eine erhebliche Performance-Steigerung bildet. Am Beispiel der *JIK*-Form soll die graphische Darstellung und die Berechnung der Wiederverwendung erläutert werden. Dabei wird angenommen, daß die Datenlokalität nur in der inneren Schleife besteht, d.h. $\texttt{N} \gg 1$.

Das Feld `B(K,J)` wird spaltenweise durchlaufen. Die Spaltenelemente werden in der mittleren Schleife `N`-fach wiederverwendet, was jedoch nicht ausgenutzt werden kann, da das erste Element `B(1,1)`, welches in

[4] DGEMM berechnet das Matrix-Matrix Produkt für 64-Bit-Werte (Double Precision).




| Zugriff | Wiederverw. zeitl. | Wiederverw. örtl. | Lokalität | Zugriffe Iteration |
|---|---|---|---|---|
| C(I,J) | K | I | K | 1/N |
| A(I,K) | J | I | K | 1 |
| B(K,J) | I | K | K | 1/4 |
| 1<<N | | | Gesamt: | ≈1+1/4 |

| Zugriff | Wiederverw. zeitl. | Wiederverw. örtl. | Lokalität | Zugriffe Iteration |
|---|---|---|---|---|
| C(I,J) | K | I | J,K,I | 1/4KB |
| A(I,K) | J | I | J,K,I | 1/4N |
| B(K,J) | I | K | J,K,I | 1/4IB |
| 1<<IB=KB=B<<N | | | Gesamt: | ≈2/4B |

**Abb. 5.** Matrixmultiplikation: *JIK*-Form, ungeblockt bzw. innere und mittlere Schleife geblockt

der Iteration (1,1,1) geladen wurde, sich zum Zeitpunkt der selbst-zeitlichen Wiederverwendung in (1,2,1) aufgrund der obigen Annahme nicht mehr im Cache-Speicher befindet. Dagegen führt die selbst-räumliche Wiederverwendung zu ausnutzbarer Lokalität: Mit `B(1,1)` werden gleichzeitig die nächsten Spaltenelemente `B(2,1)`, `B(3,1)` und `B(4,1)` in dieselbe Zeile geladen, so daß der externe Speicherzugriff für diese Elemente in den nächsten drei Iterationen entfällt. Damit reduziert sich die Anzahl der notwendigen Ladeoperationen für `B(I,J)` von 1 auf 1/4 Ladeoperationen pro Iteration. Das Feld `A(I,K)` wird zeilenweise durchlaufen und die Feldelemente erst in der äußeren Schleife `N`-fach wiederverwendet. Für die Ausnutzung der Datenlokalität durch die selbst-zeitliche Wiederverwendung muß die gesamte Matrix im Cache-Speicher Platz finden, was laut Annahme nicht möglich ist. Das Feld `C(I,J)` wird spaltenweise durchlaufen. Der Index von `C(I,J)` ändert sich in der inneren Schleife nicht, so daß der Wert in einem Register gehalten werden kann. Die Ausnutzung der resultierenden Datenlokalität reduziert die Hauptspeicherzugriffe pro Iteration für `C(I,J)` um den Faktor 1/`N`. Das Auftreten des Ausdrucks `C(I,J)` auf der rechten Seite der Anweisung hat keine Auswirkung auf die Berechnung der Hauptspeicherzugriffe, da die Datenlokalität von Daten im Cache-Speicher nur die Anzahl der Ladeoperationen reduziert. Auf Speichervorgänge hat die Datenlokalität keinen Einfluß. Beim Addieren der Zugriffe kann der Faktor 1/`N` gegenüber 1/4 gemäß der Voraussetzung $N \gg 1$ vernachlässigt werden. Durchschnittlich sind also pro Iteration nur 0.5 Ladeoperationen aus dem externen Speicher notwendig.

Auf der rechten Seite der Abbildung 5 ist die Iterationsfolge des doppelt geblockten Programm-Codes dargestellt. Die Blockung der beiden inneren Schleifen ermöglicht den maximalen Grad an ausgenutzter Datenlokalität. Anstelle zweier identischer Blockfaktoren können auch unterschiedliche Werte benutzt werden, so daß das Zugriffsmuster für das Feld `A(I,K)` eine rechteckige Form annimmt. Dies hat den Vorteil, daß das Zugriffsmuster auf die Felder noch feiner variiert werden kann. Die Effizienz dieser Maßnahme und der Einfluß unterschiedlicher Blockfaktoren wird in [5] näher untersucht.

Schleifentransformationen allein ergeben jedoch noch nicht die erwünschte Performance-Steigerung. Messungen ergaben für die alleinige Anwendung der Blocktransformationen Performance-Werte von nur 20 MFLOPS. Erst mit der zusätzlichen Entfaltung der Schleifen können Performance-Werte von 60 MFLOPS erreicht werden.

Die günstigsten Entfaltungsfaktoren ergeben sich aus mehreren Randbedingungen.

- Um möglichst viele Speicher- und Schreibzugriffe zu vermeiden, sollten während der Ausführung der inneren `K`-Schleife möglichst viele Feldelemente von `C(I,J)` in Registern gehalten werden.
- Die `I`-Schleife sollte vierfach entfaltet werden, damit die örtliche Wiederverwendung der Feldelemente `C(I+1,J)`,...,`C(I+3,J)` und `A(I+1,K)`,...,`(I+3,K)` ausgenutzt werden kann.
- Die Anzahl der zur Verfügung stehenden Fließpunktregister bildet die obere Grenze für die Wahl der Entfaltungsfaktoren.

Diese Forderungen führen zu dem Ergebnis, die `J`-Schleife fünffach und die `I`-Schleife vierfach zu entfalten. Damit werden die Register im Basisblock der `K`-Schleife wie folgt belegt:

- 4 Register mit Daten aus `A(I,K)`,
- 5 Register mit Daten aus `B(K,J)` und
- 20 Register mit Daten aus `C(I,J)`.

Eine stärkere Entfaltung der Schleifen würde eine Zwischenspeicherung der Registerinhalte im Speicher notwendig machen und die Performance entsprechend vermindern. Bei Anwendung aller beschriebenen Transformationen sind dann die Performance-Werte aus Abbildung 4 gemessen worden. Die Durchführung der gleichzeitigen Schleifenblockung und -entfaltung hat sich als sehr mühsam und fehleranfällig herausgestellt. Zudem vergrößerte sich der einfache Programm-Code inklusive der Variablendeklaration von 9 auf 222 Zeilen, was einen erheblichen Zuwachs an Speicherbedarf bedeutet und eine Wartung des Programm-Codes nahezu unmöglich macht. Die neu entstandenen Programmzeilen werden durch die starke Entfaltung und die notwendigen zusätzlichen Clean-Up-Schleifen verursacht.

### 4.2 Simulation des Kristallwachstums

Als Beispiel für das Optimierungspotential von realen Applikationen wurde der Programm-Code für die Simulation von Kristallwachstumsprozessen untersucht [7], der bereits von anderen Mitarbeitern des Instituts für Angewandte Mathematik parallelisiert wurde [1, 4]. Die Züchtung von Silizium-Einkristallen geschieht in der Regel nach dem Czochralski-Verfahren. Silizium wird in einem rotierenden Quarztiegel erhitzt und wächst um einen ebenfalls rotierenden Kristallisationspunkt zu einer Siliziumscheibe, von der später die *Wafer* für die Halbleiterproduktion abgeschieden werden. Eine Störquelle beim Herstellungsprozeß stellt der Sauerstoff dar, der laufend aus den Quarztiegelwänden herausgelöst und durch Diffusion und Strömungsprozesse in die Schmelze und in den wachsenden Kristall transportiert wird. Aufgrund der zeitlichen Schwankung der Strömungsvorgänge schwankt auch der Einbau von Sauerstoff in den wachsenden Kristall. Da der Sauerstoffeinbau in den *Wafer* von entscheidender Bedeutung für die spätere Qualität der integrierten Schaltkreise ist und das Auftreten von Sauerstoff nicht vermieden werden kann, sollte der Einbau in möglichst gleichmäßiger Konzentration erfolgen. Da experimentelle Information über die Transportprozesse aufgrund der hohen Temperaturen in der undurchsichtigen Schmelze außerordentlich schwierig zu erhalten sind, können diese Vorgänge derzeit im wesentlichen nur durch rechnerische Simulation erforscht werden. Das mathematische Modell, welches für diesen Prozeß erstellt wurde, ist durch ein System von gekoppelten partiellen nichtlinearen Differentialgleichungen definiert, das nur numerisch gelöst werden kann.

Der simulierte Tiegel, er mißt 3 cm im Radius und 4 cm in der Höhe, wird in 30 x 90 x 40 Elemente diskretisiert. Der Algorithmus besteht aus Initialisierungsphase, Zeitschleife und Ausgabe. Die Zeitschleife, in der für jeden Zeitschritt die Temperatur, der Druck und die Geschwindigkeit neu berechnet werden, beansprucht den größten Anteil der gesamten Ausführungszeit. In diesem Programmteil wird das lineare Gleichungssystem, das aus der Diskretisierung der partiellen Differentialgleichungen hervorgeht, gelöst. Die wesentlichen Operationen bilden, neben den Datenbewegungen zum Austausch der Randbedingungen, Differenzensterne auf den Datenfeldern, d.h. zur Berechnung eines Feldelementes werden nur die Werte der Nachbarelemente benötigt. Die selbst-zeitliche Wiederverwendung berechnet sich aus der Größe des Differenzensterns bzw. der Anzahl der beteiligten Feldelemente, beim Austauschen der Randbedingungen findet keine Wiederverwendung statt. Zusammengenommen ist die Wiederverwendung deutlich geringer als bei der Matrixmultiplikation. Die Vermutung, daß durch das Blocken der Schleifen keine Performance-Steigerung zu erreichen sind, konnte durch Messungen belegt werden. Durch Schleifenentfaltung und *Padding* konnten dagegen signifikante Verbesserungen erzielt werden, wie die Werte in Tabelle 2 belegen.

| **Modellbezeichnung** | **DEC 3000 Model 300 AXP** | **DEC 3000 Model 400S AXP** | **CRAY T3D** |
|---|---|---|---|
| **Optimierung -O4** | 150 s | 84 s | 91 s |
| **+ Schleifen entfalten** | 60 s | 40 s | 65 s |
| **+ Padding** | 54 s | 36 s | 61 s |

**Tabelle 2:** Performance-Ergebnisse des Simulationsprogramms

Es ist anzumerken, daß die Programme für den CRAY T3D mit dem CRAY Fortran-Compiler übersetzt wurden. Die geringeren Performance-Werte im Vergleich zu den Workstations sind auf den fehlenden externen Cache-Speicher zurückzuführen.

## 5 Zusammenfassung

Der Trend zu immer schneller getakteten Prozessoren mit wachsendem prozessorinternen Parallelismus läßt das I/O-Verhalten zum kritischen Engpaß eines Rechners werden. Unter Beibehaltung herkömmlicher Speichertechnik ohne ausreichend große Cache-Speicher führt dies auch bei massiv-parallelen Rechnern mit modernen superskalaren und superpipelined RISC-Prozessoren zu unbefriedigender Performance, auch wenn dies durch die Verwendung von Speed-Up-Werten zur Charakterisierung der Performance oft verborgen bleibt.

Die Reduktion der Hauptspeicherzugriffe durch Ausnutzung der Datenlokalität kann, wie in diesem Artikel gezeigt, zu einer Performance-Steigerung führen, die für die Matrixmultiplikation einem Faktor von ca. 6 und für eine reale Applikation immerhin einem Faktor von 1,5 bis 2,8 entspricht. Jedoch sind die durchzuführenden Programmänderungen sehr fehleranfällig und zeitaufwendig. Daher besteht die Forderung, wichtige Schleifentransformationen, wie etwa Schleifenentfaltung und Schleifenblockung, vom Compiler automatisch durchführen zu lassen. Vergleicht man die Performance-Ergebnisse der untersuchten Rechner miteinander, so wird darüberhinaus die Wichtigkeit eines der Prozessorleistung angepaßten Speichersystems deutlich.

### Dank

Dem Institutsleiter Herrn Prof. Dr. F. Hoßfeld danke ich für die Möglichkeit, meine Diplomarbeit, die die Grundlage für diesen Artikel bildet, am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich (KFA) anfertigen zu können. Mein besonderer Dank gilt Herrn Dr. M. Gerndt, Herrn Dr. W.E. Nagel und Herrn Dr. P. Weidner für viele konstruktive Anregungen. Für die Unterstützung bei der Durchführung der Performance-Messungen auf den DEC Alpha-Workstations bedanke ich mich bei Herrn Anrath. Herrn Dr. R. Vogelsang von der Firma CRAY Research Inc. danke ich für die Unterstützung bei der Optimierung des Simulations-Codes und die Möglichkeit, Messungen auf dem Rechner CRAY T3D durchführen zu können.

## Literatur

1. R. Berrendorf, H.C. Burg, U. Detert, R. Esser, M. Gerndt, R. Knecht. *Intel Paragon XP/S - Architecture, Software Environment, and Performance.* Interner Bericht KFA-ZAM-IB-9409, Forschungszentrum Jülich, Mai 1994.
2. *Alpha architecture handbook.* Digital Equipment Corporation, 1992.
3. *DEC Fortran for DEC OSF/1 AXP systems, user manual.* Digital Equipment Corporation, 1993.
4. M. Gerndt. *Automatic Parallelization of a Crystal Growth Simulation Program for Distributed-Memory Systems.* Proceedings of HPCN Europe, München, LNCS 796, 1994, pp. 281-286.
5. A. Krumme. *Performance-Analyse des DEC Alpha-Chip: Moderne Compiler-Techniken zur Programmoptimierung.* Forschungszentrum Jülich, Jül-2912, April 1994.
6. E. McLellan. *The Alpha AXP architecture and 21064 processor.* IEEE Micro, June 1993, pp. 152-168.
7. M. Mihelcic, H. Wenzl, K. Wingerath. *Flow in Czochralski Crystal Growth Melts.* Bericht des Forschungszentrums Jülich, No. 2697, ISSN 0366-0885, December 1992.
8. B.R. Rau, J.A. Fisher. *Instruction-level parallel processing: history, overview and perspective.* The Journal of Supercomputing, Vol. 7, No. 1/2, 1993, pp. 9-50.
9. S.W.K. Tjiang, M.E. Wolf, M.S. Lam, K. Pieper, J.L. Hennessy. *Integrating scalar optimization and parallelization.* Computer Systems Laboratory, Stanford University, February 1992.

This article was processed using the LaTeX macro package with LLNCS style