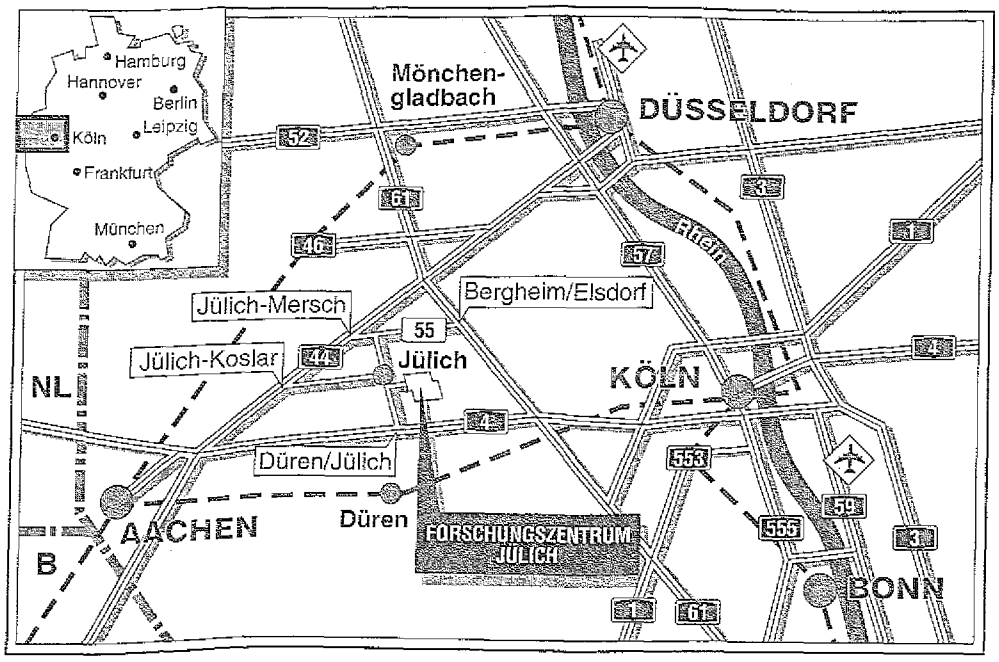
FORSCHUNGSZENTRUM JÜLICH GmbH

Zentralinstitut für Angewandte Mathematik

Vergleichende Untersuchung von synchronen und asynchronen Algorithmen und deren Implementierung auf CRAY-Mehrprozessorsystemen

Heike Schmitz



Berichte des Forschungszentrums Jülich ; 2426
ISSN 0366-0885
Zentralinstitut für Angewandte Mathematik Jül-2426

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
Postfach 1913 · D-5170 Jülich · Bundesrepublik Deutschland
Telefon: 02461/61-6102 · Telefax: 02461/61-6103 · Telex: 833556-70 kfa d

Vergleichende Untersuchung von synchronen und asynchronen Algorithmen und deren Implementierung auf CRAY-Mehrprozessorsystemen

Heike Schmitz

Zusammenfassung

In dieser Arbeit werden parallele synchrone und asynchrone Algorithmen untersucht und anhand charakteristischer Merkmale verglichen. Das Hauptcharakteristikum synchroner Algorithmen ist die notwendige Synchronisation parallel ablaufender Prozesse zu bestimmten Zeitpunkten der Ausführung. Dadurch entsteht zusätzlicher Zeitaufwand, und eventuell werden Wartezeiten durch unterschiedlich lange Ausführungszeiten der einzelnen Prozesse verursacht. Diesen zusätzlichen Zeitaufwand (Synchronisations-Overhead) vermeiden asynchrone Algorithmen. Ihr Hauptmerkmal ist der Verzicht auf Synchronisation, so daß die parallelen Prozesse unabhängig voneinander Berechnungen mit den gerade zur Verfügung stehenden Daten durchführen.
Im Rahmen dieser Arbeit werden ausgewählte iterative Verfahren zur Lösung linearer und nichtlinearer Gleichungssysteme jeweils in einer synchronen und asynchronen Version auf CRAY-Mehrprozessorsystemen implementiert und deren Leistungsverhalten gemessen. Als Fazit dieser Untersuchungen wird festgestellt, daß die asynchronen Verfahren für diese Anwendungen keine überragenden Vorteile gegenüber den synchronen Verfahren in Bezug auf das Laufzeitverhalten bieten.

Inhaltsverzeichnis

1.0 Einleitung

In den letzten Jahrzehnten hat sich herausgestellt, daß Parallelverarbeitung dasjenige Konzept ist, welches die größten Möglichkeiten in Bezug auf Leistungssteigerung beinhaltet [H&J,88].
Dies beruht vor allen Dingen auf der Tatsache, daß die Entwicklung bei der Verbesserung von Schaltzeiten und Signal-Ausbreitungsgeschwindigkeiten physikalisch begrenzt ist und damit auch die Geschwindigkeit eines einzelnen Prozessors determiniert ist. So existiert heute eine große Anzahl von Mehrprozessorsystemen verschiedener Typen und verschiedener Leistungsmerkmale.

Das Konzept der Parallelverarbeitung stellt auch eine Herausforderung dar an die Entwicklung guter paralleler Algorithmen, denn dadurch können die Möglichkeiten paralleler Rechnerarchitekturen erst genutzt werden [Hoß,83].
Mit der parallelen Ausführung eines Programms wird die Ausführungszeit verkürzt, jedoch wird die insgesamt zu leistende Arbeit nicht verringert. Es entsteht sogar noch zusätzlicher Arbeitsaufwand, um die parallele Ausführung zu koordinieren. Diese Koordination beinhaltet auch eine Synchronisation zwischen parallel ablaufenden Prozessen. Das bedeutet, daß zu gewissen Zeitpunkten in der Programmausführung die Prozesse ihre Berechnungen erst dann weiter fortführen können, wenn alle Prozesse an diesen Punkten der Programmausführung angelangt sind. Dies impliziert ein Warten der schnelleren Prozesse auf die langsameren Prozesse und damit auch Idle-Zeiten (Leerlauf-Zeiten). Synchronisation ist somit eine der Hauptursachen für Leistungsminderung in Mehrprozessorsystemen.

Parallele Algorithmen, die eine Synchronisation zu gewissen Zeitpunkten benötigen heißen auch *synchrone Algorithmen*. Synchronisation in einem parallelen Algorithmus garantiert eine weitgehend deterministische Berechnungsreihenfolge, die der Reihenfolge der Berechnungen im entsprechenden seriellen Algorithmus häufig entspricht. Jedoch müssen parallele Algorithmen nicht unbedingt aus seriellen Algorithmen heraus entwickelt werden. In einer parallelen Umgebung sollte man sich vom seriellen Denken loslösen, um Effizienz zu erreichen [Hoß,83].

Um die Nachteile der synchronen Algorithmen zu umgehen, wurden in den beiden letzten Jahrzehnten *asynchrone Algorithmen* entwickelt. In asynchronen Algorithmen wird fast ganz auf die Synchronisation zwischen Prozessen verzichtet. Es ist hier nicht notwendig, daß an bestimmten Stellen der Programmausführung eines Prozesses auf Daten von anderen Prozessen gewartet wird, denn die Berechnungen laufen ohne Unterbrechung mit den gerade zu diesem Zeitpunkt von anderen Prozessen zur Verfü-

gung gestellten Daten weiter. Dies führt dazu, daß durch Auslassen der Wartezeiten mehr Berechnungen in der gleichen Zeit durchgeführt werden können und damit eine schnellere Ausführung des Algorithmus erzielt werden kann. Eine vorhersehbare Berechnungsreihenfolge kann bei asynchronen parallelen Algorithmen nicht garantiert werden. Hier besteht die Gefahr, daß Berechnungen, die nicht auf den eigentlich vom Algorithmus geforderten Daten beruhen, sich nicht optimal im Hinblick auf das Ziel des Algorithmus verhalten. Jedoch hat sich gezeigt, daß, wenn gewisse Bedingungen eingehalten werden, asynchrone Algorithmen korrekt arbeiten und leistungsfähig im Vergleich zu den synchronen Algorithmen sein können.

Aus den oben genannten Gründen ist es schwierig vorherzusagen, wie schnell ein asynchroner Algorithmus zu einer Problemlösung führt, und wie das Verhalten in Bezug auf einen adäquaten synchronen Algorithmus ist. Deshalb ist es sinnvoll Ergebnisse hierzu durch Untersuchungen an konkreten Algorithmen in der Praxis durchzuführen.

In der vorliegenden Arbeit wurde eine Implementierung von synchronen und asynchronen Algorithmen zur Lösung von linearen und nichtlinearen Gleichungssystemen auf dem in der KFA Jülich installierten Parallelrechner CRAY Y-MP8/832 vorgenommen, wobei verschiedene Aspekte untersucht wurden.

In Kapitel 2 werden zunächst einige Konzepte der Parallelverarbeitung vorgestellt, wobei auch auf Synchronisationsmechanismen eingegangen wird. Anschließend wird die Systemarchitektur des CRAY-Rechners beschrieben und das CRAY-Multitasking-Konzept dargelegt.
In Kapitel 3 folgt eine Einführung in synchrone und asynchrone Algorithmen und die Gegenüberstellung ihrer charakterisierenden Merkmale.
Kapitel 4 geht auf iterative Verfahren zur Lösung von linearen und nichtlinearen Gleichungssystemen ein, und reißt kurz deren Konvergenzbedingungen an.
In Kapitel 5 wird die Implementierung dieser Verfahren beschrieben und das Leistungsverhalten untersucht.
Kapitel 6 beschreibt das Verhalten der entwickelten Programmversionen im Multiprogramming-Betrieb, und Kapitel 7 gibt eine Zusammenfassung der gewonnenen Erkenntnisse mit einem kurzen Ausblick.

2.0 Konzepte der Parallelverarbeitung

2.1 Entwicklung der Parallelverarbeitung

Der Inhalt dieses Abschnitts beruht im wesentlichen auf [H&J,88].
Der natürlichen sequentiellen Denkweise des Menschen angepaßt weisen die ersten elektronischen Rechner, die ab 1940 gebaut wurden, eine sequentielle Struktur auf. Diese Computer der ersten Generation beruhen auf einer Organisation, die als von Neumann-Organisation bezeichnet wird. Die wesentlichen Merkmale eines von Neumann-Rechners stellen sich wie folgt dar:

- eine Eingabe/Ausgabe Einheit
- ein gemeinsamer Speicher für Daten und Befehle
- eine Kontrolleinheit zur Befehlsinterpretation
- eine arithmetische und logische Einheit (ALU) zur Befehlsausführung

Die beiden letzten Einheiten werden unter dem Begriff CPU (central processing unit) zusammengefaßt. In einer von Neumann-Organisation laufen alle Operationen wie Speicherzugriff, Ein-/Ausgabe, arithmetische oder logische Operation streng sequentiell ab. Diese Struktur hat u.a. den Nachteil, daß bei Speicherzugriffen, die verhältnismäßig langsam ablaufen, die CPU lange Zeit untätig ist, und damit ein großes Potential an Leistungsvermögen ungenutzt bleibt.
Ab Mitte der 60er Jahre entwickelte man deshalb Rechnerstrukturen, die in der Lage waren mehrere Operationen überlappend, bzw. gleichzeitig durchzuführen. So entstand Parallelismus in Rechnerarchitekturen, der sich in mehreren Ansätzen ausdrückte. Die wichtigsten realisierten Ansätze stellen sich wie folgt dar:

- Pipelining (Fließbandverarbeitung)

In einer linearen Anordnung von Funktionseinheiten (Pipe) können Teiloperationen taktsynchron an verschiedenen Daten vorgenommen werden.

- Funktional-Parallelismus

Verschiedene Operationen wie Additionen, Multiplikationen oder logische Operationen können in speziellen Funktionseinheiten der CPU gleichzeitig auf mehreren Daten durchgeführt werden.

- Feld-Parallelismus

Viele zu einem Feld angeordnete, identische Prozessoren führen unter gemeinsamer Kontrolle die gleichen Befehle auf verschiedenen Daten aus.

- Multiprocessing

Mehrere Prozessoren führen zur selben Zeit verschiedene Operationen mit unterschiedlichen Daten durch.

Meist sind mehrere dieser Ansätze in existierende Parallelrechner integriert. So ist die CRAY Y-MP8/832 ein Multiprozessorsystem, welches Vektor-Pipelines und Funktionseinheiten besitzt.

2.2 Parallele Programmierung

Auch die Programmierung entwickelte sich aus sequentiellen Strukturen heraus hin zum Parallelismus. Denn es war notwendig geworden den parallelen Rechnerarchitekturen Rechnung zu tragen, um eine effiziente Nutzung zu erzielen. Der Parallelismus in der Programmierung drückt sich in verschiedenen Ebenen aus:

- Job-Ebene

In Mehrbenutzersytemen versucht man die Anzahl der ausgeführten Jobs zu maximieren indem man mehrere Jobs zur gleichen Zeit bearbeitet und damit den Durchsatz des Systems erhöht.

- Programm-Ebene

In der Programm-Ebene können Teile eines Programms gleichzeitig abgearbeitet werden. Dabei werden hauptsächlich voneinander unabhängige Programmteile und Schleifeniterationen parallel ausgeführt.

- Instruktions-Ebene

Innerhalb einer Anweisung werden die einzelnen Operationen parallel bearbeitet.

Die Job-Ebene wird vom Betriebssystem eines Rechners realisiert, das für eine adäquate Verteilung der verfügbaren Betriebsmittel an die einzelnen Jobs sorgt. Während die Instruktions-Ebene hardwaremäßig installiert ist, kann man die Programm-Ebene der parallelen Programmierung zuordnen. Unter paralleler Programmierung versteht man das Erkennen und Kennzeichnen von unabhängigen, parallel ausführbaren Programmteilen, bzw. Schleifeniterationen. Diese Programmierung kann durch Software automatisiert oder vom Benutzer unterstützt werden. Die Anweisungsfolgen, welche parallel mit anderen Anweisungsfolgen ausgeführt werden können, werden als *Prozesse* bezeichnet. Ein paralleles Programm ist damit eine Kollektion von Prozessen, wobei die Ausführung innerhalb eines Prozesses streng sequentiell erfolgt.

2.3 Prozeßkommunikation und Synchronisation

Damit ein paralleles Programm korrekt ablaufen kann, ist es notwendig, daß die einzelnen Prozesse des Programms zusammenarbeiten. Diese Zusammenarbeit erfordert einen Datenaustausch und eine Synchronisation der Prozesse. Um untereinander Daten auszutauschen, muß eine Möglichkeit zur Kommunikation zwischen Prozessoren gegeben sein. Dies kann über einen gemeinsamen Speicher mittels globaler Variablen realisiert sein, oder durch ein Kommunikationsnetzwerk geschehen. Da die Implementierungen in dieser Ausarbeitung auf einem Multiprozessorsystem mit gemeinsamen Speicher durchgeführt wurden, soll im folgenden nur diese Kommunikationsmöglichkeit weiter betrachtet werden. Jedoch sind die hier vorgestellten Konzepte durchaus auch für ein Mehrprozessorsystem mit Kommunikationsnetzwerk zutreffend.

Synchronisation mehrerer Prozesse kann man als Kommunikation verbunden mit Kontrollinformationen zur Koordination betrachten. Dabei unterscheidet man in einem Multiprozessorsystem zwei Arten von Synchronisation, die verschiedenen Zwecken dienen:

- Sicherstellung des exklusiven Zugriffs auf Betriebsmittel
- Sicherstellung einer erwünschten Ausführungsreihenfolge

Der exklusive Zugriff bewirkt, daß jeweils nur ein Prozeß auf gemeinsame Betriebsmittel (z.B. gemeinsame Variablen) zugreifen kann, und damit die anderen Prozesse vom Zugriff ausgeschlossen sind. In einem Programm ist ein solcher Zugriff innerhalb eines kritischen Abschnitts vorzunehmen.

Def.: Ein *kritischer Abschnitt* ist ein Anweisungsteil eines Prozesses, der exklusiv ausgeführt werden muß und nicht unterbrechbar ist.

Diese Form der Synchronisation wird als *gegenseitiger Ausschluß* (mutual exclusion) bezeichnet.
Eine erwünschte Ausführungsreihenfolge mehrerer Prozesse kann durch Bedingungen realisiert werden, so daß ein Prozeß seine Codeabarbeitung nur dann fortführen kann, wenn eine Bedingung erfüllt ist. Diese Synchronisationsart nennt man *Bedingungssynchronisation.*

Im folgenden werden einige Mechanismen, die die Zusammenarbeit von Prozessen ermöglichen, dargestellt, und die Synchronisation zwischen Prozessen näher betrachtet. Die Angaben der nächsten Abschnitte sind im wesentlichen [Qui,87] entnommen.

2.3.1 Aktives Warten

Eine einfach zu implementierende Methode zur Synchronisation ist die Benutzung einer unteilbaren test&set-Operation auf einer gemeinsamen Variablen. Prozesse, die ihren kritischen Abschnitt ausführen wollen, testen und setzen die Variable und gehen in ihren kritischen Abschnitt, falls die Variable nicht gesetzt war. Durch das Setzen der Variablen werden die anderen Prozesse an der Ausführung ihres kritischen Abschnitts gehindert. Prozesse, die in ihren kritischen Abschnitt gehen wollen, testen permanent die gemeinsame Variable solange bis sie zurückgesetzt ist. So realisiert man den gegenseitigen Ausschluß mit aktivem Warten.
Eine Bedingungssynchronisation erreicht man mittels aktivem Warten, indem man das Setzen der Variablen als Nicht-Erfüllung einer Bedingung interpretiert. Die Prozesse, die synchronisiert werden sollen, testen die Variable wiederholt und können solange nicht mit ihren Ausführungen fortfahren bis die Bedingung erfüllt ist. Ein großer Nachteil beim aktiven Warten besteht allerdings darin, daß durch die Warteschleifen sehr viel Prozessorzeit verschwendet wird, und damit die Ausführungszeit der Programme, die aktives Warten benutzen, drastisch verlängert werden kann.

2.3.2 Semaphore

Um den Nachteil des aktiven Wartens zu umgehen, stellte Dijkstra 1968 das Konzept der Semaphore vor. Die Idee war, daß man durch Suspendierung und Reaktivierung der Ausführung von Prozessen das aktive Warten vermeiden kann. Die Suspendierung wird durch einen Entzug des Prozessors vom Prozeß durch das Betriebssytem realisiert, so daß der Prozessor während der Blockade des Prozesses andere Prozesse bearbeiten kann. Ein Semaphor (deutsch: Zeichenträger) ist eine positive Integervariable, welche die Anzahl derjenigen Prozesse angibt, die in ihren kritischen Abschnitt eintreten wollen. Semaphore, die nur die Werte 0 und 1 annehmen, heißen binäre Semaphore. Ein Semaphor s kann nur durch zwei unteilbare Operationen P(s) und V(s) verändert werden. Die Wirkungsweise der beiden Operationen ist folgendermaßen umgangssprachlich erklärt:

P(s): "Passieren der Schranke zum kritischen Abschnitt"
Wenn $s > 0$, dann vermindere s um 1,
sonst suspendiere die Ausführung des Prozesses.

V(s) "Verlassen des kritischen Abschnitts und Freigabe für andere Prozesse"
Wenn ein Prozeß bezüglich s blockiert ist, dann lasse diesen P(s) ausführen, sonst erhöhe s um 1.

Der gegenseitige Ausschluß ist mit binären Semaphoren leicht zu gewährleisten, indem s mit 1 initialisiert wird, und jeder kritische Abschnitt der Prozesse von P(s) und V(s) eingeschlossen wird. Auch eine Bedingungssynchronisation kann mit Semaphoren erreicht werden. Dazu muß nach Initialisierung von s mit 0 jeder Prozeß nach Erfüllung einer Bedingung V(s) durchführen, und alle Prozesse, die auf die Erfüllung der Bedingung warten, führen P(s) aus. So können auch mehrere Prozesse bei Eintreffen eines Ereignisses aktiviert werden.
Semaphore werden durch Systemaufrufe an den Scheduler des Betriebssystems implementiert. Sie sind einfach und effizient zu handhaben. Allerdings muß gewährleistet sein, daß suspendierte Prozesse irgendwann wieder aktiviert werden, sonst besteht die Möglichkeit der Aussperrungen eines Prozesses oder sogar eines *deadlocks*.

2.3.3 Monitore

Monitore stellen im Vergleich zu den Semaphoren ein strukturierteres Mittel zur Synchronisation dar. In einem Monitor sind alle für die Synchronisation relevanten Daten und zugehörigen Operationen gebündelt. Damit erreicht man eine bessere Übersichtlichkeit und kann Fehler leichter lokalisieren. Ein Monitor besteht aus Variablen, Prozeduren und Initialisierungscode, worin die Variablen mit Anfangswerten belegt werden. Prozeduren im Monitor ähneln Prozeduren in Programmiersprachen, allerdings schließen sich alle Monitorprozeduren im Monitor gegenseitig aus und sind ununterbrechbar. Monitorvariablen können nur durch Aufrufe an die Monitorprozeduren verändert werden. So kann der gegenseitige Ausschluß durch Aufrufe an sich ausschließende Monitorprozeduren realisiert werden. Bedingungssynchronisation erreicht man durch die Implementierung von zwei Operationen *signal* und *wait*, die auf einer FIFO-Warteschlange bezüglich einer Monitorvariablen c arbeiten. Durch die Anweisung signal(c) wird, falls vorhanden, der erste Prozeß in der Warteschlange bezüglich c aktiviert. Wait(c) in einem Prozeß bewirkt eine Blockierung des Prozesses und Einreihung in die Warteschlange. So kann man eine Bedingungssynchronisation erreichen, indem man in Abhängigkeit von der Erfüllung einer Bedingung wait(c) bzw. signal(c) ausführt [Ben,85].

2.4 Die Architektur der CRAY Y-MP8/832

1988 wurde das Mehrprozessorsystem CRAY Y-MP von CRAY Research auf den Markt gebracht, welches das bisherige Spitzenmodell der erfolgreichen Vektorrechner-Serie CRAY-1, CRAY X-MP, CRAY Y-MP darstellt.

Die in der KFA Jülich installierte CRAY Y-MP8/832 stellt ein Mehrprozessorsystem mit gemeinsamen Speicher und acht identischen Vektor-Prozessoren dar, die mit einer Taktzeit von 6 ns arbeiten.
Jeder der acht Prozessoren beinhaltet mehrere unabhängige Funktionseinheiten, welche parallel und im Pipeline-Modus arbeiten können. Als Quelle und Ziel von Vektoroperationen fungieren in jeder CPU acht Vektorregister, die je 64 Elemente à 64 Bit aufnehmen können.
Die Prozessoren sind durch je vier Pfade mit dem Hauptspeicher verbunden, welcher eine Kapazität von 32 Megaworten hat und mit einer Speicherzykluszeit von 30 ns arbeitet. Die weiterentwickelte Hauptspeicherorganisation weist eine hierarchische Struktur auf, welche sich in vier Sektionen, 32 Subsektionen und 256 Bänken ausdrückt. Um Zugriffskonflikte zwischen den CPUs und dem Speicher aufzulösen, wurde ein automatischer Mechanismus in die Hardware implementiert, der bei auftretenden Konflikten Prioritäten setzt und den Zugriff auf Speicherbänke bis zu vier Takte lang verzögert.
Zusätzlich besitzt die Y-MP wie auch schon das Vorgängermodell X-MP einen sehr schnellen Erweiterungsspeicher SSD (solid state storage device), welcher mit einer Übertragungsgeschwindigkeit von 1000 Megabytes pro Sekunde bei der 8/832-Version arbeitet und eine Kapazität von 128 Megaworten besitzt.
Die Kommunikation zwischen den acht Vektorprozessoren geschieht über den gemeinsamen Speicher oder durch spezielle Kommunikationsregister. Dabei beträgt die Transferrate vom Hauptspeicher zu den Vektorregistern der Prozessoren vier Gigabytes pro Sekunde. Die Kommunikationsregister sind in neun Cluster zusammengefaßt, wobei jedes Cluster je acht Adreß- und Skalarregister sowie 32 binäre Semaphorregister enthält. Diese Kommunikations-Hardware stellt die Voraussetzung für Koordination und Datenaustausch zwischen den Prozessoren im Mehrprozessorbetrieb ([CRA,88a], [BEK,90]).

2.5 Multitasking bei der CRAY Y-MP

Neben dem Prinzip der Vektorverarbeitung, welches die Vektorstruktur der Prozessoren effizient nutzt, existiert seit der Entwicklung der CRAY X-MP auch das CRAY-Multitasking-Konzept. Multitasking bietet die Möglichkeit, ein Programm durch mehrere

Tasks auszuführen, welche dann parallel auf mehreren Prozessoren bearbeitet werden. Dabei wird der Begriff Task im Sinne von CRAY wie folgt definiert:

Def.: Eine Task ist eine eindeutig benannte Ausführungseinheit, der ein Prozeß zur Bearbeitung zugeordnet wird.

Die Zuteilung der Tasks auf die Prozessoren wird vom Scheduler des Betriebssystems vorgenommen.
Das Ziel des Multitasking-Konzepts ist es, den impliziten Parallelismus eines Programms auszunutzen, um eine kurze Ausführungszeit (wall clock time) zu erreichen [CRA,87]. Wenn mehrere Tasks parallel verarbeitet werden, kann die Reihenfolge ihrer Ausführung nicht vorhergesagt werden, da diese in einem Mehrprozessorsystem nichtdeterministisch erfolgt. Um trotzdem eine vom Benutzer erwünschte Reihenfolge der Taskbearbeitung zu erhalten, stellt Multitasking die Möglichkeit zur Koordination und Synchronisation zur Verfügung. Allerdings trägt dies zu einem Mehraufwand an Rechenzeit bei, der sich in dem Begriff *Overhead* widerspiegelt.
Zum heutigen Zeitpunkt drückt sich Multitasking in drei Erscheinungsformen aus, welche drei Entwicklungsphasen der CRAY-Parallelverarbeitungs-Software entstammen:

- Macrotasking
- Microtasking
- Autotasking

Microtasking und Macrotasking behandeln Parallelismus auf verschiedenen Ebenen im Programm, während Autotasking eine weiterentwickelte, automatische Form des Microtasking darstellt.

2.5.1 Macrotasking

Das mit der Entwicklung der CRAY X-MP eingeführte Macrotasking ermöglicht die parallele Ausführung eines Programms auf der Unterprogramm-Ebene.
Jedes vom Benutzer als parallel gekennzeichnete Unterprogramm entspricht einem einzelnen Prozeß, welcher dann von einer Task bearbeitet wird. In FORTRAN 77 wird Macrotasking durch CALL-Anweisungen im Quellcode realisiert, die Aufrufe an Bibliotheksroutinen darstellen. Dem Benutzer ergibt sich damit die Möglichkeit zur Erzeugung und Beendigung von Tasks und zur deren Synchronisation. So gibt es u.a. Routinen zur Sicherung des kritischen Abschnitts und zur Kontrolle von Ereignissen. Die Kommunikation zwischen Tasks im Macrotasking geschieht über gemeinsame Variablen im Hauptspeicher. Diese Kommunikationsart ist langsam gegenüber der Kommunikation

über Register, so daß ein größerer Overhead entsteht. Ein weiterer Nachteil des Macrotasking ist, daß die Tasks ein ganzes Unterprogramm bearbeiten und damit relativ groß sind. Dadurch kann bei Unterschieden in der Ausführungszeit der Tasks leicht Ineffizienz entstehen [HKN,89].

2.5.2 Microtasking

Um eine Parallelisierung auf einer niedrigeren Ebene zu ermöglichen, wurde Microtasking entwickelt, worin die Ebene der Schleifen und der Anweisungen behandelt wird. Die Benutzerschnittstelle zum Microtasking wurde durch Direktiven erreicht, die in den Fortrancode eingefügt werden. Diese Direktiven, welche immer mit dem Buchstaben C anfangen um Portabilität zu erhalten, werden von dem Präprozessor PREMULT in Aufrufe an Bibliotheksroutinen übersetzt. Auf diese Weise können vom Benutzer Kontrollstrukturen (z.B. DO-Schleifen) gekennzeichnet werden, die parallel ausführbare Prozesse enthalten. Microtasking-Direktiven bieten auch die Möglichkeit zur Anforderung und Rückgabe von CPUs, zur Kennzeichnung von Microtasking-fähigen Unterprogrammen und zur Sicherung kritischer Abschnitte. Am Ende einer jeden Kontrollstruktur wird eine Synchronisation vorgenommen, so daß der Benutzer weitgehend von dieser Aufgabe befreit ist. Allerdings ist eine intensive Betrachtung der Datenabhängigkeiten erforderlich, um unabhängige Programmsegmente zu entdecken.
Eine weiterführende Darstellung der wichtigsten Microtasking-Direktiven und ihrer Wirkungsweise kann man [Lie,88] entnehmen.
Die Kommunikation beim Microtasking wird über Register vorgenommen, so daß der Einfluß auf die Ausführungszeit eines Programms durch häufige Kommunikation nicht so groß wie beim Macrotasking ist. Wegen der i.a. kleinen Größe der Tasks im Microtasking und dem damit verbundenden geringen Zeitverbrauch, fällt ein Schwanken in der Prozessorzahl während der Ausführung eines Programms nicht sehr ins Gewicht. Dadurch wirkt sich Microtasking sehr effizient im Mehrbenutzerbetrieb aus. Die Microtasking-Programmierung gestaltet sich durch das Einfügen von Direktiven strukturiert und damit übersichtlich ([Nag,88],[CRA,87]).

2.5.3 Autotasking

Durch die Anwendung von Macro- oder Microtasking kann die Laufzeit eines Programms verringert werden. Allerdings stellt die Benutzung dieser beiden Multitasking-Komponenten hohe Anforderungen an den Anwender (s.o.). Die jüngste Multitasking-Entwicklung Autotasking bietet dem Benutzer die Möglichkeit, seine Programme zu beschleunigen, wobei die Entdeckung des Parallelismus und das Einfügen von Direktiven auf Schleifen- und Anweisungsebene weitgehend automatisch geschieht.

Autotasking operiert in drei Phasen, die bei der Anwendung hintereinander ausgeführt werden:

- Abhängigkeitsanalyse
- Übersetzungsphase
- Codegenerierungsphase

Die Abhängigkeitsanalyse wird vom Präprozessor fpp auf dem Fortran Quellcode durchgeführt. In dieser Phase werden parallelisierbare und vektorisierbare Programmsegmente entdeckt und mit Direktiven versehen. Dabei werden auch Informationen über Wertebereiche von Variablen in den Programmcode eingefügt. Außerdem können Umstellungen in der Ausführungsreihenfolge einzelner Programmsegmente vorgenommen werden, um Datenabhängigkeiten zu beseitigen.
Die Übersetzungsphase wird vom Präprozessor fmp auf dem veränderten Fortran-Code geleistet. Hier werden die vorhandenen Direktiven in Aufrufe an Autotasking-Bibliotheksroutinen übersetzt.
Die Codegenerierungsphase übernimmt der CRAY-Fortrancompiler CFT77, der optimierten Maschinencode erzeugt.
Der veränderte Quellcode nach der Abhängigkeitsanalyse gibt dem Anwender Informationen über parallelisierbare, bzw. vektorisierbare Programmkonstrukte. Durch nachträgliches Einfügen von Direktiven kann der Benutzer dem Autotasking zusätzliche Informationen geben.
Autotasking-Direktiven bilden eine Obermenge der Microtasking-Direktiven und liefern die Möglichkeit zum Hinweis auf Parallelismus, Schleifentransformationen und Datenabhängigkeiten. Die drei Phasen des Autotasking können beliebig oft unabhängig voneinander wiederholt werden.
Kommunikation und Synchronisation erfolgen im Autotasking fast ausschließlich direkt über Register, wobei die Synchronisation über Semaphorregister geschieht.

Autotasking stellt eine benutzerfreundliche Multitasking-Komponente dar, die auch durch die Entwicklung neuer, schneller Bibliotheksroutinen sehr leistungsfähig ist.
In einem Programm können gleichzeitig Macro-, Micro- und Autotasking verwendet werden. Allerdings schließen sich innerhalb eines Unterprogramms Micro- und Autotasking gegenseitig aus ([CRA,88b],[Nag,90]).

3.0 Synchrone und asynchrone parallele Algorithmen

Viele komplexe Probleme aus Wissenschaft und Technik erfordern heutzutage ein Höchstmaß an Rechenzeit, um sie zu lösen. Oft ist aber eine Problemlösung selbst bei paralleler Ausführung und mit der besten Rechnerleistung nicht in akzeptabler Zeit zu finden. Bei Parallelverarbeitung liegt eine Ursache für diese Tatsache beim auftretenden *Synchronisations-Overhead*, worauf in Abschnitt 3.2 eingegangen wird. Ein Konzept zur Verringerung des Synchronisations-Overhead ist deshalb die *asynchrone* Vorgehensweise, die in diesem Kapitel untersucht wird.
Die Angaben des Kapitels beruhen im wesentlichen auf [Kun,76], [T&B,89] und [Bau,78].

3.1 Grundlagen

In diesem Abschnitt werden einige Sachverhalte dargelegt und Begriffe erläutert, die als Grundlage der nächsten Abschnitte dienen.

Wie schon im letzten Kapitel dargestellt, besteht ein paralleles Programm aus mehreren Prozessen, zwischen denen Interaktionen ablaufen. Im Programmcode eines parallelen Programms existieren spezifizierte Stellen, an denen die einzelnen Prozesse miteinander kommunizieren können. Diese Stellen, welche *Interaktions-Punkte* genannt werden, teilen einen Prozeß in *Stufen* auf. Dabei findet während der Ausführung einer Stufe keinerlei Kommunikation zwischen den Prozessen statt, sondern lediglich nach dem Ende einer jeden Stufe. In einem Prozeß wechseln sich also Bearbeitung einer Stufe und Kommunikation gegenseitig ab.
In Parallel-Rechnern mit gemeinsamen Speicher geschieht diese Interaktion durch Aktualisieren von Variablen, auf die alle Prozesse zugreifen können. Ab dem Zeitpunkt wo diese gemeinsamen Variablen einen aktualisierten Wert erhalten haben, steht dieser Wert somit allen Prozessen zur Verfügung.

Eine wichtige Tatsache, die man beim Entwurf von parallelen Algorithmen und deren Aufteilung in Prozesse beachten muß, ist, daß man häufig die Zeit, die ein Prozeß für die Bearbeitung einer Stufe benötigt, nicht von vorneherein festlegen kann. Selbst bei zwei identischen Prozessen können die Ausführungszeiten von gleichen Stufen verschieden lang sein. Diese *Schwankungen in der Prozeßausführungszeit* entstehen im wesentlichen durch den Einfluß von zwei Faktoren. Zunächst hängt die Ausführungszeit einer Prozeß-Stufe von dem jeweiligen Mehrprozessorsystem ab, auf dem das parallele Programm ausgeführt wird. So beeinflussen Speicherkonflikte, Mehrbenutzerbetrieb und

Unterbrechungen durch das Betriebssystem die Bearbeitungszeiten. Andererseits können auch unterschiedliche Eingaben in einen Prozeß verschieden lange Ausführungszeiten bewirken. Als Beispiel soll hier die aufsteigende Sortierung eines schon geordneten und eines absteigend geordneten Feldes mit n Elementen durch einen Quicksort-Prozeß genannt werden. Im ersten Fall ist die Anzahl der benötigten Sortierschritte $O(n^2)$, wohingegen im zweiten Fall nur O(n log n) Schritte durchgeführt werden müssen.

Um die Leistungsfähigkeit eines parallelen Algorithmus im Vergleich zu dem entsprechenden sequentiellen Algorithmus auszudrücken, wird im folgenden der Begriff *Speedup* eingeführt.

Def.: Unter *Speedup* S eines Algorithmus versteht man den Quotienten aus sequentieller Ausführungszeit $T_{sequentiell}$ und paralleler Ausführungszeit $T_{parallel}$ des Algorithmus.

Der Speedup beschreibt also die Beschleunigung, die man mit der parallelen Ausführung eines Algorithmus erhalten kann.
In einem Mehrprozessorsystem mit k Prozessoren kann man einen Algorithmus im Idealfall k-mal schneller ausführen als in einem Einprozessorsystem.
Somit gilt in diesem Fall folgende Gleichung für den Speedup:

$$S(k) = \frac{T_{sequentiell}}{T_{parallel}} = k$$

Allerdings kann man einen Speedup der Größe k nur als theoretischen Wert betrachten, denn in der Praxis wirken mehrere Einflußfaktoren vermindernd auf den Speedup ein [Fox&&,88].
Im nächsten Abschnitt werden einige dieser Einflußgrößen auf den Speedup näher betrachtet.

3.2 Synchrone Algorithmen

Wenn in einem parallelen Algorithmus ein Prozeß existiert, der eine bestimmte Stufe erst dann bearbeiten kann, nachdem mindestens ein anderer Prozeß einen Teil seiner Berechnungen durchgeführt hat, so handelt es sich um einen *synchronen Algorithmus*.
In einem synchronen Algorithmus werden also Prozesse nach Abarbeitung von Stufen zeitlich koordiniert. Realisiert wird diese Synchronisation durch Ausführen von Synchronisationsmechanismen, die in Kapitel 2.3 beschrieben wurden.

Als Beispiel wird hier ein synchroner Algorithmus betrachtet, der die Berechnung $Z = (A \times B) + (C \times D \times E)$ durchführt. Dieser Algorithmus, welcher in Beispiel 1 in Pseudocode-Notation beschrieben ist, besteht aus zwei Prozessen P_1 und P_2, die mit Hilfe von Semaphoren synchronisiert werden.

```
   shared variable : X
   semaphor        : S
   S = 0
cobegin
   process P1 : X = A × B
                V(S)
   process P2 : Y = C × D × E
                P(S)
                Z = X + Y
coend
```

Beispiel 1: Synchroner Algorithmus

Die in diesem Beispiel benutzte cobegin-coend Anweisung markiert die Grenzen eines Programmabschnitts, der mehrere parallel ausführbare Prozesse enthält.
Prozeß P_1 berechnet in einer Stufe die gemeinsame Variable X und aktualisiert ihren Wert. Die anschließende Durchführung von V(S) signalisiert dem anderen Prozeß P_2, daß der Variablen X ein Wert zugewiesen wurde. P_2 besteht aus zwei Stufen, wobei in der ersten Stufe der Wert einer lokalen Variable Y ausgerechnet wird. Durch die anschließende Ausführung von P(S) wird erreicht, daß mit dem Beginn der zweiten Stufe von P_2 solange gewartet wird, bis P_1 die Variable X aktualisiert hat. Dies ist notwendig, da bei der Berechnung von Z in der zweiten Stufe von P_2 der Wert von X eingeht.

Die für den korrekten Ablauf eines synchronen Algorithmus erforderlichen Anweisungen für die Synchronisation, welche im folgenden mit Synchronisations-Direktiven bezeichnet werden, beeinflussen das Zeitverhalten des Algorithmus nachteilig. Insgesamt kann man den Zeitverbrauch eines Prozesses in einem synchronen Algorithmus in drei Komponenten unterteilen:

- Grundzeit für Berechnungen
- Ausführungszeit der Synchronisations-Direktiven
- Wartezeit

Unter Grundzeit für Berechnungen versteht man diejenige Zeit, die sich allein aus den reinen Berechnungszeiten für die Stufen eines Prozesses addiert. Die Ausführung von Synchronisations-Direktiven beansprucht zusätzliche Rechenzeit, denn u.a. sind Aufrufe

und Durchführung von Synchronisationsroutinen erforderlich. Unter Wartezeit eines Prozesses werden diejenigen Zeiträume zusammengefaßt, in denen der Prozeß blockiert ist, weil auf das Eintreffen eines Ereignisses gewartet werden muß. Dabei kann ein Ereignis entweder die Verfügbarkeit von Daten oder das Erreichen einer Stelle in der Berechnung eines anderen Prozesses sein. Schwankungen in der Prozeßausführungszeit bewirken außerdem, daß selbst bei gleicher Lastverteilung auf die einzelnen Prozesse fast immer Wartezeiten entstehen.
Die beiden letzten Zeitkomponenten werden unter dem Begriff *Synchronisations-Overhead* zusammengefaßt. Der Synchronisations-Overhead eines synchronen Algorithmus trägt im erheblichen Maße zur Verringerung des Speedup und damit zur Leistungsminderung bei. Dieser Sachverhalt wird im folgenden näher betrachtet.

Auf einem Mehrprozessorsystem mit k Prozessoren kann man einen theoretischen Speedup von k erreichen. Hier wird nun ein paralleler, synchroner Algorithmus mit k Prozessen angenommen, dessen gesamter Zeitverbrauch $T_{parallel}$ beträgt. Wenn nun t_i denjenigen Zeitanteil von $T_{parallel}$ angibt, in dem i Prozesse ($0 \leq i \leq k$) aktiv und k-i Prozesse blockiert sind, gilt:

$$T_{parallel} = \sum_{i=0}^{k} t_i$$

Obiger Algorithmus könnte auf einem Einprozessorsystem in einer Zeit $T_{sequentiell} = \sum_{i=0}^{k} i \times t_i - \omega$ ausgeführt werden. Hierbei gibt ω die Ausführungszeit für die Synchronisations-Direktiven an, welche bei sequentieller Ausführung wegfällt.
Für den Speedup S(k) gilt in diesem Falle:

$$S(k) = \frac{T_{sequentiell}}{T_{parallel}} = \frac{\sum_{i=0}^{k} i \times t_i - \omega}{\sum_{i=0}^{k} t_i} \leq k$$

Man erkennt also, daß durch Wartezeit und Ausführungszeit der Synchronisations-Direktiven der Speedup vermindert wird.

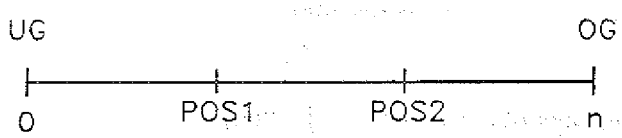
Um obige Nachteile zu umgehen, wurden *asynchrone Algorithmen* entwickelt, in denen Wartezeiten und Ausführungszeiten der Synchronisations-Direktiven weitgehend vermieden werden.

3.3 Asynchrone Algorithmen

Asynchrone parallele Algorithmen lassen sich allgemein durch folgende Merkmale charakterisieren:
Kommunikation findet nach jeder Stufe eines Prozesses durch Lesen und Schreiben von globalen Variablen statt, die allen Prozessen zugänglich sind. Danach führt jeder Prozeß seine Berechnungen, abhängig von den aktuellen Werten einiger globaler Variablen, aber unabhängig von allen anderen Prozessen, weiter fort, oder er wird beendet. Da die Prozesse unabhängig voneinander ihre Berechnungen durchführen, müssen zu keiner Zeit Prozesse auf andere Prozesse warten, wodurch keine Wartezeiten entstehen. In einem asynchronen Algorithmus werden die Prozesse nicht zeitlich koordiniert, d.h. es findet keine Synchronisation der Prozesse statt, so daß auch keine Synchronisations-Direktiven erforderlich sind.

Allerdings werden zur Garantie des gegenseitigen Ausschlusses beim Schreiben von globalen Variablen kritische Abschnitte benötigt, so daß in diesem Fall Wartezeiten anfallen, jedoch in einem sehr viel geringeren Maße als bei synchronen Algorithmen.

Im folgenden wird als Beispiel ein asynchroner Algorithmus mit zwei Prozessen beschrieben, der eine Nullstelle einer monoton wachsenden, stetigen Funktion findet, welche auf dem Intervall [0,n] definiert ist. Dabei wird die Existenz genau einer Nullstelle der Funktion im Intervall vorausgesetzt.
In dem Algorithmus gibt es zwei gemeinsame Variablen: UG und OG, die jeweils die Werte der aktuellen unteren und oberen Grenzen desjenigen Intervalls enthalten, welches noch zu untersuchen ist. In folgender Abbildung ist die Ausgangssituation des Algorithmus auf dem Intervall dargestellt.



Der Algorithmus geht so vor, daß die Prozesse P_i die Funktion an der Position i (POSi i= 1,2) auswerten und in Abhängigkeit davon die gemeinsamen Variablen UG und OG aktualisieren. Beispiel 2 stellt die Pseudocode-Notation des Algorithmus dar.

```
shared variables : UG,OG,X
     UG = 0
     OG = n
     X  = -1
     POS1 = n/3
     POS2 = 2*n/3
cobegin
   process P_i : while (X < 0) do
                 begin
                  if f(POSi) = 0 then X  = POSi
                  if f(POSi) < 0 then UG = POSi
                  if f(POSi) > 0 then OG = POSi
                  POSi = (OG-UG)/2
                 end
coend
```

Beispiel 2: Asynchroner Algorithmus

Auf die Verwendung von kritischen Abschnitten wurde in diesem Beispiel der Übersicht halber verzichtet.
Man erkennt, daß in einer Iteration der While-Schleife sich jeweils eine Stufe eines Prozesses P_i befindet. Im Beispiel wird auch deutlich, daß P_1 und P_2 für ihre Berechnungen von POS1 bzw. POS2 immer die zu diesem Zeitpunkt aktuellsten Werte für UG und OG benutzen, ohne aufeinander zu warten. Die beiden Prozesse arbeiten ganz unabhängig voneinander, und es wird keinerlei Synchronisation verwendet.

Während der Ausführung des obigen Algorithmus kann es allerdings vorkommen, daß beide Prozesse die Funktion an der selben Position auswerten, oder daß ein Prozeß an einer Stelle im Intervall auswertet, welche schon vom anderen Prozeß als jenseits der Grenzen liegend erkannt wurde. Es können also überflüssige Berechnungen ausgeführt werden, die durch Synchronisation vermeidbar wären.
Diese Tatsache stellt eine typische Eigenschaft von asynchronen Algorithmen dar, die die Leistungsfähigkeit negativ beeinflußt. In vielen Fällen übertrifft jedoch die Zeit, die durch Wegfall des Synchronisations-Overhead eingespart wird, diejenige Zeit, welche durch überflüssige Berechnungen verbraucht wird.

Vor allem bei iterativen asynchronen Algorithmen wird dies deutlich, so daß in der Klasse der iterativen Algorithmen gute Ergebnisse mit Asynchronität erzielt werden. Das nächste Kapitel beschäftigt sich deshalb mit iterativen parallelen Algorithmen.

3.4 Iterative parallele Algorithmen

Viele numerische Probleme werden in der Praxis iterativ gelöst. Iterative Methoden gehen so vor, daß schrittweise von mehreren Startwerten ausgehend eine Folge von Werten $(x(1),x(2), \dots)$ erzeugt wird, die gegen eine Lösung des Problems x_{opt} konvergiert. Ein Iterationsschritt beschreibt dabei den Übergang von $x(t-1)$ zu $x(t)$, $(t \in \mathbb{N})$.
Ein allgemeines Iterationsmodell stellt sich folgendermaßen dar:

$$x(t) = f(x(t-1), \dots, x(t-d)) \qquad (1 \leq d \leq t) \tag{3.1}$$

Hierbei gibt f die Iterationsfunktion an, die vorschreibt, wie der Iterationsschritt durchzuführen ist und x_{opt}, für das gilt $x_{opt} = f(x_{opt})$, ist ein Fixpunkt.
Beispielsweise lautet die Iterationsvorschrift beim Newton-Verfahren zur Bestimmung einer Nullstelle einer Funktion g:

$$x(t) = x(t-1) - \frac{g(x(t-1))}{g'(x(t-1))} \tag{3.2}$$

Parallelismus kann man in iterativen Algorithmen durch Aufteilung der Iterationsfunktion auf mehrere Prozesse erreichen. Ein Iterationsschritt wird dann jeweils von mehreren Prozessen gleichzeitig in einer Schleifeniteration durchgeführt. Beispielsweise kann diese Aufteilung komponentenweise geschehen, wenn x ein n-dimensionaler Vektor ist, $f: \mathbb{R}^n \mapsto \mathbb{R}^n$ und n Prozesse vorhanden sind. Dann stellt sich obiges Iterationsmodell mit $d = 1$ wie folgt dar:

$$x_i(t) = f_i(x_1(t-1), \dots, x_n(t-1)) \qquad (i = 1, \dots, n, f_i: \mathbb{R}^n \mapsto \mathbb{R}, f = (f_1, \dots, f_n)) \tag{3.3}$$

In diesem Fall aktualisiert der i-te Prozeß die i-te Komponente von x nach obiger Vorschrift. Dabei werden in jedem Iterationsschritt alle Ergebnisse von allen Prozessen der vorherigen Iteration benötigt.
Beispielsweise weist eine iterative Methode zur Lösung linearer Gleichungssysteme die folgende allgemeine Form auf:

$$x(t) = A \times x(t-1) + b \qquad (x, b \in \mathbb{R}^n, A \in \mathbb{R}^{n \times n}), \tag{3.4}$$

und deren komponentenweise Darstellung sieht folgendermaßen aus:

$$x_i(t) = \sum_{j=1}^{n} a_{ij} x_j(t-1) + b_i \qquad ((a_{ij}) = A). \tag{3.5}$$

3.4.1 Synchrone iterative Algorithmen

Wenn in einem iterativen parallelen Algorithmus am Ende eines jeden Iterationsschrittes eine Synchronisation aller Prozesse erfolgt, so handelt es sich um einen synchronen iterativen Algorithmus. Durch die Synchronisation wird erreicht, daß alle Teilergebnisse der einzelnen Prozesse aufgesammelt werden, bevor der nächste Iterationsschritt beginnen kann. Diese Vorgehensweise erzeugt die gleiche Iterationssequenz, wie beim entsprechenden sequentiellen iterativen Algorithmus.
Da alle Prozesse ihre Iteration beendet haben müssen, bevor irgendein Prozeß die nächste Iteration beginnen kann, wird die Zeit, die ein Iterationsschritt benötigt, durch den langsamsten Prozeß festgelegt. Beim Beispiel des Newton-Verfahrens wird dies deutlich, wenn man die Iterationsfunktion auf zwei Prozesse aufteilt und den Algorithmus wie in Beispiel 3 gestaltet.

```
    shared variable : Z
    semaphor : S
    S = 0
cobegin
    process P1 : for i=0 until cond do
                 begin
                  Y = g(X)
                  P(S)
                  X = X - Y/Z
                 end
    process P2 : for i=0 until cond do
                  begin
                   Z = g'(X)
                   V(S)
                  end
coend
```

Beispiel 3: Synchrones Newton-Verfahren

Hier führt P_1 die Berechnung von g(x) und x aus, während P_2 g'(x) berechnet. *Cond* ist in diesem Algorithmus eine boolesche Variable, die eine Bedingung für den Abbruch der Iteration darstellt.
In der Regel erfordert eine Auswertung der ersten Ableitung von g, etwa durch numerische Approximation, wesentlich mehr Schritte als eine Auswertung von g selbst. In Abb. 1 ist ein mögliches Zeitverhalten des Algorithmus dargestellt.

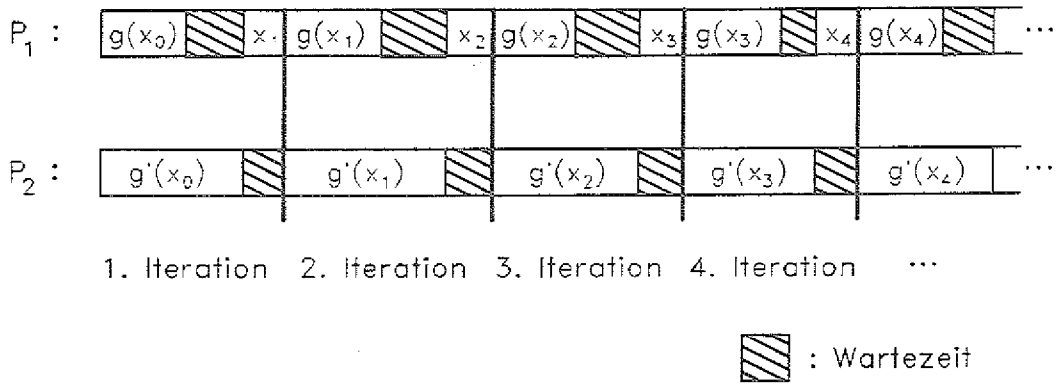


Abb. 1. Zeitverhalten beim synchronen Newton-Verfahren

In diesem Beispiel sind die Lasten also sehr unterschiedlich auf die beiden Prozesse verteilt, so daß P_1 und P_2 häufig blockiert sind. Synchrone iterative Algorithmen können somit nur dann leistungsfähig sein, wenn die Iterationsfunktion in gleichkomplexe Teile auf die Prozesse verteilt wird.
Insbesondere bei vielen Iterationsschritten wird der Einfluß des Synchronisations-Overhead auf das Zeitverhalten des Algorithmus sehr groß.
Außerdem treten bei synchronen iterativen Algorithmen häufig Speicherkonflikte auf, weil oft alle Prozesse in jedem Iterationsschritt zur gleichen Zeit auf die gleichen Variablen zugreifen. Hierdurch wird zusätzlicher Zeitverbrauch verursacht.

3.4.2 Asynchrone iterative Algorithmen

Um den Synchronisations-Overhead zu vermeiden, wird bei asynchronen iterativen Algorithmen auf die Synchronisation am Ende eines jeden Iterationsschrittes verzichtet, so daß alle Prozesse ihre Berechnungen zeitlich unabhängig voneinander, auf den ihnen gerade zur Verfügung stehenden Werten basierend, ausführen.

Baudet [Bau,78] führte ein allgemeines asynchrones Iterationsmodell ein, das auf dem Modell der *Chaotischen Relaxation* von Chazan und Miranker [C&M,69] beruht. Da dieses Modell die Vorgehensweise von asynchronen, iterativen Algorithmen darstellt, wird es an dieser Stelle vorgestellt und anschließend erläutert.
Demnach ist eine allgemeine asynchrone Iteration eine Folge von Vektoren aus $\mathbb{R}^n$ $(x(0),x(1), \ldots)$, die nach folgender Vorschrift gebildet wird:

$$x_i(t) = \begin{cases} x_i(t-1) & falls\ i \notin T_t \\ f_i(x_1(S_1(t)), \ldots, x_n(S_n(t))) & falls\ i \in T_t \end{cases} \tag{3.6}$$

Wobei $(T_t \mid t = 1,2, \ldots)$ eine unendliche Folge von Teilmengen von $\{1, \ldots, n\}$ und $((S_1(t), \ldots, S_n(t)) \mid t = 1, 2, \ldots)$ eine unendliche Folge über $(\mathbb{N} \cup \{0\})^n$ ist.
Zusätzlich müssen in diesem Modell folgende drei Bedingungen für alle i= 1,...,n erfüllt sein:

(a) $S_i(t) \leq t - 1 \quad , t = 1,2, \ldots$
(b) $\lim_{t \to \infty} S_i(t) = \infty$
(c) i erscheint unendlich oft in den Mengen der Folge $(T_t \mid t = 1,2, \ldots)$

Durch $t = 1,2, \ldots$ wird eine Folge von Zeitpunkten τ_t aufgezählt, zu denen Berechnungen von Komponenten von x stattfinden, und die nicht unbedingt zeitlich äquidistant zueinander liegen müssen. Hierbei bezeichnen die Elemente i einer Menge T_t diejenigen Komponenten x_i von x, die zum t-ten Zeitpunkt τ_t neu berechnet werden. Die Folge $(S_1(t), \ldots, S_n(t))$ gibt dabei an, von welchen Zeitpunkten die Werte der Komponenten von x stammen, die in die Berechnung von x_i eingehen.
Bedingung (a) sagt also aus, daß diese Werte nur aus Zeitpunkten stammen dürfen, die vor dem t-ten Zeitpunkt τ_t liegen.
Bedingung (b) verlangt, daß im Laufe der Berechnungszeit immer aktuellere Werte der Komponenten zur Berechnung hinzugezogen werden müssen. Dies bedeutet, daß zu alte Werte irgendwann nicht mehr benutzt werden dürfen.
In Bedingung (c) wird gefordert, daß jede Komponente unendlich oft aktualisiert wird. D.h. insbesondere, daß keine Komponente von den Berechnungen ausgeschlossen wird.

Ein paralleler Algorithmus, welcher auf diesem asynchronen Iterationsmodell beruht, geht so vor, daß ein Prozeß, der zu einem Zeitpunkt τ_t gerade keine Komponente bearbeitet, die Berechnung der Komponenten x_i für $i \in T_t$ beginnt. Der Vektor x stellt im Algorithmus eine Menge von gemeinsamen Variablen dar. Die Zeitpunkte τ_t und die Werte für $S_i(t)$ sind nicht von vorneherein bekannt und ergeben sich erst bei der Ausführung des Algorithmus. Jede Festlegung von vorneherein würde hier eine Synchronisation erfordern. Die Mengen T_t sind insofern durch einen Algorithmus festlegbar, als angeben wird, wie die gemeinsamen Variablen auf die Prozesse zu verteilen sind. Allerdings ist die zeitliche Reihenfolge der Mengen T_t nichtdeterministisch, weil die Dauer der Berechnungen variieren kann.

Zur Veranschaulichung ist in folgender Abbildung ein mögliches Zeitverhalten bei der Berechnung von drei Komponenten in einem Mehrprozessorsystem dargestellt:

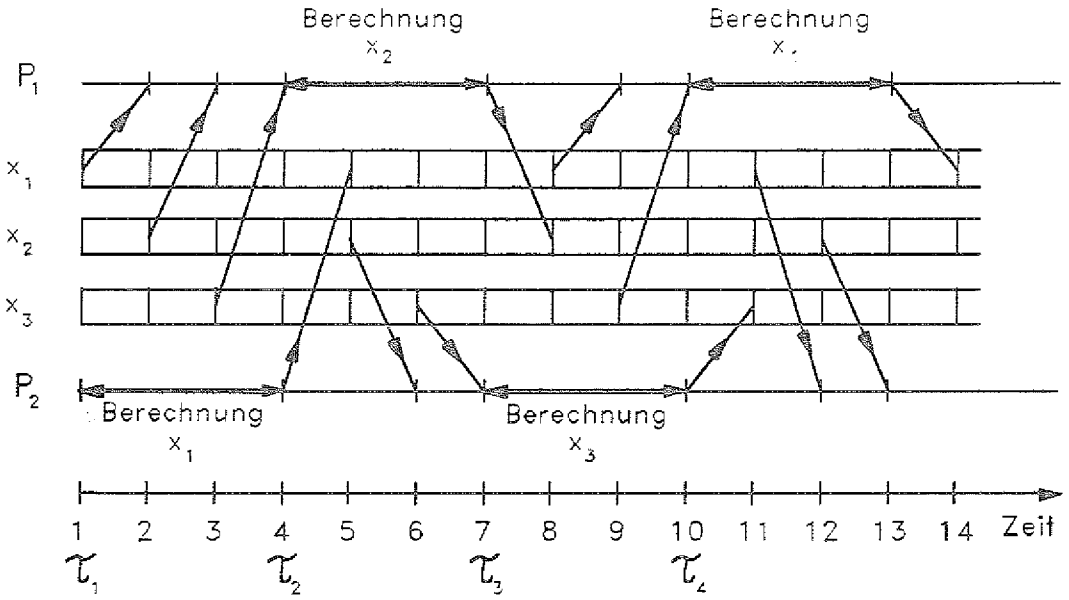


Abb. 2. Zeitliches Verhalten zweier Prozesse im Mehrprozessorsystem

Dargestellt sind die Speicherplätze der Komponenten x_1, x_2 und x_3, sowie die Aktionen Lesen, Berechnen und Schreiben einer Komponente der Prozesse P_1 und P_2 im zeitlichen Ablauf. Hier gilt beispielsweise $1 \in T_4$, weil zum vierten Zeitpunkt $\tau_4 = 10$ mit der Aktualisierung von x_1 begonnen wird. $S_1(4) = 1$, denn der Wert von x_1, der in die Berechnung eingeht, stammt vom Zeitpunkt $\tau_1 = 1$. Ebenso gilt: $S_2(4) = 2$ und $S_3(4) = 0$, da x_3 zum vierten Zeitpunkt noch keinmal aktualisiert wurde.

Die Differenz $D_i = t - S_i(t)$ kann man somit als *Kommunikations-Verzögerung* bezeichnen, denn sie gibt an, um wieviele Zeitpunkte die Berechnung des benutzten Wertes der Komponente x_i zurückliegt.

Das Verzichten auf Synchronisation hat den Effekt, daß die Prozesse in asynchronen Algorithmen mehr Iterationen in der gleichen Zeit durchführen können, weil sie nicht angewiesen sind, auf die neuesten Ergebnisse der anderen Prozesse zu warten. Außerdem bewirken eine ungleiche Lastverteilung und Schwankungen in der Prozeßausführungszeit bei asynchroner Vorgehensweise keine Wartezeiten.

Da bei asynchronen iterativen Algorithmen die Aktualisierung der einzelnen Komponenten zu verschieden Zeitpunkten und in variabler Reihenfolge erfolgen kann, wird allgemein eine andere Iterationssequenz erzeugt, als diejenige eines entsprechenden sequentiellen Algorithmus. In der Regel erzeugen sogar zwei Durchläufe des gleichen asynchronen Algorithmus zwei verschiedene Iterationsfolgen. Zudem treten wenig Speicherkonflikte auf, da der Zugriff der Prozesse auf die gemeinsamen Variablen meist zeitverschoben geschieht.
Eine weitere Tatsache ist, daß in asynchronen Algorithmen zwar mehr Iterationen pro Zeit als bei synchronen Algorithmen durchgeführt werden, die jedoch nicht unbedingt alle optimal zur Problemlösung beitragen, weil sie nicht auf den aktuellsten Werten beruhen.

Obige Sachverhalte tragen mit dazu bei, daß asynchrone Algorithmen in Bezug auf das Zeitverhalten, aber auch insbesondere im Hinblick auf Konvergenzeigenschaften schwer zu analysieren sind. Untersuchungen etwa in [T&B,89] zeigen, daß asynchrone Algorithmen, die auf Modell (3.6) beruhen, oft genau dann konvergieren, wenn die entsprechende sequentielle Iterationsmethode konvergiert, und dann auch zu einer schnelleren Problemlösung führen.
Im Rahmen dieser Arbeit soll jedoch nicht näher auf Konvergenz und Konvergenzgeschwindigkeit eingegangen werden, weil hier die Implementierung und das Zeitverhalten von Algorithmen auf einem Mehrprozessorsystem im Vordergrund stehen. Für genauere Betrachtungen des Konvergenzverhaltens von synchronen und asynchronen iterativen Algorithmen sei deshalb an dieser Stelle auf [Bau,78] oder [T&B,89] verwiesen.

3.4.3 Einfache asynchrone iterative Algorithmen

In den vorigen Abschnitten wurde die Iterationsfunktion auf mehrere Prozesse aufgeteilt, um eine parallele Ausführung zu erhalten. Falls aber die Aufteilung der Iterationsfunktion schwierig oder nicht möglich ist, kann man mehreren Prozessen die Berechnung der gesamten Iterationsfunktion zuweisen, um eine parallele Ausführung zu erhalten. Wenn dabei die einzelnen Prozesse immer mit den gerade zur Verfügung stehen Werten der gemeinsamen Variablen rechnen, entstehen *einfache asynchrone iterative Algorithmen*, die folgende allgemeine Form in Pseudocode-Notation aufweisen:

```
shared variable X
cobegin
  Prozeß P_i : while not cond do
                 begin
                  X = f(X)
                 end
coend
```

Durch obige Vorgehensweise werden Schwankungen in der Prozeßausführungszeit ausgenutzt, wie man in folgender Abbildung erkennen kann:

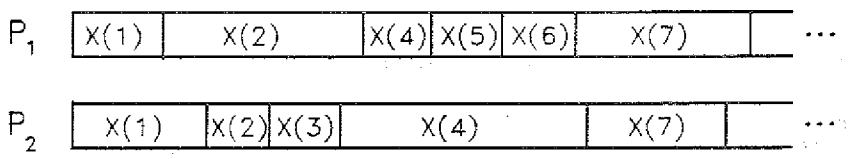


Abb. 3. Zeitverhalten bei einfachen asynchronen iterativen Algorithmen

Dargestellt ist ein mögliches Zeitverhalten bei der Berechnung der Iterationsvorschrift $x(t) = f(x(t-1))$ mit zwei Prozessen. Da die Berechnung der zweiten Iteration $x(2)$ von P_1 erst dann abgeschlossen ist, wenn P_2 schon $x(3)$ beendet hat, basiert die nächste Iteration von P_1 auf dem Wert von $x(3)$. Dies hat zur Folge, daß die vierte Iteration begonnen wird, d.h. ein Iterationsschritt wird übersprungen. Einfache asynchrone Algorithmen profitieren also davon, daß ein Prozeß Iterationsschritte überspringt, wenn ein anderer Prozeß schon spätere Iterationsschritte berechnet hat. Zwar werden viele Iterationsschritte von mehreren Prozessen bearbeitet, trotzdem kann aber eine Beschleunigung erreicht werden. In Abb. 3 ist etwa der siebte Iterationsschritt von P_2 schon nach fünf Auswertungsschritten bearbeitet.

Zur Bewertung von einfachen asynchronen Algorithmen ist zu sagen, daß ein Vorteil ihre generelle Anwendbarkeit ist. Die Aufteilung der Iterationsfunktion und ihre Zuteilung auf die Prozesse wird dem Anwender abgenommen. Da jedoch viele Iterationsschritte mehrfach berechnet werden, ist der Speedup, der mit dieser Vorgehensweise zu erreichen ist, nach oben sehr begrenzt, so daß keine bedeutende Beschleunigung der Ausführungszeit erwartet werden kann. Eine genaue Untersuchung von einfachen asynchronen Algorithmen und ihrer Leistungsfähigkeit findet sich in [BBK,80].

3.4.4 Gegenüberstellung

In folgender Tabelle sind zusammenfassend die Merkmale von synchronen und asynchronen Algorithmen gegenübergestellt.

Synchrone Algorithmen	Asynchrone Algorithmen
Ausführungszeit für Synchronisations-Direktiven	keine Synchronisation notwendig (evtl. kritische Abschnitte)
Wartezeit auf Ereignisse	kein Warten auf Ereignisse
gleiche Iterationsfolge wie beim entspr. sequentiellen Algorithmus	bei jeder Ausführung verschiedene Iterationsfolge möglich
gleiche Anzahl von Iterationsschritten bis zur Konvergenz wie beim sequentiellen Algorithmus	evtl. überflüssige und nicht optimal zur Problemlösung beitragende Iterationsschritte
Speicherkonflikte, da Zugriff auf gemeinsame Variablen zu gleicher Zeit	wenig Speicherkonflikte, da Zugriff auf gemeinsame Variablen zeitverschoben
Implementierung i.a. nicht einfach, weil Synchronisation beachtet werden muß	i.a. einfach zu implementieren, da keine Synchronisation
Konvergenz- und Zeitverhalten einfach zu analysieren	Konvergenz- und Zeitverhalten i.a. schwierig zu analysieren

Tab. 1. Gegenüberstellung: synchrone/asynchrone Algorithmen

Bei der Implementierung von synchronen und asynchronen Algorithmen im Rahmen dieser Arbeit wird gemessen, in welchem Ausmaß das Zusammenwirken obiger Merkmale das Zeitverhalten der Algorithmen beeinflußt.

3.4.5 Semi-asynchrone iterative Algorithmen

In den vorigen Abschnitten wurden Vor- und Nachteile von synchronen und asynchronen Algorithmen dargelegt. Semi-asynchrone iterative Algorithmen bilden einen Kompromiß beider Formen, um möglichst deren vorteilhafte Merkmale zu nutzen. Semi-asynchrone Algorithmen wurden in zwei Ansätzen entwickelt, die hier vorgestellt werden.

Der erste Ansatz geht so vor, daß Bedingung (a) aus dem asynchronen Iterationsmodell (3.6) durch eine Bedingung (a'): $t - B \le S_j(t) \le t - 1$ ersetzt wird, mit $1 \le B \le t$. Bedingung (a') sagt also aus, daß in die Berechnung einer Komponente x_j nur Werte von

Komponenten x_i eingehen, die frühestens von B Zeitpunkten vor dem t-ten Zeitpunkt stammen dürfen. Durch die Ersetzung der Bedingung (a) durch (a') wird es leichter möglich, Eigenschaften bezüglich der Konvergenz von vorneherein zu erkennen. In [T&B,89] werden diese Algorithmen als *partiell asynchrone Algorithmen* bezeichnet und deren Konvergenzverhalten analysiert im Vergleich zu den *total asynchronen Algorithmen*, die durch Modell (3.6) widergespiegelt werden.
Um Bedingung (a') in einem Algorithmus zu gewährleisten, muß eine Synchronisation der Prozesse vorgenommen werden, denn gegebenenfalls muß gewartet werden, bis einige Komponenten oft genug aktualisiert worden sind.
Der Wert von B gibt den Grad der Synchronisation des Algorithmus an. Wenn gilt $B = 1$, dann handelt es sich um einen synchronen Algorithmus, denn bei jedem Iterationsschritt gehen die Werte aus der vorigen Iteration ein. Für $B = t$ ist das asynchrone Iterationsmodell (3.6) erfüllt.
Der Synchronisations-Overhead, der durch semi-asynchrone Algorithmen verursacht wird, liegt also zwischen dem von synchronen und asynchronen Algorithmen.

Der zweite Ansatz für semi-asynchrone Algorithmen entstand aus der Tatsache heraus, daß in manchen Algorithmen die gemeinsamen Variablen nicht gleichzeitig aktualisiert werden können, weil sie voneinander abhängen, und damit nur in einer bestimmten Reihenfolge berechnet werden können. Diese Abhängigkeit tritt zum Beispiel beim *Gauß-Seidel-Verfahren* auf, welches im nächsten Kapitel vorgestellt wird. Allerdings besteht die Abhängigkeit oft nur insofern, als Komponenten der gemeinsamen Variablen, deren Indizes sich um einen Wert $d > 0$ unterscheiden, unabhängig voneinander sind. Dies kann man in einem semi-asynchronen Algorithmus ausnutzen, wo die Berechnung der Komponenten auf mehrere Prozesse verteilt wird. Jedoch wird durch Synchronisation sichergestellt, daß niemals Komponenten, deren Indizes sich um weniger als d unterscheiden, gleichzeitig berechnet werden. Durch diese Vorgehensweise erreicht man also trotz der Abhängigkeiten zwischen den gemeinsamen Variablen eine parallele Ausführbarkeit. Dabei ist der zu erwartende Synchronisations-Overhead geringer als bei synchronen Algorithmen, weil voneinander unabhängige Variablen asynchron berechnet werden.

Die Hauptmerkmale von semi-asynchronen iterativen Algorithmen lassen sich wie folgt zusammenfassen:

- Synchronisation in geringem Maße, so daß kein allzu großer Synchronisations-Overhead anfällt
- Erzwingung von erwünschten Eigenschaften durch Synchronisation

4.0 Iterative Verfahren zur Lösung von Gleichungssystemen

Gleichungssysteme ergeben sich aus Problemen der Wissenschaft und Technik. Beispielsweise treten sie bei der Diskretisierung von Randwertproblemen auf, wobei Differentialgleichungen in Gleichungssysteme überführt werden, welche man dann mit bekannten Methoden aus der Numerik lösen kann. Einige dieser Methoden zur Lösung linearer wie auch nichtlinearer Gleichungssysteme werden im folgenden vorgestellt.

4.1 Verfahren zur Lösung linearer Gleichungssysteme

Ist $A = (a_{ij})$ eine reelle $n \times n$ Matrix und b ein n-dimensionaler reeller Vektor, so stellt sich ein lineares Gleichungssystem folgendermaßen dar:

$$Ax = b \tag{4.1}$$

Ein n-elementige Vektor x, für den Gleichung (4.1) erfüllt ist, heißt eine Lösung des linearen Gleichungssystems. Der Lösungsvektor x existiert und ist eindeutig, genau dann, wenn A eine nichtsinguläre Matrix ist. Explizit läßt sich x dann mit $x = A^{-1}b$ darstellen, wobei A^{-1} die zu A inverse Matrix ist. Im folgenden sei die Regularität der Matrix A immer vorausgesetzt.
Die Berechnung der Lösung x mittels eines direkten Verfahrens, wie etwa der Gauß-Elimination oder dem Cholesky-Verfahren, erfordert in der Regel zwar weniger Rechenaufwand als das Finden der Lösung x durch iterative Methoden. Jedoch bei großen und schwach besetzten Matrizen sind die iterativen Methoden den direkten Methoden oft überlegen, sofern sie überhaupt konvergieren.

Wie schon in Abschnitt 3.4 dargelegt, werden die Komponenten des Vektors x nach einer Iterationsvorschrift aktualisiert. Diese Iterationsvorschrift erhält man bei linearen Gleichungssystemen aus der komponentenweisen Darstellung von (4.1):

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \qquad (1 \le i \le n) \tag{4.2}$$

durch Auflösen nach x_i, für alle $1 \le i \le n$, falls $a_{ii} \neq 0$:

$$x_i = -\frac{1}{a_{ii}} \left[\sum_{j \neq i} a_{ij} x_j - b_i \right] \tag{4.3}$$

Auf dieser Grundlage wurden verschiedene Iterationsverfahren entwickelt, die an dieser Stelle genauer betrachtet werden.

4.1.1 Jacobi-Verfahren

Die Iterationsvorschrift des Jacobi-Verfahrens für den Schritt t lautet wie folgt:

$$x_i(t) = -\frac{1}{a_{ii}}\left[\sum_{j \neq i} a_{ij}\, x_j(t-1) - b_i\right] \tag{4.4}$$

Beim Jacobi-Verfahren werden in jedem Iterationsschritt die Werte aller Komponenten der vorigen Iteration zur Berechnung einer Komponente x_i benötigt, weshalb dieses Verfahren auch als *Gesamtschrittverfahren* bezeichnet wird. Ein Nachteil des Jacobi-Verfahrens ist, daß der Vektor x immer zweifach gespeichert werden muß, da jeweils die Werte der aktuellen und der vorigen Iteration von x benötigt werden.

4.1.2 Gauß-Seidel-Verfahren

Um eine schnellere Konvergenz als beim Jacobi-Verfahren zu erzielen, nutzt das Gauß-Seidel-Verfahren die Tatsache aus, daß zum Zeitpunkt der Berechnung einer Komponente $x_i(t)$ für $j < i$ schon die Werte von $x_j(t)$ der aktuellen Iteration vorhanden sind. Die Iterationsfunktion für das Gauß-Seidel Verfahren ergibt sich damit zu:

$$x_i(t) = -\frac{1}{a_{ii}}\left[\sum_{j=1}^{i-1} a_{ij}\, x_j(t) + \sum_{j=i+1}^{n} a_{ij}\, x_j(t-1) - b_i\right] \tag{4.5}$$

Bei dieser Vorgehensweise benötigt man nur eine Speicherung des Vektors x, in der immer die neuesten Werte der Komponenten enthalten sind. Allerdings erfordert dieses Verfahren eine streng sequentielle Abarbeitungsreihenfolge, weshalb es *Einzelschrittverfahren* genannt wird.
Die notwendige sequentielle Bearbeitungsreihenfolge beim Gauß-Seidel-Verfahren verbietet deshalb eine parallele Berechnung der Komponenten, die hingegen beim Jacobi-Verfahren leicht realisierbar ist, da die Komponenten jeder Iteration in beliebiger Reihenfolge berechnet werden können.

4.1.3 Relaxationsverfahren

Eine schnellere Konvergenz als in den beiden vorigen Verfahren wird mit Relaxationsverfahren erreicht, die durch Einführung eines Relaxationsparameters ω aus dem Jacobi-Verfahren oder dem Gauß-Seidel-Verfahren entstehen. In diesen Verfahren wird jede Komponente $x_i(t)$ aus einem gewichtetem Mittel der Werte der Komponenten aus der vorigen und der aktuellen Iteration wie folgt gebildet:

$$x_i(t) = (1-\omega)\, x_i(t-1) + \omega\, \tilde{x}_i(t)$$

Hierbei wird $\tilde{x}_i(t)$ nach der Iterationsvorschrift des Jacobi-Verfahrens (4.4) oder des Gauß-Seidel-Verfahrens (4.5) berechnet.
Eingesetzt ergibt sich folgende Berechnungsvorschrift für die **Verallgemeinerung des Jacobi-Verfahrens**:

$$x_i(t) = (1-\omega)\, x_i(t-1) - \frac{\omega}{a_{ii}} \left[\sum_{j \neq i} a_{ij}\, x_j(t-1) - b_i \right] \tag{4.6}$$

Die **Verallgemeinerung des Gauß-Seidel-Verfahrens**, welche auch **SOR-Methode** (*successive overrelaxation method*) genannt wird, hat folgende Berechnungsvorschrift:

$$x_i(t) = (1-\omega)\, x_i(t-1) - \frac{\omega}{a_{ii}} \left[\sum_{j=1}^{i-1} a_{ij}\, x_j(t) + \sum_{j=i+1}^{n} a_{ij}\, x_j(t-1) - b_i \right] \tag{4.7}$$

Für $\omega = 1$ ergeben sich die ursprünglichen Verfahren [Var,62].

4.2 Konvergenz der Verfahren

Die oben betrachteten Verfahren zur Lösung von linearen Gleichungssystemen konvergieren nur gegen die Lösung x, wenn die Matrix A gewissen Bedingungen genügt. Zur Veranschaulichung wird die Matrix A in drei Teilmatrizen aufgeteilt, so daß $A = D - L - U$, wobei D die Diagonalelemente von A enthält, sowie $-L$ und $-U$ jeweils diejenigen Elemente von A besitzen, die unter bzw. über der Diagonalen liegen. Mit dieser Aufteilung kann z.B. die Berechnungsvorschrift des Jacobi-Verfahrens nun in Matrix-Notation wie folgt geschrieben werden:

$$x(t) = D^{-1}(L+U)\, x(t-1) + D^{-1} b$$

Diese Schreibweise kann für alle oben vorgestellten Verfahren einführt werden, so daß man eine allgemeine Form $x(t) = Gx(t-1) + g$ erhält, wobei die Matrix G und der Vektor g für jedes Verfahren verschieden sind.
Beispielsweise ist $G = (D - \omega L)^{-1}[(1-\omega)\, D + \omega U]$ für die SOR-Methode. Die Matrix G wird auch *Iterationsmatrix* genannt.
Es läßt sich nun folgender Satz über die Konvergenz der hier betrachteten Verfahren formulieren:

Satz 4.1: Die Methoden (4.4) bis (4.7) konvergieren genau dann gegen die Lösung x, wenn für die jeweilige Iterationsmatrix G gilt: $\rho(G) < 1$.

Der Beweis des Satzes ist in [Var,62] nachzulesen. $\rho(G)$ bezeichnet den *Spektralradius* der Matrix G, der gleich dem Betrag des betragsgrößten Eigenwerts der Iterationsmatrix ist. Es gilt allgemein, daß ein Iterationsverfahren um so schneller gegen die Lösung konvergiert, je kleiner $\rho(G)$ ist.
Für den Relaxationsparameter ω kann man aus Satz 4.1 für die Relaxationsverfahren schließen, daß sie nur für Werte $0 < \omega < 2$ konvergieren.

4.3 Matrizen mit Property A

Da von den oben vorgestellten Verfahren mit der SOR-Methode die schnellste Konvergenz erzielt wird, soll im folgenden auf dieses Verfahren näher eingegangen werden.
Wie weiter oben schon erwähnt, ist es notwendig, die Komponenten beim Gauß-Seidel-Verfahren, wie auch bei der SOR-Methode, in sequentieller Reihenfolge zu bearbeiten. Das verbietet eine parallele Ausführung dieser Methoden. Um trotzdem die gute Konvergenzgeschwindigkeit der SOR-Methode auch bei paralleler Ausführung zu nutzen, kann man Matrizen betrachten, die die sogenannte *property A* besitzen, welche von Young [You,71] eingeführt wurde.

Def:. Eine Matrix A der Ordnung n besitzt *property A*, falls zwei disjunkte Teilmengen S_1 und S_2 von $\{1, \ldots, n\}$ existieren mit $S_1 \cup S_2 = \{1, \ldots, n\}$, so daß für beliebige $i, j \in \{1, \ldots, n\}$ mit $i \neq j$ und entweder $a_{ij} \neq 0$ oder $a_{ji} \neq 0$ gilt:
$i \in S_1$ und $j \in S_2$ oder $i \in S_2$ und $j \in S_1$.

Beispielsweise besitzt untenstehende Matrix property A, denn die Indizes lassen sich auf zwei disjunkte Mengen verteilen:

$$\begin{pmatrix} 4 & -1 & 0 & 1 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 1 & 0 & -1 & 4 \end{pmatrix} \qquad \{1,3\} \in S_1 \,,\, \{2,4\} \in S_2$$

Wendet man die SOR-Methode auf Gleichungssysteme mit Matrizen an, die property A besitzen, so verwenden die Berechnungsvorschriften für die Komponenten, deren Indizes in der einen Menge sind, nur diejenigen Komponenten, deren Indizes in der anderen Menge sind. Komponenten, deren Indizes in einer Menge sind, können also un-

abhängig voneinander und dadurch in beliebiger Reihenfolge berechnet werden. Eine parallele Berechnung ist damit möglich.

Für Matrizen mit property A gibt es einen optimalen Relaxationsparameter ω_{opt}, für den die SOR-Methode am schnellsten konvergiert, d.h. für den $\rho(G) \geq 0$ den kleinsten Wert annimmt. Eine weitere vorteilhafte Eigenschaft dieser Matrizen mit property A ist, daß, falls die Eigenwerte von $D^{-1}(L + U)$ reell sind, dieser optimale Parameter explizit angegeben werden kann:

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho^2(D^{-1}(L + U))}}$$

Beispielsweise entstehen bei der Diskretisierung von elliptischen Randwertproblemen Gleichungssysteme mit Matrizen, die property A besitzen.

4.4 Das Newton-Verfahren zur Lösung nichtlinearer Gleichungssysteme

Nichtlineare Gleichungssysteme entstehen u.a. bei der numerischen Lösung von nichtlinearen Randwertproblemen durch Differenzenverfahren oder mit der Methode der finiten Elemente.
Im Gegensatz zu den linearen Gleichungssystemen gibt es kaum direkte Verfahren zur Lösung von nichtlinearen Gleichungssystemen, so daß die iterativen Methoden hier eine größere Bedeutung einnehmen. Allerdings bergen die iterativen Methoden die Schwierigkeit, daß die Konvergenz der Verfahren meist nur in einer bestimmten Umgebung einer Lösung gewährleistet ist. Da außerdem bei nichtlinearen Gleichungssystemen in der Regel mehrere Lösungen existieren, besteht die Gefahr, daß das Verfahren gegen eine andere Lösung als die gewünschte konvergiert. Dadurch stellt sich zusätzlich die Aufgabe, eine gute Ausgangsnäherung zu finden, bevor mit der Lösung des Gleichungssystems begonnen werden kann.

Das Problem ist die Bestimmung einer Nullstelle $x^* \in \mathbb{R}^n$ einer nichtlinearen Funktion $f = (f_1, \ldots, f_n) : B \subset \mathbb{R}^n \to \mathbb{R}^n$. Dies bedeutet, daß folgendes nichtlineare System von Gleichungen zu lösen ist:

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, \ldots, x_n) &= 0 \end{aligned} \tag{4.8}$$

Für den eindimensionalen Fall $f: B \subset \mathbb{R} \to \mathbb{R}$, mit f differenzierbar, lautet die Berechnungsvorschrift des Newton-Verfahrens wie folgt:

$$x(t) = x(t-1) - \frac{f(x(t-1))}{f'(x(t-1))} \tag{4.9}$$

Übertragen auf den mehrdimensionalen Fall ergibt sich folgende Berechnungsvorschrift für das allgemeine Newton-Verfahren zur Lösung von nichtlinearen Gleichungssystemen, wobei alle $f_i(x(t-1))$ differenzierbar sein müssen:

$$x(t) = x(t-1) - [Df(x(t-1))]^{-1} f(x(t-1)) \tag{4.10}$$

Hierbei gibt $Df(x)$ die *Funktionalmatrix* zu f an, die auch als *Jacobi-Matrix* bezeichnet wird und wie folgt gebildet wird:

$$Df(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

Für die Anwendung des allgemeinen Newton-Verfahrens muß die Regularität der Jacobi-Matrix vorausgesetzt werden.

Geometrisch kann man einen Iterationsschritt des Newton-Verfahrens von $x(t-1)$ nach $x(t)$ interpretieren, indem jede Funktion $f_i: \mathbb{R}^n \to \mathbb{R}$ durch eine Hyperebene $\nabla f_i(x(t-1))(x - x(t-1)) + f_i(x(t-1))$ approximiert wird, die f_i am Punkt $x(t-1)$ tangiert.

$\nabla f_i(x) = \left(\frac{\partial f_i}{\partial x_1}, \ldots, \frac{\partial f_i}{\partial x_n} \right)$ gibt hierbei den *Gradienten* einer Funktion f_i an einer Stelle x an.

$x(t)$ ergibt sich dann als Schnittpunkt im $\mathbb{R}^{n+1}$ der n Hyperebenen für f_i, $i = 1, \ldots, n$, mit der Hyperebene, die durch $x_{n+1} = 0$ festgelegt ist.

Für die praktische Anwendung von (4.10) ist es erforderlich, zu jeder Näherungslösung $x(t)$ die Funktionalmatrix zu invertieren. Um den damit verbundenen hohen Rechenaufwand zu vermeiden, multipliziert man beide Seiten der Gleichung (4.10) mit $Df(x(t-1))$ und erhält folgendes Gleichungssystem:

$$Df(x(t-1))\, x(t) = Df(x(t-1))\, x(t-1) - f(x(t-1)) \tag{4.11}$$

Dieses lineare Gleichungssystem kann dann mittels einer iterativen Methode (siehe Kap. 4.1) oder direkt gelöst werden [Sto,76].

4.4.1 Konvergenz des allgemeinen Newton-Verfahrens

Bei linearen Systemen von Gleichungen mit nichtsingulären Matrizen A ist die Konvergenz der Iterationsverfahren gegen die eindeutige Lösung x mit der Bedingung, daß für die Iterationsmatrix G $\rho(G) < 1$ gilt, gewährleistet. Dabei wird die Lösung für beliebige Ausgangsnäherungen $x(0) \in \mathbb{R}^n$ erreicht, so daß *globale Konvergenz* vorliegt.
Im Gegensatz dazu kann die Konvergenz des Newton-Verfahrens nur in einer geeigneten Umgebung B_0 einer Lösung x^* gewährleistet werden, die u.U. sehr klein sein kann. Es handelt sich hier also um *lokale Konvergenz*.
Ähnlich zu den Verfahren zur Lösung linearer Gleichungssysteme weisen auch iterative Verfahren zur Lösung nichtlinearer Gleichungssysteme eine allgemeine Form $x = G(x)$ auf. Speziell beim Newton-Verfahren ergibt sich $G(x)$ zu $G(x) = x - [Df(x)]^{-1} f(x)$.
Von Eigenschaften der Funktionalmatrix DG zu G kann man nun die lokale Konvergenz des Newton-Verfahrens ableiten, wie folgender Satz aussagt.

Satz 4.2: Wenn ein x^* existiert, mit $x^* = G(x^*)$, und $\rho(DG(x^*)) < 1$, dann existiert eine Umgebung B_0 von x^*, $B_0 \subset B$, so daß für alle Ausgangsnäherungen $x(0) \in B_0$ das Iterationsverfahren $x(t) = G(x(t-1))$ gegen x^* konvergiert.

Der Beweis dieses Satzes würde an dieser Stelle zu weit führen und ist in [Ort,70] nachzulesen.
Der für Verfahren der Form $x = G(x)$ allgemein gültige Satz 4.2 erfährt für das Newton-Verfahren die Vereinfachung, daß die Voraussetzung $\rho(DG(x^*)) < 1$ stets erfüllt ist. Denn $DG(x^*) = I - [Df(x^*)]^{-1} Df(x^*) = I - I = 0$, und damit ist auch der Spektralradius von $DG(x^*)$ gleich null.

Es bleibt also in der praktischen Anwendung des Newton-Verfahrens, auch wenn man die Existenz einer Nullstelle x^* von f voraussetzen kann, die Schwierigkeit, eine Umgebung B_0 von x^* zu finden, für die die Konvergenz gewährleistet ist.

4.4.2 Das Einschritt-Newton-SOR-Verfahren

Alle Anforderugen an die Funktion f und an die Jacobi-Matrix zu f im vorigen Abschnitt sollen auch für das nun betrachtete Verfahren gelten.
Wie in Kapitel 4.4 erwähnt, besteht durch Multiplizieren der Berechnungsvorschrift für das Newton-Verfahren (4.10) mit der Jacobi-Matrix auf beiden Seiten die Möglichkeit, ein lineares Gleichungssystem innerhalb einer jeden Newton-Iteration zu lösen. Dieses Gleichungssystem soll im folgenden mit der SOR-Methode gelöst werden. Dabei gibt es zwei Ansätze für die Behandlung des Gleichungssystems. Entweder führt man soviele

SOR-Iterationsschritte durch, bis eine gewünschte Genauigkeit der Lösung des linearen Systems vorliegt, oder man strebt eine Approximation der Lösung des linearen Gleichungssystems an, indem lediglich eine vorher festgelegte Anzahl an SOR-Iterationsschritten ausgeführt wird. Letzteres bietet sich wegen des damit verbunderen geringeren Rechenaufwandes an, insbesondere die einmalige SOR-Iteration innerhalb einer Newton-Iteration, so daß aus der Kombination beider Verfahren das **Einschritt-Newton-SOR-Verfahren** zur Lösung von nichtlinearen Gleichungssystemen entsteht.
Die Berechnungsvorschrift für das Einschritt-Newton SOR-Verfahren läßt sich erhalten, indem man für das Gleichungssystem (4.11) die Berechnungsvorschrift für einen Schritt der SOR-Methode anwendet. Wenn mit Df_{ij} das Element in der i-ten Zeile und der j-ten Spalte der Jacobi-Matrix $Df(x(t-1))$ bezeichnet wird und mit f_i die i-te Komponente von $f(x(t-1))$, so erhält man die komponentenweise Darstellung

$$x_i(t) = (1-\omega)\,x_i(t-1) - \frac{\omega}{Df_{ii}}\left[\sum_{j=1}^{i-1} Df_{ij}\,x_j(t) + \sum_{j=i+1}^{n} Df_{ij}\,x_j(t-1) - \sum_{j=1}^{n} Df_{ij}\,x_j(t-1) + f_i\right],$$

vereinfacht also:

$$x_i(t) = x_i(t-1) - \frac{\omega}{Df_{ii}}\left[\sum_{j=1}^{i-1} Df_{ij}\,(x_j(t) - x_j(t-1)) + f_i\right]. \tag{4.12}$$

4.4.3 Konvergenz des Einschritt-Newton-SOR-Verfahrens

Zerlegt man die Funktionalmatrix zu f auf gleiche Weise wie die Matrix A in Abschnitt (4.2), so daß gilt: $Df(x) = D(x) - L(x) - U(x)$, so kann man auch für dieses Verfahren eine Iterationsmatrix $G(x)$ bilden. Sie ergibt sich zu

$$G(x) = [D(x) - \omega L(x)]^{-1}[(1-\omega)D(x) + \omega U(x)].$$

Folgender Satz sagt aus, daß man aus dieser Iterationsmatrix die lokale Konvergenz des Einschritt-Newton-SOR-Verfahrens ableiten kann.

Satz 4.3: Wenn ein x^* existiert, mit $x^* = G(x^*)$, wobei $D(x^*)$ nichtsingulär sein muß, und $\rho(G(x^*)) < 1$, dann existiert eine Umgebung B_0 von x^*, $B_0 \subset B$, so daß für alle Ausgangsnäherungen $x(0) \in B_0$ das Einschritt-Newton-SOR-Verfahren gegen x^* konvergiert.

Zum Beweis sei wiederum auf [Ort,70] verwiesen.

Wie auch bei dem allgemeinen Newton-Verfahren kann man also auch beim Einschritt-Newton-SOR-Verfahren nur lokale Kovergenz erhalten, wobei sich die damit verbundenen Schwierigkeiten für die praktische Anwendung auch hier ergeben.

Nachdem einige Methoden zur Lösung von linearen und nichtlinearen Gleichungssystemen an dieser Stelle beschrieben wurden, werden im nächsten Kapitel deren Implementierungen vorgestellt.

5.0 Implementierung der Algorithmen

Ziel dieser Arbeit ist die Implementierung von Algorithmen zur Lösung von Gleichungssystemen auf CRAY-Mehrprozessorsystemen. Das Hauptaugenmerk ist dabei auf die Untersuchung von synchronen und asynchronen parallelen Versionen dieser Algorithmen gerichtet.
Die Programmierungen wurden auf den CRAY-Rechnern X-MP/416 und Y-MP8/832 in der Programmiersprache FORTRAN 77 vorgenommen und mit dem autovektorisierenden Compiler CFT77 4.0.1 übersetzt. Die Parallelisierung der Programme wurde mit der CRAY-Multitasking-Komponente Autotasking erreicht, wobei der Compiler CFT77 nach Durchlauf der Programmcodes durch zwei Präprozessoren als letzte Komponente aufgerufen wird (siehe Kap. 2.5.3).
Feinheiten der Programmierung resultierten oft daraus, daß eine Anpassung an die Struktur der hochleistungsfähigen CRAY-Vektorrechner angestrebt wurde, und insbesondere die parallelen Programme sind speziell für CRAY-Rechner entwickelt, so daß diese nicht unmittelbar auf andere Rechnerarchitekturen übertragbar sind.
Um die parallelen Programme zu bewerten und zu vergleichen, wurden Zeitmessungen zunächst in einer dedizierten Umgebung vorgenommen. Im nächsten Kapitel werden aber auch Untersuchungen im Multiprogramming-Betrieb durchgeführt.
Es soll jedoch schon an dieser Stelle erwähnt werden, daß die asynchronen Programme in einer dedizierten Umgebung nicht ihre spezielle Leistungsfähigkeit entwickeln können.

5.1 Globale Programmstruktur

Alle im folgenden beschriebenen Programme weisen eine gemeinsame Grobstruktur auf, die der Übersicht halber zunächst vorgestellt werden soll und in Abb. 4 schematisch dargestellt ist.
Den eigentlichen Programmkern bildet dabei jeweils der in der Abbildung fett umrandete Teil, der im wesentlichen aus der als UNTIL-Schleife realisierten sogenannten Iterationsschleife besteht, denn ein Schleifendurchlauf entspricht jeweils einem Iterationsschritt.
Alle durchgeführten Zeitmessungen beziehen sich daher nur auf den Programmkern, da die übrigen Programmteile nur Initialisierung bzw. Ein/Ausgabe enthalten. Die Zeitmessungen wurden jeweils durch Aufrufe der CRAY-Funktion IRTC vorgenommen, welche den Inhalt des Echtzeitregisters wiedergibt, so daß durch Differenzbildung der Werte zweier Aufrufe von IRTC die inzwischen verstrichene Zeit in Takten gemessen werden kann. Es wurde also jeweils ein Aufruf von IRTC unmittelbar vor und nach der Iterationsschleife in den Programmcode eingefügt.

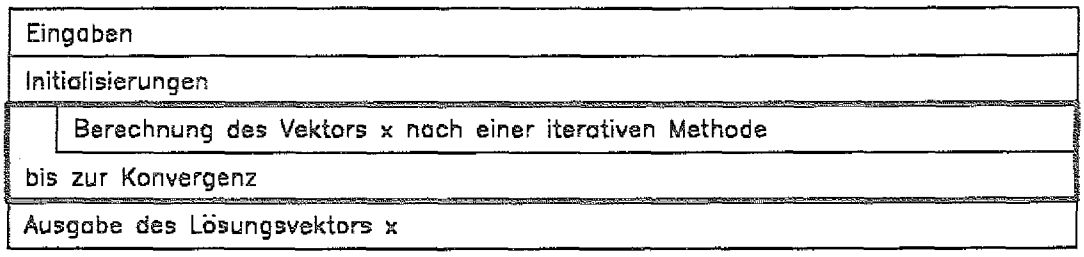


Abb. 4. Grobe Struktur aller betrachteten Programme

Als Konvergenzbedingung wurde für alle Algorithmen verlangt, daß der Vektor x sich in zwei aufeinanderfolgenden Iterationen in allen Komponenten betragsmäßig um weniger als den Wert einer Abbruchkonstante $\varepsilon > 0$ ändert. Da nur Gleichungssysteme betrachtet werden, für die die Konvergenz des jeweiligen Verfahrens sichergestellt ist, wird die Konvergenzbedingung notwendigerweise nach einer endlichen Anzahl von Iterationsschritten erfüllt.

5.2 Parallelisierung

An dieser Stelle soll zunächst der grundsätzliche Aufbau der parallelen Programmversionen vorgestellt werden; auf die zur Erreichung des Parallelismus benutzten Autotasking-Kontrollstrukturen wird in den darauffolgenden Abschnitten eingegangen.

Bei allen Programmversionen werden innerhalb eines Durchlaufs durch die Iterationsschleife alle Komponenten des Vektors x einzeln in einer DO-Schleife berechnet. Sofern diese Berechnungen unabhängig voneinander sind, bietet sich eine Parallelausführung dieser Schleife an, wodurch in jeder Iteration ein potentieller Parallelisierungsgrad der Größe n besteht, welche der Dimension des Vektors x entspricht. Dies bedeutet, daß bis zu n Komponenten theoretisch gleichzeitigt berechnet werden können, wobei die Berechnung einer Komponente x_i dann einem parallel ausführbaren Prozeß entspricht. Allerdings ist der Parallelisierungsgrad in der Praxis durch die Anzahl der Prozessoren eines Rechners nach oben begrenzt.

5.2.1 Synchrone Versionen

Das CRAY-Multitasking-Konzept Autotasking bietet die Möglichkeit zur parallelen Ausführung von DO-Schleifen, wobei eine implizite Synchronisierung aller parallel ablaufenden Prozesse am Ende der Schleife gewährleistet ist [Reg,89]. Bei paralleler Abarbeitung der Schleife, in der die Komponenten von x aktualisiert werden, ist damit sichergestellt, daß die Berechnungen aller Komponenten von x einer Iteration t abgeschlossen sind, bevor der Iterationsschritt $t+1$ beginnt. Dadurch ergibt sich also das wesentliche Merkmal eines synchronen iterativen Algorithmus (siehe Abschnitt 3.4.1).
Abb. 5 zeigt die gemeinsame Programmstruktur für alle synchronen parallelen Versionen der Verfahren.

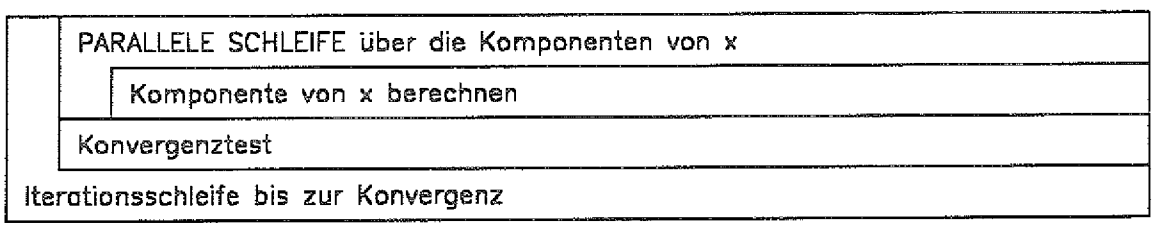


Abb. 5. Grundschema der synchronen parallelen Programmkerne

5.2.2 Asynchrone Versionen

Da es ein typisches Merkmal asynchroner iterativer Algorithmen ist, auf die Synchronisierung am Ende jedes Iterationsschrittes zu verzichten, wurde die programmtechnische Parallelisierung der asynchronen Versionen nicht auf der Ebene der Schleife über die einzelnen Komponenten von x vorgenommen. In den asynchronen Programmversionen werden die Komponenten von x aktualisiert, ohne daß nach jeder Iteration auf die Fertigstellung aller Berechnungen gewartet werden muß. Um zu erreichen, daß die Aktualisierungen parallel und unabhängig von dem jeweiligen Iterationsschritt, aber zyklisch sich wiederholend, fortgesetzt werden, wurde bei den asynchronen Programmen die Iterationsschleife parallelisiert.
In Abb. 6 ist diese Vorgehensweise verdeutlicht.

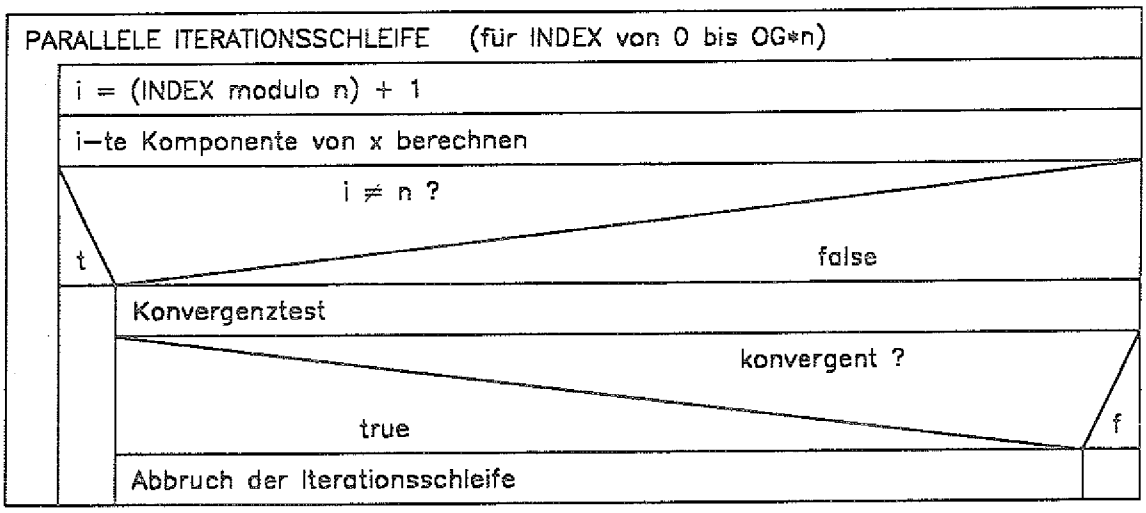


Abb. 6. Programmkern der asynchronen Versionen

Da Autotasking nicht die Möglichkeit bietet, die Iterationsschleife direkt als UNTIL-Schleife zu parallelisieren, ist diese Schleife hier durch eine DO-Schleife über eine Schleifenvariable *INDEX*, die von Null bis zu einer vorher abgeschätzten oberen Grenze *OG* läuft, realisiert worden. Dabei wurde *OG* genügend groß gewählt, um ein Ende der Iterationsschleife vor Erreichen der Konvergenz auszuschließen.

Durch die Modulo-Bildung von *INDEX* mit n und Inkrementierung um eins, wird jeweils die i-te Komponente von x angesprochen, die dadurch zyklisch alle n Berechnungen aktualisiert wird. Dabei erfolgt jeweils nach der Berechnung der n-ten Komponente ein Konvergenztest, um oben beschriebene Konvergenzbedingung auch bei der asynchronen Version zu erhalten. Je nach Ergebnis des Tests wird die Iterationsschleife dann abgebrochen oder fortgeführt.

Durch diese Vorgehensweise wird wiederum erreicht, daß die Berechnung einer Komponente von x einem parallel ausführbarem Prozeß entspricht. Allerdings sind die einzelnen Prozesse hier nicht völlig unabhängig voneinander, da beispielsweise für einen Wert von $INDEX = 0, n, 2n, 3n, \ldots$ jeweils immer die erste Komponente von x aktualisiert wird, und damit ein gleichzeitiges Berechnen für Schleifeniterationen mit diesen Werten für *INDEX* kaum Sinn machen würde.

Aus diesem Grund soll an dieser Stelle die Arbeitsweise von Autotasking bei der Zuteilung der Prozesse einer als parallel ausführbar gekennzeichneten DO-Schleife auf mehrere Tasks näher betrachtet werden, welche dann den Prozessoren zur Bearbeitung

übergeben werden. Jeder Prozeß kann dabei durch den Index der jeweiligen Schleifeniteration identifiziert werden.
Autotasking geht so vor, daß der erste Prozeß, welcher auch dem ersten Schleifendurchlauf entspricht, als erster zugeteilt wird, dann der zweite, der dritte, usw.. , bis alle an der Ausführung des parallelen Abschnitts beteiligten Tasks einen Prozeß zugeteilt bekommen haben. Sobald eine Task die Bearbeitung eines Prozesses beendet hat, wird ihr die Schleifeniteration mit dem kleinsten, noch nicht zugeteilten Index zur Bearbeitung übergeben. Dies geschieht solange, bis alle Prozesse abgearbeitet sind oder durch eine Autotasking-Direktive der vorzeitige Abbruch der parallelen Schleife erzwungen wird [Reg,89].
Dadurch daß in der Regel die Tasks in der Reihenfolge, in der die Prozesse zugeteilt wurden, auch tatsächlich von den Prozessoren bearbeitet werden, entsteht während der Abarbeitung der asynchronen Programmversionen eine in Abb. 7 verdeutlichte Situation. Dabei wird als Beispiel ein Rechnersystem mit vier Prozessoren betrachtet, auf dem ein Gleichungssystem der Größe $n = 10$ bearbeitet wird. Dargestellt sind die Bearbeitungszeiten der einzelnen Komponenten von x, und der mit K markierte Konvergenztest.

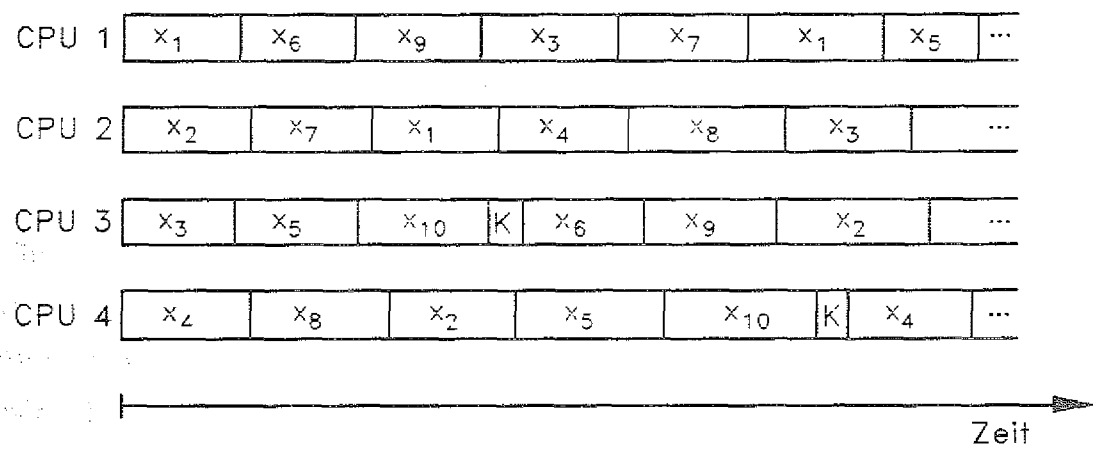


Abb. 7. Ausführung einer asynchronen Programmversion mit vier Prozessoren

Zu obiger Abbildung bleibt anzumerken, daß die Bearbeitung der Programme nur im dedizierten Betrieb auf diese Weise erfolgt, in der Praxis werden einzelne Prozessoren zwischendurch auch mit anderen Aufgaben beschäftigt sein (Multiprogramming-Betrieb).

Die asynchronen Programme nutzen also die dynamische Zuteilung der Schleifeniterationen auf die Tasks in der Reihenfolge des Schleifenindex aus, so daß ein korrektes Arbeiten gewährleistet ist.
Selbstverständlich wird die Reihenfolge der Verteilung der Prozesse auf die Prozessoren in der Regel nicht der Reihenfolge entsprechen, in der die Komponenten von x berechnet werden, da die Ausführungszeiten der Prozesse i.a. schwanken, wie auch in obiger Abbildung erkennbar ist. Doch gerade diese Tatsache beeinflußt nicht die Korrektheit asynchroner Algorithmen, da die Berechnungen der Komponenten aufgrund von Werten der anderen Komponenten aus früheren Iterationen erfolgen können.

An dieser Stelle wurde das Konzept und die generelle Vorgehensweise der asynchronen Programmversionen vorgestellt, die Beschreibung der tatsächlichen Umsetzung mit Autotasking unter Beachtung von möglicherweise auftretenden Effekten erfolgt in den nächsten Abschnitten.
Im folgenden wird die Implementierung dreier Verfahren, der SOR-Methode und dem Jacobi-Verfahren zur Lösung von linearen Gleichungssystemen, sowie dem Einschritt-Newton-SOR-Verfahren zur Lösung von nichtlinearen Gleichungssystemen, beschrieben. Dabei wird jeweils eine sequentielle, eine parallele synchrone und eine parallele asynchrone Version vorgestellt und in Bezug auf das Zeitverhalten untersucht.

5.3 Implementierung der SOR-Methode

Da bei der SOR-Methode der Wert einer Komponente, sobald er berechnet worden ist, direkt in die Berechnung der anderen Komponenten eingeht, genügt bei der Implementierung für die Speicherung des Vektors x ein eindimensionales Feld X der Größe n, welches jeweils immer die aktuellen Werte der Komponenten enthält. Für den Konvergenztest ist es deshalb erforderlich, daß der Wert einer Komponente vor deren Aktualisierung in einer Variable *XVOR* gesichert wird, um anschließend beide Werte zu vergleichen.
Es ergibt sich damit in FORTRAN folgender Programm-Code für den Programmkern der sequentiellen SOR-Methode:

```
C     Iterationsschleife
C   REPEAT
100     CONTINUE
C   Konvergenzflag zu Beginn jeder Iteration setzen
          KONV = .TRUE.
C   Schleife über die Komponenten von X
          DO 40 I=1,N
            SUM = 0
            DO 50 J=1,N
              SUM = SUM + A(J,I)*X(J)
50          CONTINUE
            SUM = SUM - A(I,I)*X(I)
C   Wert von X retten
            XVOR = X(I)
C   X nach SOR-Methode berechnen
            X(I) = (1-OMEGA)*X(I) - OMEGA/A(I,I)*(SUM-B(I))
C   Konvergenzflag zurücksetzen, falls |X(I) - XVOR| > ε
            IF ((KONV) .AND. (ABS(X(I)-XVOR)) .GE. EPS)) KONV = .FALSE.
C   Schleifenende über die Komponenten von X
40        CONTINUE
          IF (.NOT. KONV) GOTO 100
C   UNTIL Konvergenzkriterium erfüllt
```

Sequentielle SOR-Methode

Die benutzten Konstanten und Variablen sowie ihre Bedeutung ergeben sich dabei wie folgt:

N	Anzahl der Gleichungen
EPS	Abbruchkonstante
OMEGA	Relaxationsparameter ω
KONV	Konvergenzflag vom Typ LOGICAL
SUM	Die nach der Berechnungsvorschrift der SOR-Methode zu bildende Summe
A(N,N)	Matrix A
X(N)	Aktueller Wert des Vektors x
XVOR	Wert der vorigen Iteration einer Komponente von x
B(N)	Rechte-Seite-Vektor b

Die hier vorgestellten Variablen und Konstanten werden in den folgenden Programmversionen immer in obiger Bedeutung benutzt.

Bemerkt werden soll an dieser Stelle, daß die Bildung von *SUM* auf obige Weise der besseren Vektorisierbarkeit dient.

5.3.1 Synchrone SOR-Methode

Wie schon in Abschnitt 4.1.2 erwähnt, verbietet das Gauß-Seidel-Verfahren wie auch dessen Verallgemeinerung, die SOR-Methode, wegen der an sich notwendigerweise streng sequentiellen Berechnungsreihenfolge der Komponenten eine parallele Ausführbarkeit. In den letzten Jahren zeigen sich in der Literatur jedoch auch Ansätze, die SOR-Methode zu parallelisieren. Ein Ansatz, welcher in [B&E,82] vorgeschlagen wird, ist die Betrachtung von Gleichungssystemen mit Matrizen, die property A aufweisen (siehe Kapitel 4.3).
Unter Ausnutzung der Tatsache, daß bei diesen Gleichungssystemen zwei disjunkte Mengen von Indizes bestehen, wobei Berechnungen innerhalb einer Menge voneinander unabhängig sind, wird bei der synchronen Programmversion die Schleife 40 über die Komponenten in zwei Schleifen aufgeteilt. Diese beiden Schleifen berechnen jeweils eine Komponentenmenge und können jeweils parallel ausgeführt werden. Da am Ende jeder parallelen Schleife eine Synchronisation durchgeführt wird, ist die Unabhängigkeit der Komponenten von x, die gleichzeitig berechnet werden, gesichert.

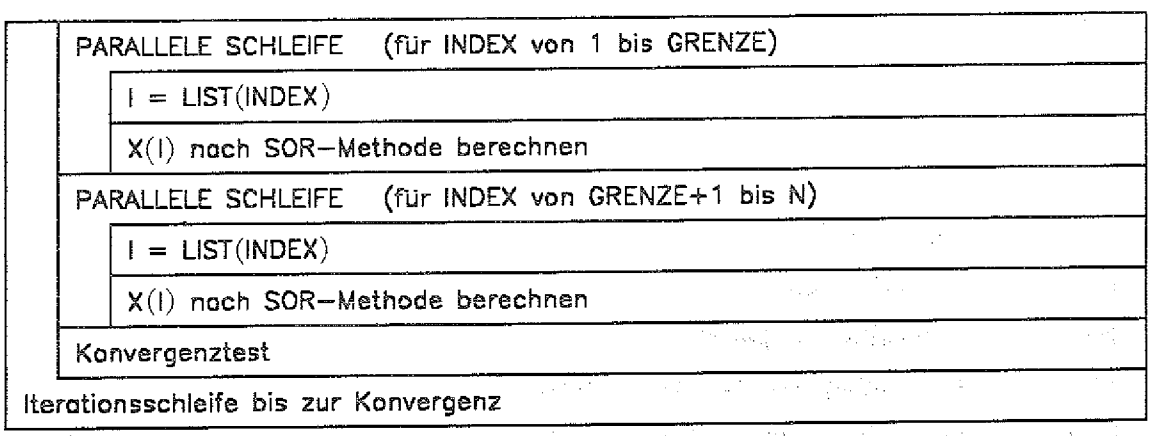


Abb. 8. Synchrone SOR-Methode

Um die Verteilung der beiden disjunkten Indexmengen auf die beiden Schleifen zu realisieren, wird ein gemeinsames Feld *LIST* der Größe N benutzt. *LIST* wird vor Ausfüh-

rung der Iterationsschleife so initialisiert, daß in einem ersten Teil die Indizes der einen Menge abgelegt werden, und in dem verbleibenden Teil die Indizes der zweiten Menge. Dabei markiert eine Variable *GRENZE* das Ende der ersten Indexmenge in *LIST*. Bei den betrachteten Gleichungssystemen sind die beiden disjunkten Indexmengen etwa gleich groß, so daß auch der Rechenaufwand für die beiden Schleifen etwa gleich ist.
In Abb. 8 erkennt man den Aufbau des synchronen Programms für die SOR-Methode.

5.3.2 Umsetzung der synchronen SOR-Methode mit Autotasking

Für die Erstellung der parallelen synchronen Programmversion für die SOR-Methode wurde die Autotasking-Direktive CMICS DO ALL für die Parallelisierung der Schleifen über die Komponenten von x benutzt. Durch Einfügen dieser Direktive jeweils unmittelbar vor den beiden Schleifen wurden diese als parallel ausführbar gekennzeichnet, und von Autotasking in parallel ausführbaren Code übersetzt.
Da Autotasking diese Direktive nicht selbsttätig in den Programmcode einfügte, war es zusätzlich notwendig, den Geltungsbereich der in dem parallelen Abschnitt benutzten Variablen festzulegen. Dabei wird unterschieden zwischen *private* Variablen, die als lokale Variablen für jede Task anzusehen sind, und *shared* Variablen, die gemeinsam von allen gleichzeitig arbeitenden Tasks benutzt werden. Da die Werte von *A*, *B*, *X*, *LIST*, *GRENZE* und *KONV* global für alle Tasks bekannt sein müssen, wurden diese Variablen als *shared* deklariert. Dahingegen werden die Werte für *SUM*, *XVOR*, *J* und *I* nur lokal verwendet, so daß diese Variablen den *private* Variablen zugeordnet wurden.
Als Option zu der Direktive *CMICS DO ALL* ist es möglich, durch Angabe von *CHUNKSIZE(ZAHL)* ein Zusammenfassen von *ZAHL* Schleifeniterationen zu einem Prozeß zu erhalten. Dadurch wird Overhead bei der Verteilung der Schleifeniterationen auf die Tasks eingespart. Im Abschnitt zu den Leistungsmessungen wird darauf noch näher eingegangen. Alternativ dazu kann durch den Einsatz von *NUMCHUNKS(CPUS)* eine optimale Aufteilung der Iterationen einer parallelen Schleife auf *CPUS* Prozesse erhalten werden, wobei *CPUS* die Anzahl der an der Ausführung beteiligten Prozessoren angibt. Letztere Option wurde bei der synchronen SOR-Methode angewendet.
Es ergab sich folgender Programmcode für den Kern der parallelen synchronen SOR-Methode, wobei die benutzten Direktiven zur Hervorhebung fett gedruckt sind.

```
C      Iterationsschleife
C    REPEAT
100     CONTINUE
          KONV = .TRUE.
C    Parallele Schleife über die ersten Komponenten von X
CMIC$ DO ALL SHARED(A,B,X,KONV,LIST,GRENZE)
CMIC$1       PRIVATE(SUM,XVOR,I,J,INDEX) NUMCHUNKS(CPUS)
          DO 41 INDEX = 1,GRENZE
            I = LIST(INDEX)
            XVOR = X(I)
C            .
C            .   X(I) nach SOR-Methode berechnen
C            .
            IF ((KONV) .AND. (ABS(X(I)-XVOR)) .GE. EPS)) THEN
             KONV = .FALSE.
CDIR$ SUPPRESS KONV
            ENDIF
41        CONTINUE

C    Parallele Schleife über die zweiten Komponenten von X
CMIC$ DO ALL SHARED(A,B,X,KONV,LIST,GRENZE)
CMIC$1       PRIVATE(SUM,XVOR,I,J,INDEX) NUMCHUNKS(CPUS)
          DO 42 INDEX = GRENZE+1,N
            I = LIST(INDEX)
            XVOR = X(I)
C            .
C            .   X(I) nach SOR-Methode berechnen
C            .
            IF ((KONV) .AND. (ABS(X(I)-XVOR)) .GE. EPS)) THEN
             KONV = .FALSE.
CDIR$ SUPPRESS KONV
            ENDIF
42        CONTINUE

         IF (.NOT. KONV) GOTO 100
C    UNTIL Konvergenzkriterium erfüllt
```

Synchrone SOR-Methode

Schwierigkeiten ergaben sich bei der Implementierung mit der Zuweisung der *shared* Variablen *KONV* mit dem Wert *.FALSE.*. Weil für die korrekte Programmausführung nicht notwendigerweise der exklusive Zugriff einer Task auf diese Variable erforderlich ist, wurde auf die Benutzung eines kritischen Abschnittes, der sich durch Einfügung der

Autotasking-Direktiven *CMICS GUARD, CMICS END GUARD* realisieren läßt, verzichtet. Dies wurde nicht zuletzt wegen des mit der Direktive verbundenen hohen Overheads vermieden. Da die Werte von *shared* Variablen bei der Anwendung von Autotasking in Registern gehalten werden, kann es allerdings vorkommen, daß ein veränderter Wert von *KONV* nicht sofort allen an der Ausführung beteiligten Tasks bekannt ist. Um eine fehlerhafte Programmausführung aus diesem Grund zu vermeiden, wurde direkt nach Beschreibung der Variablen mit *.FALSE.* die Compiler-Direktive *CDIRS SUPPRESS KONV* in den Code eingefügt. Diese Direktive erzwingt ein Zurückspeichern des Wertes von *KONV* aus dem Register in den Speicher, so daß anschließend ein Datenaustausch der Tasks stattfindet [Nag,90].

5.3.3 Asynchrone SOR-Methode

In obiger synchronen Version der SOR-Methode muß durch Synchronisation gewährleistet werden, daß die Komponenten der einen Menge aktualisiert worden sind, bevor mit der Berechnung der Komponenten aus der anderen Menge begonnen werden darf. Die im folgenden beschriebene asynchrone Version basiert auf der synchronen Version, verzichtet aber auf diese Bedingung.

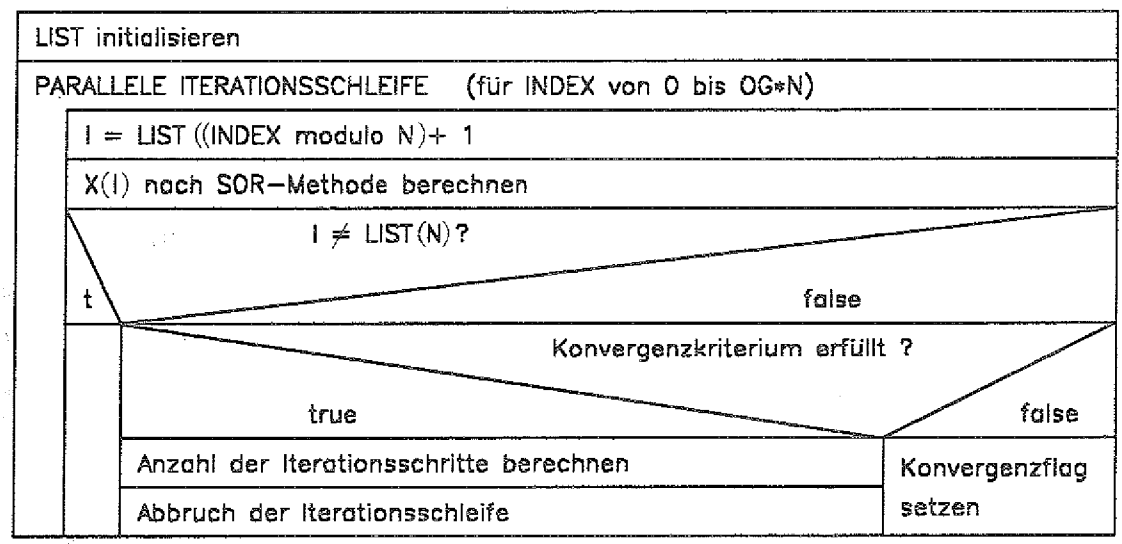


Abb. 9. Asynchrone SOR-Methode

Die Berechnung der Komponenten bei der asynchronen SOR-Methode erfolgt nicht in zwei getrennten Schleifen, sondern die zyklische Aktualisierung geschieht in einer parallelen Schleife, der Iterationsschleife. Trotzdem geschieht der Zugriff auf die einzelnen Elemente von X auch hier über die *shared* Variable *LIST*, um möglichst wenige überlappende Berechnungen der Komponenten aus beiden Mengen während der parallelen Abarbeitung zu erhalten. Die Liste der Indizes *LIST* enthält, wie auch bei der synchronen Version, in einem ersten Teil die Indizes der einen Komponentenmenge und im zweiten Teil die Indizes der anderen Menge, die dann der Reihe nach in der Schleife angesprochen werden. Dadurch kann zwar eine gleichzeitige Berechnung von Komponenten aus den beiden Mengen nicht völlig vermieden werden, jedoch wirkt sich diese Tatsache nicht auf die Korrektheit des asynchronen Programms aus. Allerdings können sich Einflüsse auf die Anzahl der benötigten Iterationsschritte bis zur Konvergenz ergeben, da nicht vorhergesagt werden kann, auf welchen Werten aus welchen Iterationsschritten diese überlappenden Berechnungen basieren. Damit zeigt sich jedoch ein charakteristisches Merkmal asynchroner Algorithmen.
Das Schema des asynchronen Programmkerns für die SOR-Methode ist in Abb. 9 dargestellt.

5.3.4 Umsetzung der asynchronen SOR-Methode mit Autotasking

Für die Implementierung des asynchronen Programms mit Autotasking wurde wiederum die Direktive *CMICS DO ALL* zur Parallelisierung der Iterationsschleife in den Programmcode eingefügt.
Um nach Beendigung des Programms die Anzahl der benötigten Iterationsschritte angeben zu können, wird eine zusätzliche *shared* Integer-Variable *ITER* eingeführt, die bei Konvergenz des Verfahrens den durch N geteilten Wert der *private* Variablen *INDEX* erhält. Diese Ermittlung der Iterationsschritte wurde dem Inkrementieren eines Iterationszählers jeweils nach der Aktualisierung der n-ten Komponente wegen des geringeren Rechenaufwands vorgezogen.
Dadurch ergab sich folgender Programmcode für die asynchrone SOR-Methode:

```
          KONV = .TRUE.
C     Parallele Iterationsschleife
CMIC$ DO ALL SHARED(A,B,X,KONV,LIST,ITER) PRIVATE(SUM,XVOR,I,J,INDEX)
          DO 100 INDEX = 0,OG*N
            I = LIST(MOD(INDEX,N)+1)
            XVOR = X(I)
C            .
C            .   X(I) nach SOR-Methode berechnen
C            .
            IF ((KONV) .AND. (ABS(X(I)-XVOR)) .GE. EPS)) THEN
              KONV = .FALSE.
CDIR$ SUPPRESS KONV
            ENDIF

C   Konvergenztest jeweils nachdem die N-te Komponente berechnet wurde
            IF (I .NE. LIST(N)) GOTO 100
            IF (KONV) THEN
              ITER=INDEX/N
CMIC$ SOFT EXIT
              GOTO 101
            ENDIF

C   Konvergenzflag für die nächste Iteration initialisieren
            KONV = .TRUE.
C   Ende der parallelen Iterationsschleife
100       CONTINUE
C   Sprungziel bei Konvergenz
101       CONTINUE
```

Asynchrone SOR-Methode

Zur Beendigung der parallelen Iterationsschleife im Falle der Konvergenz wurde hier die Autotasking-Direktive *CMIC$ SOFT EXIT* eingesetzt. *CMIC$ SOFT EXIT* bewirkt, daß die Zuteilung weiterer Prozesse an die Tasks unmittelbar gestoppt wird. Die zum Zeitpunkt der Ausführung der Direktive gerade aktiven Tasks führen ihre Berechnungen zu Ende und werden synchronisiert. Dies bedeutet, daß die Tasks solange warten, bis alle an der Berechnung beteiligten Tasks ihre Arbeit beendet haben. Anschließend wird die Codeabarbeitung an der durch die Sprungmarke gekennzeichneten Stelle fortgeführt. Auf die Verwendung von kritischen Abschnitten konnte auch bei der asynchronen Programmversion, unter Benutzung der Compiler-Direktive *CDIR$ SUPPRESS KONV*, ganz verzichtet werden.

5.4 Leistungsmessungen zur SOR-Methode

Die Zeitmessungen für die SOR-Methode wurden auf lineare Gleichungssysteme angewendet, die sich aus der Diskretisierung des Dirichletschen Randwertproblems

$$-u_{xx} - u_{yy} = 0 \qquad (0 < x, y < 1)$$

ergeben, mit $u(x, y) = 0$ für (x, y) aus dem Rande des Einheitsquadrats. Dabei wurden drei Systeme der Größen 64, 256 und 1024 betrachtet, die aus Gittern der Größen 8×8, 16×16 und 32×32 gebildet wurden.
In Abb. 10 sind die Laufzeiten der synchronen und asynchronen SOR-Methode auf der CRAY Y-MP in dedizierter Umgebung für die Lösung dieser Gleichungssysteme gegenübergestellt.

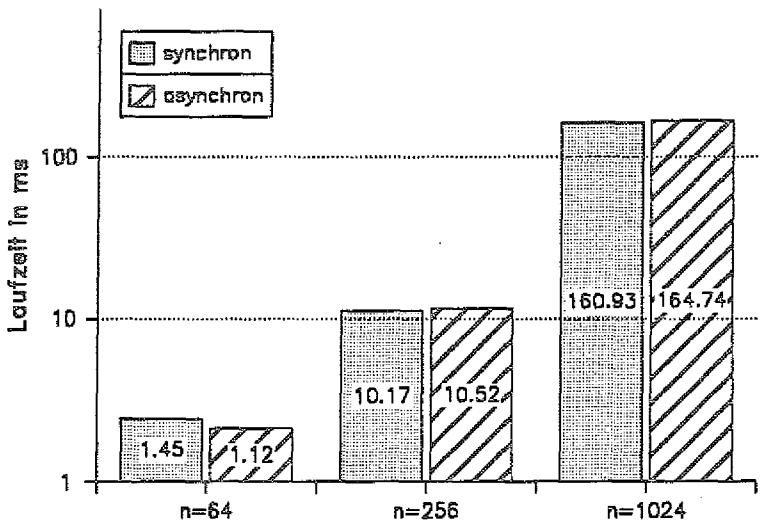


Abb. 10. Gegenüberstellung der Laufzeiten synchrones/asynchrones SOR

Man beachte, daß in obigem Diagramm eine logarithmische Skaleneinteilung für die Ordinate gewählt wurde, da sonst eine Darstellung in einem Schema wegen der großen Differenzen in den Laufzeiten der einzelnen Gleichungssysteme nicht möglich gewesen wäre.
Es wird deutlich, daß sich die Laufzeiten jeweils zwischen den synchronen und den asynchronen Programmen bei allen drei Gleichungssystemen kaum unterscheiden. Al-

lerdings kann man mit zunehmender Größe der Gleichungssysteme etwas geringere Laufzeiten für die synchrone Version im Vergleich zur asynchronen Version feststellen. Um dies zu erklären, sollen die wesentlichen Unterschiede zwischen den synchronen und asynchronen Programmen sowie deren Auswirkung auf die Laufzeiten betrachtet werden.

Bei der synchronen SOR-Methode fällt in jedem Iterationsschritt Overhead für die zweimalige Ausführung der Direktive *CMIC$ DO ALL* an, der sich bei der asynchronen Version für die Parallelisierung der Iterationsschleife nur insgesamt einmal ergibt. Allerdings wird der Aufwand der Direktive vom Aufwand für die Bestimmung des als nächstes auszuführenden Prozesses dominiert [Reg,89]. Durch die Angabe von *NUMCHUNKS(CPUS)* bei dem synchronen Programm wird die Anzahl der Prozesse in jeder parallelen Schleife auf *CPUS* begrenzt, so daß der Overhead gering ausfällt. Außerdem entsteht zusätzlicher Zeitaufwand für die Direktive *CMIC$ SOFT EXIT* bei den asynchronen Programmen.

Um diejenige Zeit zu messen, die nur von den verwendeten Autotasking-Direktiven verursacht wird, bietet sich das Einfügen der Direktive *CMIC$ GETCPUS(1)* vor dem Programmkern in den Code an. Dadurch wird bewirkt, daß der Taskscheduler für die Ausführung des parallelen Programms nur eine Task zur Verfügung stellt. Bildet man die Differenz der Zeiten für diese Ausführungsart des Programms und der sequentiellen Ausführung desselben Programms, wobei also die Direktiven als Kommentar gedeutet werden, so kann man die Zeiten erhalten, die lediglich durch die Direktiven verursacht werden. Man erkennt, daß der Zeitanteil für die Direktiven bei dem synchronen Programm etwas niedriger ist als bei dem asynchronen Programm für die SOR-Methode. Allerdings wird auch deutlich, daß die reine Rechenzeit für die asynchrone Version im Vergleich zu der synchronen Version höher ist (etwa 9 ms für $n = 256$). Die Begründung dafür ist im zusätzlichen Rechenaufwand der asynchronen SOR-Methode gegenüber der synchronen Version zu sehen, der durch die Modulo-Bildung beim Zugriff auf die einzelnen Komponenten von x entsteht, und durch die Abfrage in jedem Durchlauf durch die Iterationsschleife, ob die n-te Komponente berechnet wurde.

Zwar wird die Modulo-Bildung durch den Aufruf einer sehr effizienten CRAY-Intrinsic-Funktion *MOD* realisiert, jedoch muß dieser Aufruf jeweils für jede Komponente erfolgen, also insgesamt $n \times$Anzahl der Iterationsschritte oft, so daß ein nicht zu vernachlässigender Zusatzaufwand anfällt. Der durch die Modulo-Bildung verursachte Rechenaufwand läßt sich verringern durch Ausnutzung der Tatsache, daß die Größen der betrachteten Gleichungssysteme Potenzen von zwei sind, indem die Modulo-Bildung durch eine bitweise logische Und-Bildung mit n-1 ersetzt wird. Diese Intrinsic-Bitfunktion zeichnet sich durch geringeren Zeitaufwand aus, so daß sich die Laufzeiten der asynchronen SOR-Programme um etwa 3 Prozent verringern. Um aber eine gene-

relle Anwendbarkeit für beliebige n beizubehalten, wird diese Möglichkeit nicht weiter verfolgt.

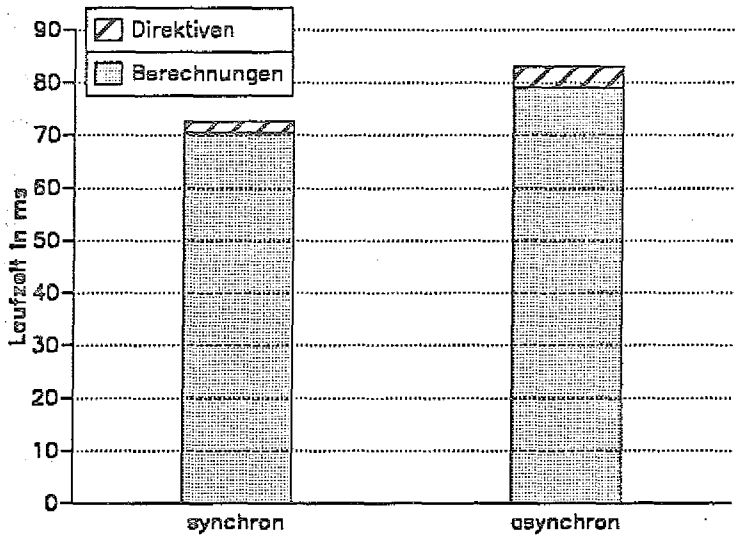


Abb. 11. Zeiten für Direktiven und Berechnungen bei sequentieller Ausführung auf der CRAY Y-MP

Abb. 11 zeigt die durch obige Vorgehensweise erhaltenen Anteile an den Zeiten für die synchrone und die asynchrone Version am Beispiel $n = 256$.

Bisher gingen in die Zeitbetrachtungen nur die reinen Rechenzeiten und der Overhead für die Autotasking-Direktiven ein. Ein wesentlicher Einflußfaktor auf die Laufzeit sind aber auch die Wartezeiten. Bei dem hier untersuchten synchronen SOR-Programm können wegen der gleichen Lastverteilung auf die einzelnen Tasks in den parallelen Schleifen Wartezeiten nur dann entstehen, wenn die Berechnungszeiten für die einzelnen Komponenten schwanken, oder wenn eine Task durch den Entzug des Prozessors suspendiert wird. Diese beiden Effekte treten in einer dedizierten Umgebung auf einem CRAY-Rechner allerdings kaum auf. Um aber den verschiedenen Einfluß der Wartezeiten auf die synchronen und asynchronen Programme zu betrachten, wurde die Option *CHUNKSIZE(ZAHL)* eingesetzt. Dieser Parameter bewirkt ein Zusammenfassen von *ZAHL* Schleifeniterationen zu einem Prozeß, der mit *Chunk* bezeichnet wird. Die Anzahl der Schleifeniterationen in einem Chunk bezeichnet man mit *Chunksize*. In folgender Vorgehensweise wird die Tatsache ausgenutzt, daß bei einer Chunksize, die nicht ohne

Rest durch die Anzahl der für die Ausführung des Programms zur Verfügung stehenden Prozessoren teilbar ist, auch Wartezeiten auftreten. Dies ist damit zu begründen, daß Autotasking die Pakete von Schleifeniterationen der Größe *ZAHL* nacheinander auf die Tasks verteilt. Die dann noch verbleibenden Schleifeniterationen werden gesamt einer Task zugeteilt, so daß die übrigen Tasks während der Synchronisation am Ende der parallelen Schleife auf das Fertigstellen der Berechnungen dieser Task warten. In Abb. 12 sind die Laufzeiten für die verschiedenen Größen der Chunks am Beispiel $n = 256$ dargestellt. Es wurde dabei der maximale Wert 16 für die Chunksize gewählt, da in dem synchronen Programm jede parallele Schleife über die Komponenten von x 128 Iterationen enthält und acht Prozessoren benutzt wurden. Auch für das asynchrone Programm macht eine größere Chunksize als 16 keinen Sinn, trotz der um ein vielfaches höheren Durchläufe durch die parallele Iterationsschleife, da sonst überlappende Berechnungen der Komponenten aus beiden Indexmengen geradezu erzwungen würden.

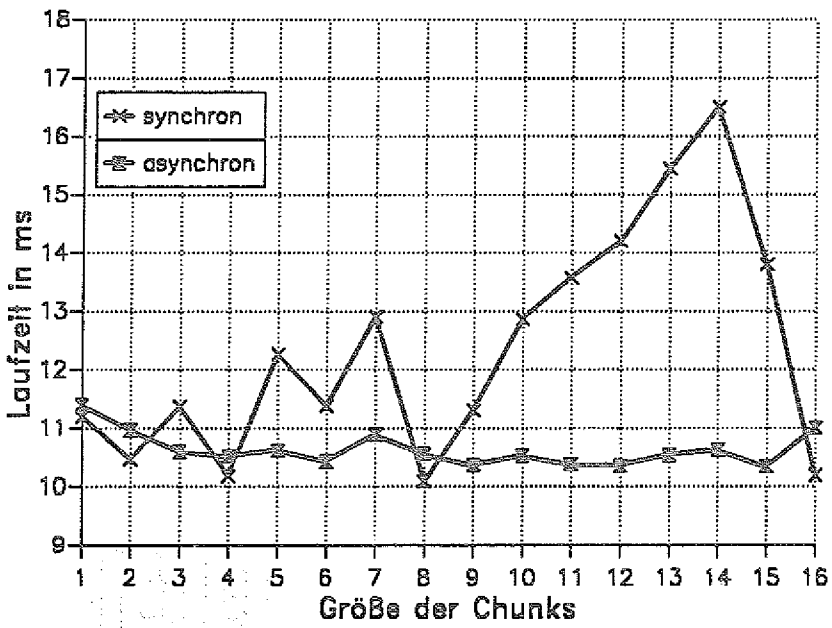


Abb. 12. Laufzeiten bei verschiedenen Chunksizes

Bei den Laufzeiten des synchronen Programms erkennt man, daß die Minima bei den Chunksizes liegen, durch die 128 ohne Rest teilbar ist, wodurch eine gleiche Lastverteilung auf die acht Prozessoren entsteht. Das Maximum liegt bei 14, da hier am Ende einer jeden parallelen Schleife 15 Iterationen von einer einzelnen Task ausgeführt wer-

den, während die übrigen Tasks warten. Durch die notwendige häufige Synchronisation bei der Ausführung des synchronen SOR-Programms können also Wartezeiten entstehen, die die Laufzeit erhöhen. Dahingegen stellt sich die asynchrone SOR-Methode als relativ unempfindlich gegenüber variierender Chunksize heraus, weil hier lediglich am Ende der Iterationsschleife nur einmal synchronisiert wird. Die leichten Schwankungen in der Laufzeit des asynchronen Programms lassen sich mit der bis zur Konvergenz benötigten unterschiedlichen Anzahl von Iterationsschritten erklären. Ein Zusammenhang zwischen Chunksize und benötigten Iterationsschritten läßt sich jedoch nicht feststellen. Deutlich wird auch, daß die Einsparung des Overheads durch Zusammenfassen von mehreren Schleifeniterationen zu einem Prozeß kaum Auswirkungen auf die Laufzeiten des asynchronen SOR-Programms hat, wenn man die Werte aus Abb. 10 mit denjenigen in Abb. 12 vergleicht. Deshalb wurde auf die Benutzung der Option *CHUNKSIZE* für die asynchronen Programmversionen im weiteren verzichtet.
Im Multiprogramming-Betrieb empfielt es sich wegen der möglicherweise schwankenden Zahl an ausführenden Prozessoren und den Schwankungen der Ausführungszeiten kleine Prozeßgrößen zu verwenden, da dadurch auch nur kurze Wartezeiten entstehen können.

In Abb. 13 ist das Verhältnis von sequentieller und paralleler Laufzeit der synchronen SOR-Methode auf der CRAY Y-MP für verschiedene Anzahlen von Prozessoren dargestellt.

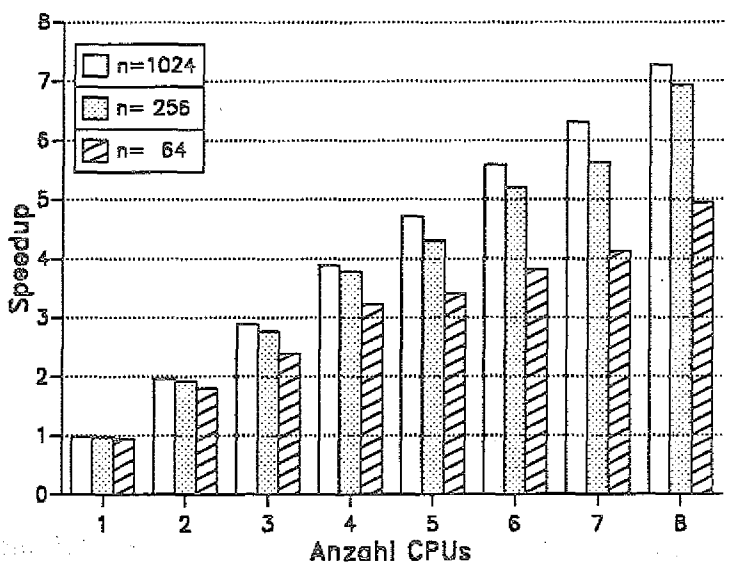


Abb. 13. Speedup der synchronen SOR-Methode

Um die Zahl der an der Ausführung des Programms beteiligten Tasks festzulegen, wurde wiederum die Direktive *CMIC$ GETCPUS(CPUS)* verwendet.

Man erkennt, daß der Speedup um so höher liegt, je größer die Ordnung des Gleichungssystems ist. Dies resultiert aus der Tatsache, daß bei größeren n auch die Rechenzeit für einen Schleifendurchlauf der parallelen Schleifen zunimmt. Dadurch wird das Verhältnis von Overhead zur Rechenzeit des Programms kleiner, und somit steigt der Speedup an. Deutlich wird auch, daß mit der Anzahl der Prozessoren auch die Differenz zwischen dem idealen Speedup, der gleich der Anzahl der Prozessoren ist, und dem erreichten Speedup zunimmt. Dies ergibt sich aus der Zunahme des Synchronisations-Overheads bei steigender Anzahl von Prozessoren, die an der Ausführung des Programms beteiligt sind.

Die geringen Unterschiede bei den Speedup-Werten der beiden kleineren Gleichungssysteme beim Übergang von vier auf fünf Prozessoren und der relativ starke Anstieg beim Übergang von sieben auf acht Prozesssoren resultieren aus der Tatsache, daß für vier und acht Prozessoren eine Aufteilung der Schleifeniterationen auf die ausführenden Tasks ohne Rest möglich ist.

Für $n = 64$ ergeben sich nur relativ schlechte Speedup-Werte, da der Synchronisations-Overhead im Verhältnis zum Rechenaufwand der parallelen Schleifen groß ist.

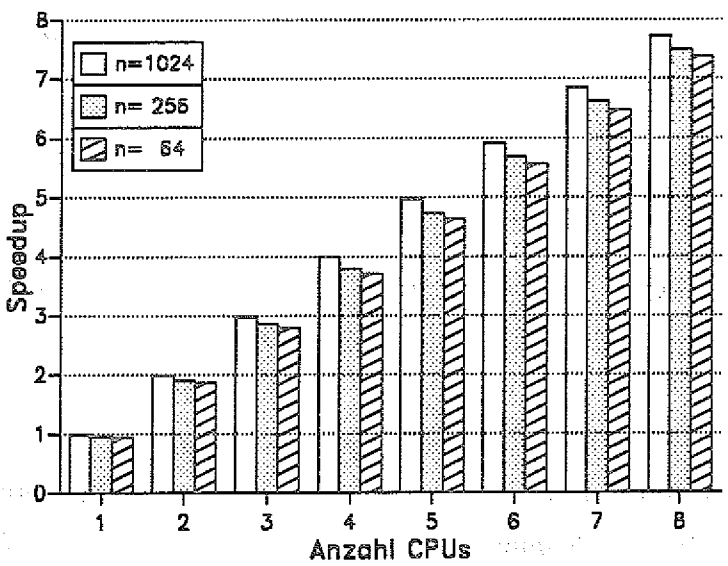


Abb. 14. Speedup der asynchronen SOR-Methode

Bei den Speedup-Werten der asynchronen SOR-Methode stellt man geringere Unterschiede zwischen den verschiedenen Größen der Gleichungssysteme als bei der synchronen Methode fest. Auch ergeben sich höhere Werte als bei der synchronen Methode. Dies ist mit dem kleineren Anteil des Synchronisations-Overheads im Vergleich zu den Rechenzeiten zu begründen, der sich dadurch insbesondere bei kleinen n nicht so stark im Speedup bemerkbar macht.
Wenn man allerdings das Verhältnis der Laufzeiten der beiden parallelen SOR-Programme zu der Laufzeit der sequentiellen SOR-Methode bildet, liegen die Werte für das asynchrone Programm etwas niedriger wegen der höheren sequentiellen Laufzeit der asynchronen Version.
In Abb. 15 ist dieser Sachverhalt für acht Prozessoren verdeutlicht.

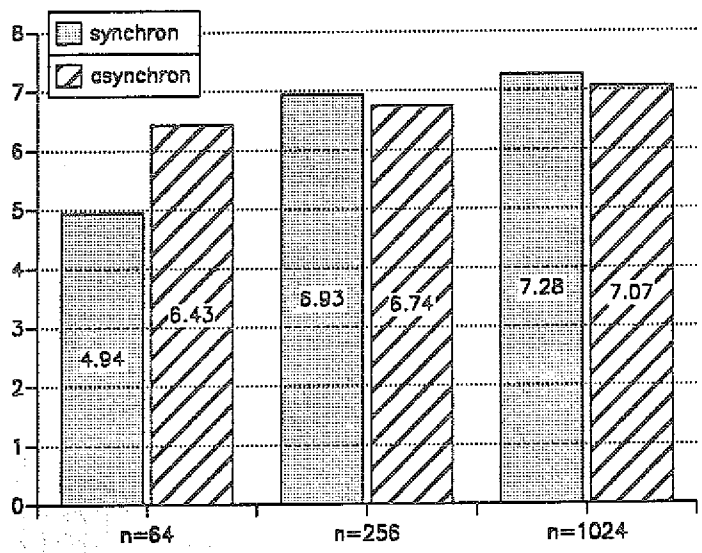


Abb. 15. Verhältnis der Laufzeiten parallele/sequentielle Verfahren zur SOR-Methode

5.5 Implementierung des Jacobi-Verfahrens

Da die im folgenden vorgestellten Programme, die das Jacobi-Verfahren anwenden, Gemeinsamkeiten mit der in Kapitel 5.3 beschriebenen Implementierung der SOR-Methode aufweisen, soll bei der Beschreibung der Implementierung des Jacobi-Verfahrens insbesondere auf die Unterschiede eingegangen werden.
Wie bereits in Abschnitt 4.1.1 erwähnt, ist das Jacobi-Verfahren ein Gesamtschrittverfahren, welches in jedem Iterationsschritt die Werte aller Komponenten aus der vorigen

Iteration benötigt. Aus diesem Grunde ist die Variable *XVOR* nun als Feld der Größe *N* deklariert, und enthält immer jeweils den Wert aller Komponenten aus der vorherigen Iteration. Sobald mit der Berechnung eines neuen Iterationsschritts begonnen wird, werden in einer Schleife alle Komponenten der Vorgängeriteration von *x* mit den aktuellen Werten beschrieben. Erst nachdem diese Schleife abgearbeitet ist, wird mit der erneuten Berechnung aller Komponenten von *x* begonnen, so daß sich das in Abb. 16 dargestellte Programmschema für das sequentielle Jacobi-Verfahren ergibt.

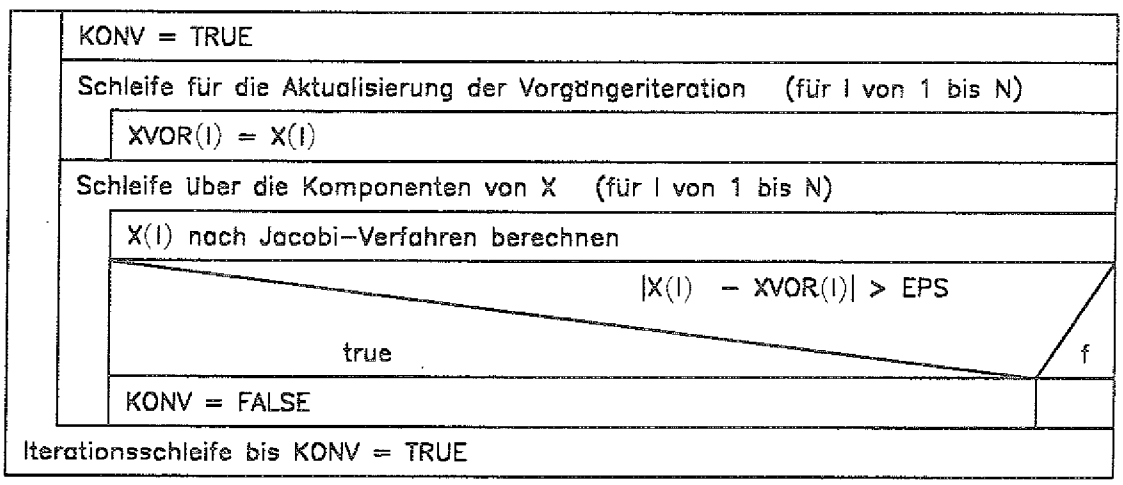


Abb. 16. Sequentielles Jacobi-Verfahren

5.5.1 Synchrones Jacobi-Verfahren

Bei der parallelen synchronen Implementierung des Jacobi-Verfahrens ist es möglich, die Schleife über die Komponenten von *x* zu parallelisieren, unter Ausnutzung der Tatsache, daß die Berechnungen aller Komponenten nur auf den Werten der vorherigen Iteration beruhen, und damit untereinander unabhängig sind. Ein Vorteil gegenüber der SOR-Methode ist deshalb die generellere Anwendbarkeit des parallelen Jacobi-Verfahrens auf lineare Gleichungssysteme, deren Matrizen nicht property A aufweisen müssen.

Für die Schleife, in der die Werte von *XVOR* mit den Werten von *X* aktualisiert werden, lohnt sich eine Parallelisierung nicht. Es würde im Verhältnis zur geringen Rechenzeit einer Schleifeniteration ein zu großer Overhead entstehen. Außerdem vektorisiert diese Schleife sehr gut, so daß auf eine Parallelausführung verzichtet wurde.

Für die programmtechnische Realisierung des synchronen Jacobi-Verfahrens wurde wiederum die Autotasking-Direktive *CMICS DO ALL* in Verbindung mit der Option *NUMCHUNKS(CPUS)* eingesetzt, mit dem Unterschied zur SOR-Methode, daß *XVOR* nun als *shared* Variable gekennzeichnet sein muß, da alle Tasks die Werte des Feldes für ihre Berechnungen benötigen.

5.5.2 Ein asynchrones Jacobi-ähnliches Verfahren

Asynchrone Algorithmen haben die Eigenschaft, daß keine strenge Festlegung des Aktualisierungsgrades der in die Berechnungen eingehenden Werte vorliegt. Die Berechnungsvorschrift für das Jacobi-Verfahren verlangt jedoch, daß genau die Werte der Vorgängeriteration in die Berechnung von x eingehen. Das hier vorgestellte asynchrone Verfahren basiert auf der Jacobi-Berechnungsvorschrift, hält sich aber nicht an die strenge Vorschrift, nur Werte aus der Vorgängeriteration zu benutzen.

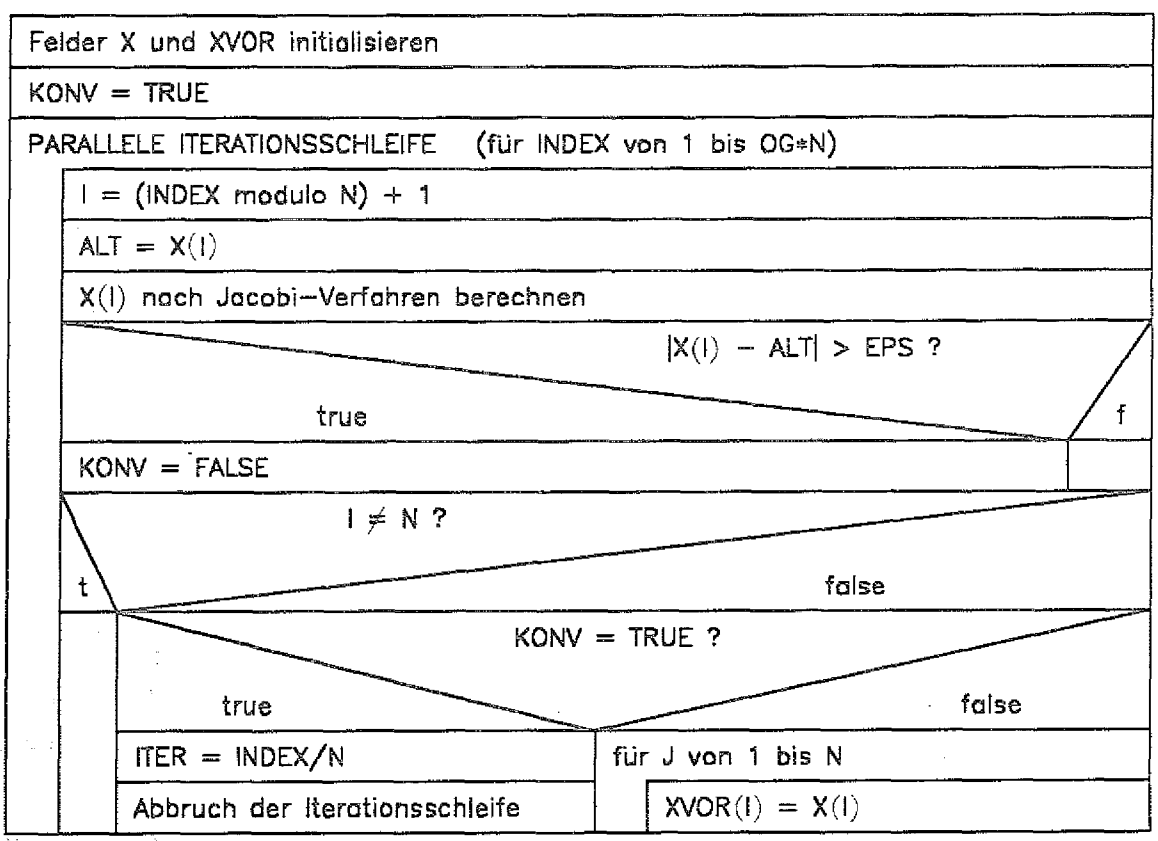


Abb. 17. Asynchrones Jacobi-ähnliches Verfahren

Wiederum wurde das asynchrone Verhalten des Verfahrens durch Parallelisierung der Iterationsschleife erreicht, so daß sich das in Abb. 17 verdeutlichte Schema für das asynchrone Jacobi-ähnliche Programm ergibt.

Durch die Aktualisierung der *XVOR*-Werte jeweils nach der Berechnung der *n*-ten Komponente wird angestrebt, die Berechnungsvorschrift des Jacobi-Verfahrens zu erfüllen, erst bei Beginn eines neuen Iterationsschrittes die Vorgängeriteration neu festzusetzen. Da bei Ausführung dieses asychronen Programms i.a. jedoch während der Aktualisierung der Vorgängeriteration andere Komponenten gleichzeitig berechnet werden, kann in diesem Verfahren nicht garantiert werden, daß Berechnungen nicht aufgrund von Werten aus anderen als der Vorgängeriteration durchgeführt werden. Dadurch ergibt sich ein Einfluß auf die Anzahl der bis zur Konvergenz benötigten Iterationsschritte, die gegenüber dem sequentiellen Jacobi-Verfahren nach oben und unten abweichen kann.
Bemerkt werden soll an dieser Stelle, daß sich durch die Aktualisierung des gesamten Feldes *XVOR* von derjenigen Task, welche die *n*-te Komponente aktualisiert, deren Ausführungszeit im Vergleich zu den übrigen Tasks, die nur die Berechnung einer Komponente durchführen, verlängert. Dies resultiert in einer leicht unbalancierten Lastverteilung, die sich jedoch wegen der großen Anzahl von Durchläufen der parallelen Iterationsschleife und dem Nichtvorhandensein von Wartezeiten nicht negativ auswirkt.

Die für die parallele Implementierung dieses Verfahrens benutzten Kontrollstrukturen sind die gleichen wie bei der asynchronen SOR-Methode und werden auch in der gleichen Weise angewandt, so daß auf den parallelen Programmcode des asynchronen Jacobi-ähnlichen Verfahrens hier verzichtet wird.

5.6 Leistungsmessungen zum Jacobi-Verfahren

Die im vorigen Abschnitt vorgestellten Programme, die das Jacobi-Verfahren anwenden, wurden anhand von mehreren linearen Gleichungssystemen getestet. Für die genauere Untersuchung wurde ein Gleichungssystem ausgewählt, welches das Laufzeitverhalten der drei Programmversionen für die betrachten Systeme in etwa repräsentiert. Die Matrix *A* des betrachteten linearen Systems stellt dabei eine Abwandlung der *Hilbert-Matrix* dar, die wie folgt konstruiert wurde:

$$a_{ij} = \begin{cases} \dfrac{1}{i+j} & falls\ i \neq j \\ \displaystyle\sum_{j=1, j\neq i}^{n} a_{ij} + 1 & falls\ i = j \end{cases} \qquad (1 \leq i, j \leq n)$$

Die Komponenten des Rechte-Seite-Vektors b wurden jeweils auf eins festgesetzt, und für die Größen der Gleichungssysteme wurden $n = 50$, 250 und 500 ausgewählt.
Die Anzahl der Iterationsschritte, die bis zum Erreichen der Konvergenz benötigt werden, stimmt bei dem sequentiellen und dem synchronen Jacobi-Programm jeweils überein, da auch die gleiche Iterationssequenz erzeugt wird. Bei dem asynchronen Jacobi-ähnlichen Programm wird die Konvergenz für die hier betrachteten Systeme hingegen schon meist etwa 10 bis 15 Iterationsschritte früher erreicht. Dies ist mit dem im Vergleich zur synchronen Methode unterschiedlichen Aktualisierungsgrad der in die Berechnungen eingehenden Werte der Komponenten zu begründen.
In Abb. 18 sind die Laufzeiten der synchronen und asynchronen Versionen gegenübergestellt.

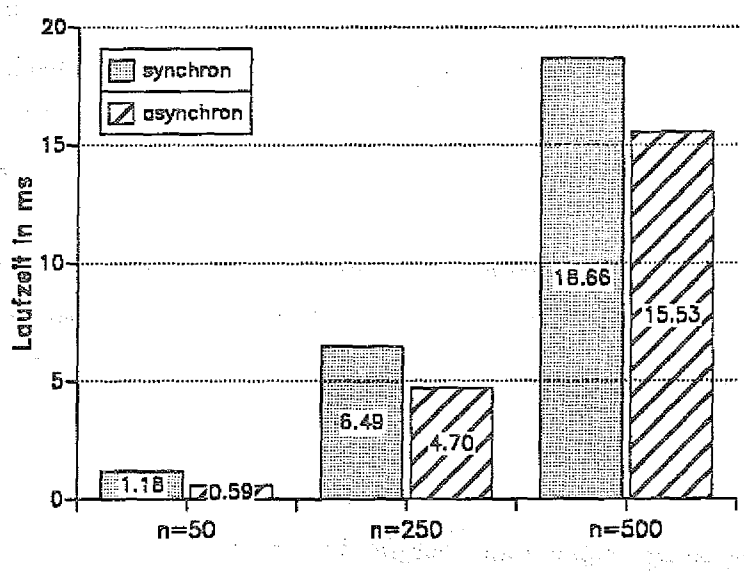


Abb. 18. Laufzeiten des synchronen und asynchronen Jacobi-Verfahrens

Man erkennt den im Vergleich zum synchronen Verfahren kürzeren Zeitverbrauch der asynchronen Version, der im wesentlichen durch die geringere Anzahl von den bis zur Konvergenz benötigten Iterationsschritten verursacht wird. Dabei beobachtet man einen

prozentual geringer werdenden Unterschied zwischen den Laufzeiten der beiden Versionen mit zunehmender Größe des linearen Systems. Zur Erklärung dazu sind zwei Ursachen zu nennen. Zum einen ist der Anteil an Overhead zur Rechenzeit bei der synchronen Version bei großen Gleichungssystemen kleiner, so daß die Laufzeit im Vergleich zu der asynchronen Version kürzer ist. Zum anderen ist die Einsparung der bis zur Konvergenz benötigten Iterationsschritte bei dem asynchronen Programm für kleine n etwas größer.

Bei der Betrachtung des Overheads, der für die Ausführung der benutzten Autotasking-Direktiven und für die Synchronisation der Tasks anfällt, ergeben sich ähnliche Ergebnisse wie bei der SOR-Methode. Allerdings kann man einen geringeren Anteil an Ausführungszeit für die Direktiven beim synchronen Jacobi-Verfahren erwarten als bei der synchronen SOR-Methode, da lediglich einmal in jedem Iterationsschritt Aufwand für die Direktive *DO ALL* anfällt.

In Abb. 19 sind die Zeitanteile für die Ausführung der Direktiven und für die Berechnungen bei sequentieller Ausführung der synchronen und asynchronen Version am Beispiel $n = 250$ gegenübergestellt. Bei den Zeiten kann hier der Einfluß, der durch schnellere Konvergenz bei der asynchronen im Vergleich zur synchronen Version entsteht, ausgeschlossen werden, da bei sequentieller Ausführung in beiden Programmen die gleiche Iterationsfolge erzeugt wird.

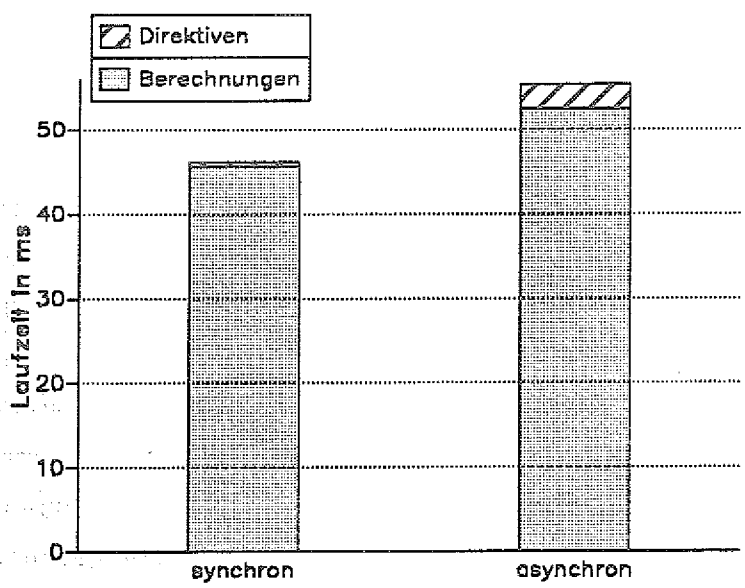


Abb. 19. Zeiten für Direktiven und Berechnungen bei sequentieller Ausführung

Der Zeitanteil für die reinen Berechnungen liegt bei der asynchronen Methode wiederum höher als bei der synchronen Methode, da zusätzlicher Rechenaufwand für die Modulo-Bildung und die Abfrage nach der Berechnung der n-ten Komponente anfällt.

Der Speedup des synchronen Jacobi-Verfahrens für die drei betrachteten Gleichungssysteme, der auf der CRAY Y-MP erzielt wurde, ist in Abb. 20 aufgezeichnet.

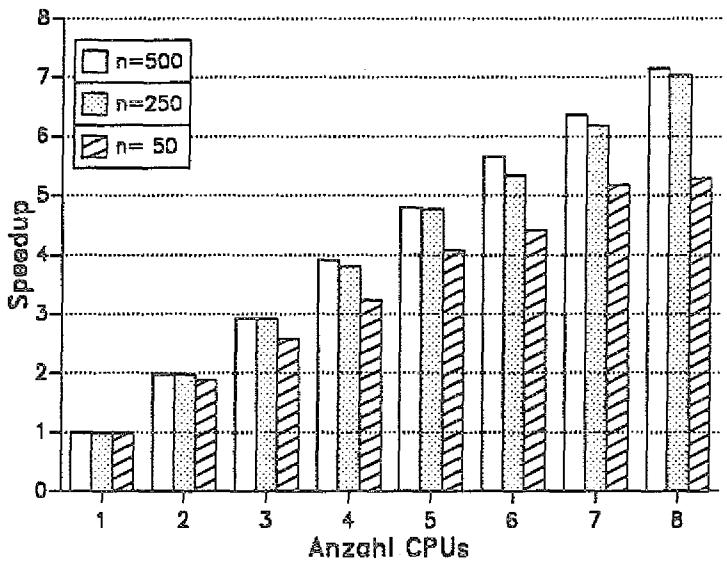


Abb. 20. Speedup des synchronen Jacobi-Verfahrens

Man erkennt eine Zunahme des Speedups mit wachsender Größe der Gleichungssysteme, die mit dem kleiner werdenden Verhältnis von Overhead zu Rechenzeit zu erklären ist. Wie auch schon bei der synchronen SOR-Methode stellt man eine Zunahme des Unterschieds zwischen dem erreichten und dem idealen Speedup mit wachsender Anzahl der an der Ausführung beteiligten Prozessoren fest. Dies wird insbesondere bei dem kleineren Gleichungssystem deutlich, in dessen Laufzeit sich eine Vergrößerung des Anteils an Synchronisations-Overhead wegen des geringeren Rechenaufwands in der parallelen Schleife deutlich bemerkbar macht. Für die Ausführung mit fünf Prozessoren ergeben sich relativ hohe Speedup-Werte, da die Anzahl der Schleifeniterationen in jeder parallelen Schleife ohne Rest durch fünf teilbar ist, und somit keine Wartezeiten entstehen. Die insgesamt geringeren Werte des Speedup sind eine Folge der nicht parallelisierten Vektor-Schleife zu Beginn einer jeden Iteration des synchronen Jacobi-Verfahrens. Während der Ausführung dieser Schleife arbeitet nur ein Prozessor, wodurch

sich die gemessene parallele Ausführungszeit nicht linear mit wachsender Anzahl von Prozessoren verringern kann.

Für das asynchrone Jacobi-ähnliche Verfahren wird im folgenden das Verhältnis von sequentieller und paralleler Ausführungszeit mit *Beschleunigungsfaktor* bezeichnet, da wegen der Unterschiede in der Anzahl der benötigten Iterationsschritte auch Werte, die größer als der ideale Speedup sind, auftreten können. In Abb. 21 ist dieser Beschleunigungsfaktor für die verschiedenen Größen der Gleichungssysteme aufgezeichnet.

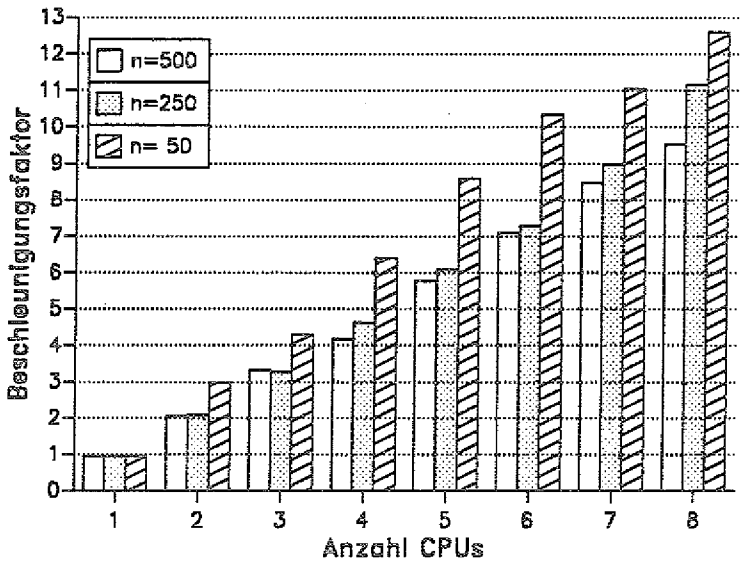


Abb. 21. Beschleunigungsfaktor des asynchronen Jacobi-ähnlichen Verfahrens

Wenn man von der Ausführung des Programms mit einem Prozessor absieht, wo der Einfluß der Iterationsschritte wegfällt, wird deutlich, daß der Beschleunigungsfaktor um so größer ist, je kleiner das betrachtete Gleichungssystem ist. Dieser Effekt läßt sich wiederum durch die unterschiedliche Anzahl an Iterationsschritten deuten. Hierbei kann man bei den beiden größeren Systemen eine etwa gleichmäßige Verringerung der benötigten Iterationsschritte mit wachsender Zahl von Prozessoren beobachten, wodurch der etwa lineare Anstieg des Beschleunigungsfaktors erklärt wird. Dieser Sachverhalt läßt sich aber so nicht auf andere Gleichungssysteme und andere Größen verallgemeinern, was das Beispiel $n = 50$ hier schon deutlich macht. Generell hängt die Anzahl der Iterationsschritte beim asynchronen Jacobi-ähnlichen Verfahren immer jeweils von dem betrachteten Gleichungssystem ab.

In Abb. 22 ist das Verhältnis der Ausführungszeit des sequentiellen Programms, welches dem sequentiell ausgeführten synchronen Programm entspricht, jeweils zu den beiden parallelen Programmversionen gebildet.

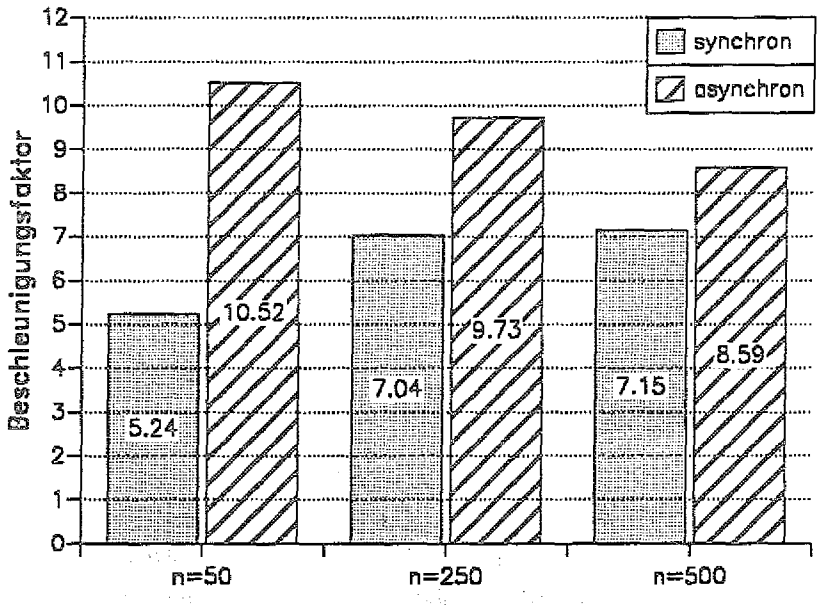


Abb. 22. Beschleunigungsfaktoren der parallelen Jacobi-Verfahren mit acht Prozessoren

Da die Ausführungszeit des sequentiellen Jacobi-Verfahrens geringer ist, als die des sequentiell ausgeführten asynchronen Verfahrens, ergeben sich hier geringere Beschleunigungsfaktoren für das asynchrone Programm als in Abb. 21.

5.7 Implementierung des Einschritt-Newton-SOR-Verfahrens

Da das Einschritt-Newton-SOR-Verfahren zur Lösung von nichtlinearen Gleichungssystemen benutzt wird, wurden bei der Implementierung für die Funktion f und die Jacobi-Matrix Df ein Feld F der Größe n und ein zweidimensionales Feld DF der Größe $n \times n$ eingesetzt. Die beiden Variablen enthalten in jedem Iterationsschritt t jeweils die Werte für die Jacobi-Matrix und die Funktion f auf der Grundlage der Werte des gerade aktuellen Lösungsvektors $x(t)$ und der vorherigen Iteration $x(t-1)$, um die Berechnungsvorschrift (4.12) des Verfahrens auszuführen. Andere im vorigen Kapitel vorgestellte Variablen und Konstanten behalten auch bei diesem Verfahren ihre Funktion.

Die für jeden Iterationsschritt t zu berechnende Funktionalmatrix $Df(x(t-1))$ wird in den im folgenden vorgestellten Programmen analytisch bestimmt, d.h. durch direkte Angabe der Berechnungsvorschrift im Programmcode.
In Abb. 23 ist der Aufbau des sequentiellen Programmkerns beschrieben.

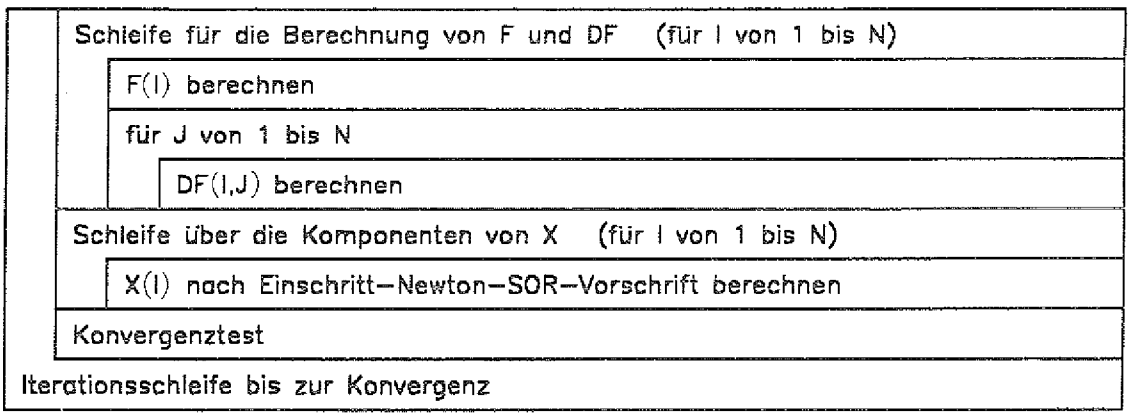


Abb. 23. Sequentielles Einschritt-Newton-SOR-Verfahren

Wiederum wurde das gleiche Konvergenzkriterum wie auch bei den vorher beschriebenen Verfahren angewendet.

5.7.1 Synchrones Einschritt-Newton-Verfahren

Bei der Parallelisierung des Einschritt-Newton-SOR-Verfahrens zeigt sich wiederum die Schwierigkeit, daß die Berechnungen der Komponenten von x nicht unabhängig voneinander sind, weil in die Berechnung der Funktionalmatrix auch Werte der gleichen Iteration von anderen Komponenten eingehen. Falls jedoch die Funktionalmatrix property A aufweist, ergeben sich auch in diesem Verfahren zwei disjunkte Indexmengen, in denen die Berechnungen der zugehörigen Komponenten von x nicht untereinander abhängen. Dies wird für die Parallelisierung des Verfahrens ausgenutzt, indem die Schleife für F und DF und die Schleife für die Berechnung der Komponenten von x zu einer einzigen Schleife zusammengefaßt werden. Dies ist möglich, da für die Berechnung einer Komponente x_i nur der Funktionswert der i-ten Funktion und die i-te Zeile der Jacobi-Matrix benötigt werden. Diese Schleife wird wiederum in zwei parallele Schleifen über die Indizes der beiden disjunkten Komponentenmengen geteilt. Die Indizes werden auf die gleiche Weise wie bei der SOR-Methode durch ein Feld $LIST$ angesprochen.

Nichtlineare Gleichungssysteme, deren Funktionalmatrizen property A besitzten, ergeben sich beispielsweise bei der Diskretisierung von nichtlinearen Randwertaufgaben. Das synchrone parallele Programm weist dann die in Abb. 24 verdeutlichte Struktur auf.

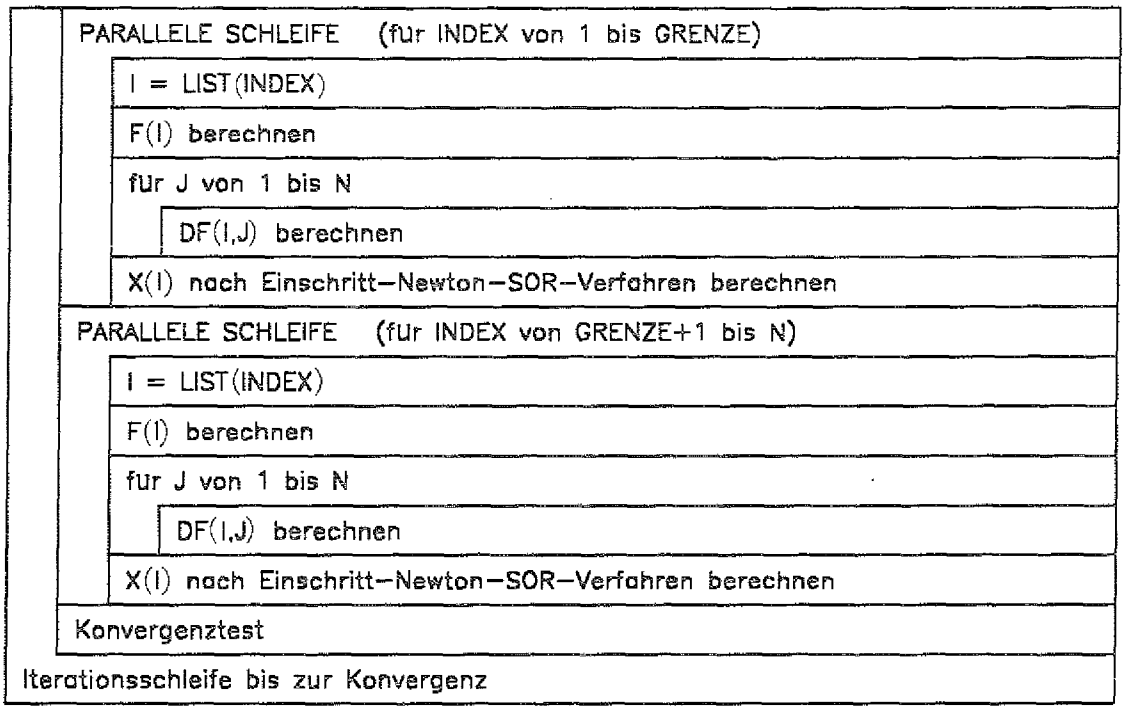


Abb. 24. Synchrones Einschritt-Newton-SOR-Verfahren

Die Umsetzung mit Autotasking erfolgte dann analog zur synchronen SOR-Methode.

5.7.2 Asynchrones Einschritt-Newton-SOR-Verfahren

Das asynchrone Programm wurde für das Einschritt-Newton-SOR-Verfahren wie bei den Verfahren zur Lösung von linearen Gleichungssytemen durch Umformung der Iterationsschleife in eine DO-Schleife und deren Parallelisieren mit Hilfe der Autotasking-Direktiven *CMICS DO ALL* und *CMICS SOFT EXIT* erhalten.

Da die Implementierung analog zu den anderen vorher beschriebenen Verfahren vorgenommen wurde, soll hier nur kurz der asynchrone Programmcode mit den verwendeten Autotasking-Direktiven vorgestellt werden.

```
C   Konvergenzflag initialisieren
      KONV = .TRUE.
C   Parallele Iterationsschleife
CMIC$ DO ALL SHARED(DF,LIST,X,F,XVOR,KONV,ITER) PRIVATE(I,J,SUM,INDEX)
      DO 1000 INDEX=0,OG*N
        I=LIST(MOD(INDEX,N)+1)
C        .
C        .  Funktionswert der i-ten Funktion F(I) berechnen
C        .
C   i-te Zeile der Funktionalmatrix berechnen
        DO 3 J=1,N
C        .
C        .  DF(I,J) berechnen
C        .
3       CONTINUE

        SUM=0
        DO 4 J=1,I-1
          SUM=SUM+DF(J,I)*(X(J)-XVOR(J))
4       CONTINUE
        XVOR(I)=X(I)
C   X(I) aktualisieren
        X(I) = X(I) - (OMEGA/DF(I,I))*(SUM+F(I))
        IF ((KONV) .AND. (ABS(X(I)-XVOR(I)) .GE. EPS)) THEN
         KONV = .FALSE.
CDIR$ SUPPRESS KONV
       ENDIF
C   Konvergenztest
       IF (I .NE. N) GOTO 1000
C   Abbruch der Iterationsschleife, falls konvergent
       IF (KONV) THEN
         ITER=INDEX/N
CMIC$ SOFT EXIT
         GOTO 1010
       ENDIF
C   Konvergenzflag für die nächste Iteration setzen
       KONV= .TRUE.
C   Ende der Iterationschleife
1000  CONTINUE
C   Sprungziel bei Konvergenz des Verfahrens
1010  CONTINUE
```

Asynchrones Einschritt-Newton-SOR-Verfahren

5.8 Leistungsmessungen zum Einschritt-Newton-SOR-Verfahren

Für das Einschritt-Newton-SOR-Verfahren wurden die Zeitmessungen für nichtlineare Gleichungssysteme durchgeführt, die aus den Diskretisierungen von Randwertproblemen entstehen und damit eine Funktionalmatrix mit property A aufweisen.
Als Beispiel dient das das Randwertproblem

$$-u_{xx} - u_{yy} = -e^{u} \qquad (0 < x, y < 1)$$

mit den Randbedingungen $u(x, y) = 0$. Für die Ordungen der Gleichungssysteme wurden die Größen $n = 100$, 400 und 1600 ausgewählt, da Gitter der Größen 10×10, 20×20 und 40×40 betrachtet wurden.
In Abb. 25 sind die Ausführungszeiten der parallelen synchronen und asynchronen Versionen zur Lösung der nichtlinearen Systeme dargestellt.

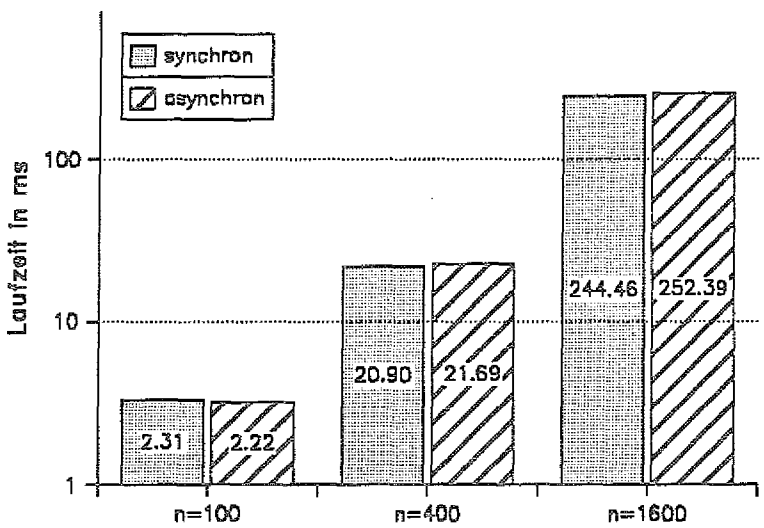


Abb. 25. Gegenüberstellung der Laufzeiten synchrones/asynchrones Einschritt-Newton-SOR-Verfahren

Man beobachtet geringe Unterschiede in den Laufzeiten der beiden parallelen Versionen, wobei für größere Systeme die asynchrone Version mehr Zeit zur Berechnung der Lösung benötigt als die synchrone Version. Diese schon bei den vorher untersuchten Verfahren beobachtete Tendenz ist auch hier mit dem Verhältnis von Rechenaufwand in den parallelen Schleifen zum Synchronisations-Overhead zu erklären.

Die Betrachtung des Overhead für die Direktiven in den beiden parallelen Programmen gestaltete sich in ähnlicher Weise wie bei der SOR-Methode, so daß keine neuen Erkenntnisse gewonnen werden konnten, und somit darauf hier verzichtet werden soll.

Der Speedup des synchronen Einschritt-Newton-SOR-Verfahrens weist für alle betrachteten Größen hohe Werte auf, wie man in Abb. 26 erkennen kann. Dies war nach den vorher gewonnenen Erkenntnissen zu erwarten, da in den parallen Schleifen des synchronen Einschritt-Newton-SOR-Verfahrens im Vergleich zu den vorherigen Methoden mehr Rechenaufwand geleistet werden muß. So muß jeweils eine Komponenete der Funktion f und eine Zeile der Funktionalmatrix für jede Komponente des Lösungsvektors x explizit berechnet werden, wodurch der Overhead keinen sehr starken Einfluß auf den Speedup hat.

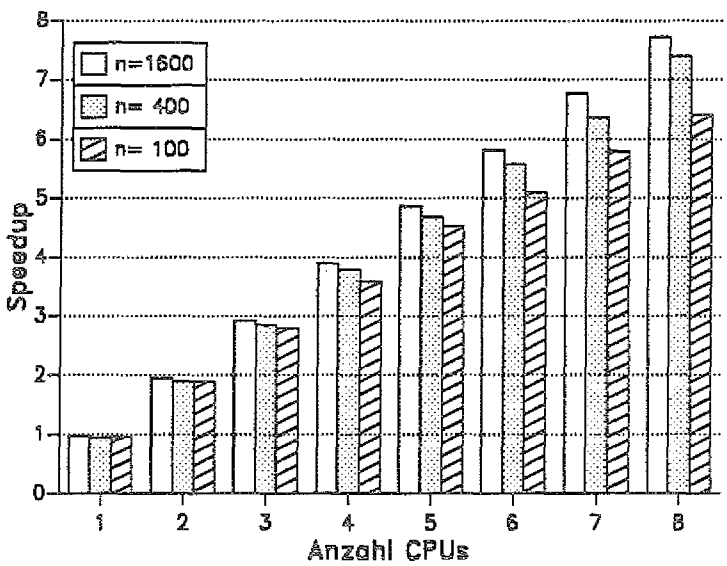


Abb. 26. Speedup des synchronen Einschritt-Newton-SOR-Verfahrens

Wenn beim Speedup der synchronen Version noch Unterschiede zwischen den verschiedenen Größen beobachtet werden können, die mit wachsender Zahl, der an der Ausführung beteiligten Prozessoren, zunehmen, so sind diese beim asynchronen Verfahren in Abb. 27 kaum mehr wahrnehmbar.

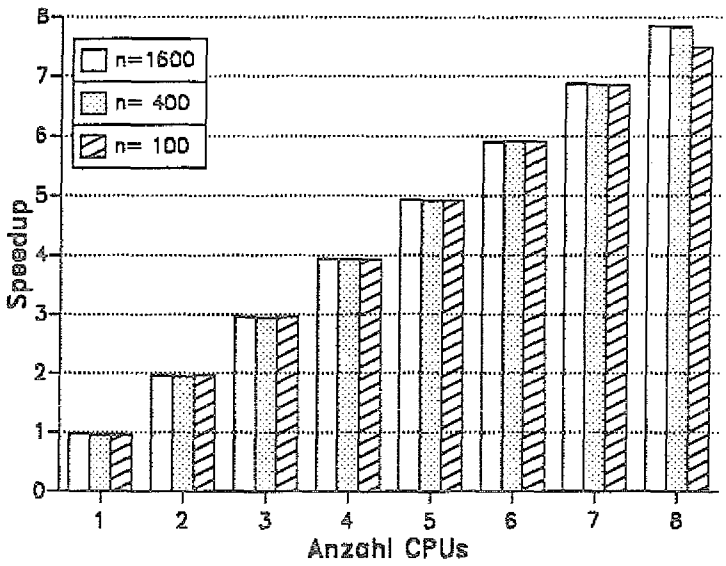


Abb. 27. Speedup des asynchronen Einschritt-Newton-SOR-Verfahrens

Der ohnehin schon geringe Synchronisations-Overhead der asynchronen Version wirkt sich hier also kaum aus.

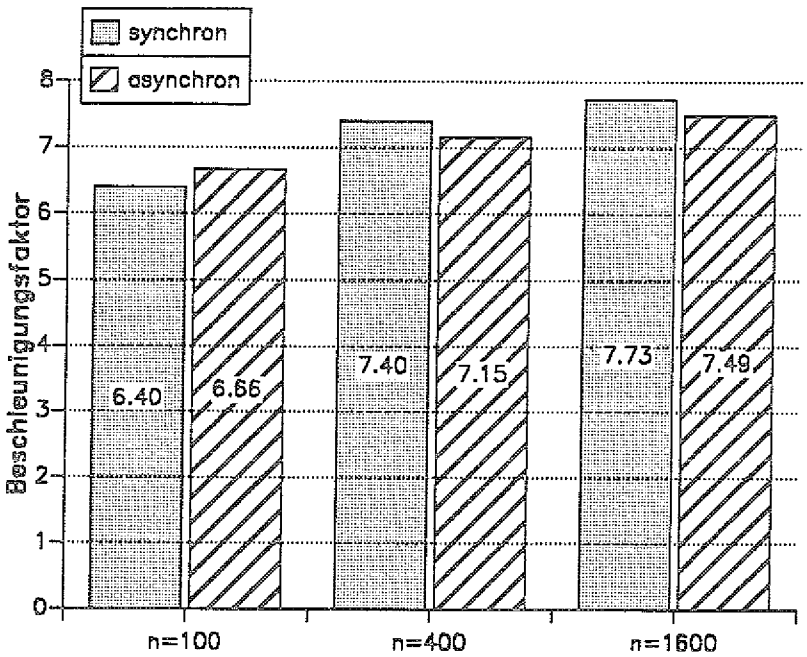
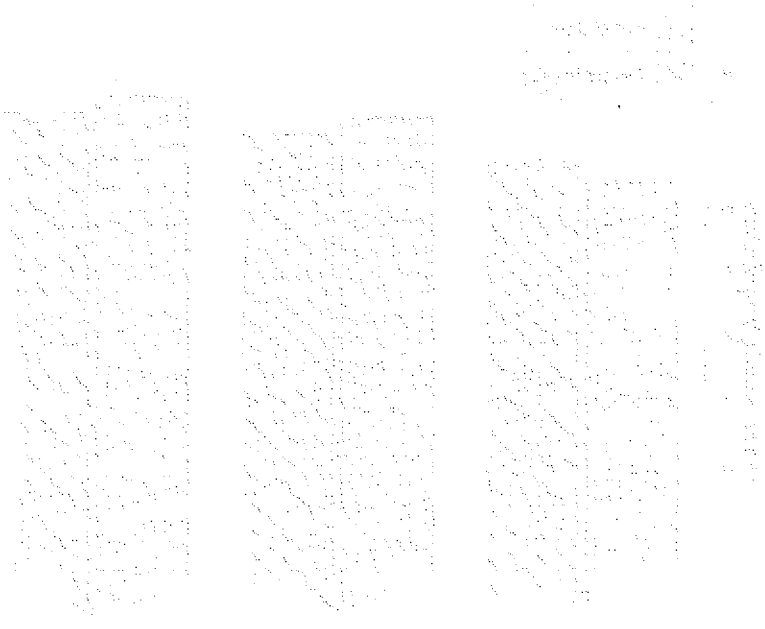


Abb. 28. Beschleunigungsfaktoren der parallelen Einschritt-Newton-SOR-Verfahren

Bei der Gegenüberstellung der Beschleunigungsfaktoren für die beiden parallelen Programme in Abb. 28 erkennt man für die beiden großen Systeme niedrigere Werte beim asynchronen Programm, die sich aber zwischen den Größen der Gleichungssysteme nicht sehr stark unterscheiden. Für das synchrone Verfahren ist die Zunahme des Beschleunigungsfaktors mit der Größe der betrachteten Systeme relativ groß.

Diese Beobachtungen stellen die typischen Merkmale der untersuchten synchronen und asynchronen Algorithmen gut heraus, deren Begründung in den Abschnitten über Leistungsmessungen dargelegt wurde.



6.0 Untersuchungen im Multiprogramming-Betrieb

Nachdem in den vorigen Abschnitten Untersuchungen der entwickelten Programme für eine dedizierte Umgebung durchgeführt wurden, soll in diesem Kapitel das Verhalten der parallelen Programmversionen im Multiprogramming-Betrieb kurz angerissen werden.

6.1 Verhalten der parallelen Programme im Multiprogramming-Betrieb

Im Multiprogramming-Betrieb eines CRAY-Rechners, wo mehrere Programme um die vorhandenen Betriebsmittel konkurrieren, ist es möglich, daß Tasks, die an einer Programmausführung beteiligt sind, unterbrochen werden. Diese Unterbrechungen, die durch den Entzug des Prozessors einer Task verursacht werden, können an beliebigen Stellen bei der Abarbeitung des Programmcodes erfolgen, selbst innerhalb einer FORTRAN-Anweisung. Während für die synchronen Programme dies, wegen der Synchronisierung aller Prozesse nach jedem Iterationsschritt, keinerlei Auswirkungen auf eine korrekte Ausführung hat, ergeben sich bei den asynchronen Versionen Schwierigkeiten.

Durch die Situation im Multiprogramming-Betrieb kann es zu einer Verletzung der in Abschnitt 3.4.2. vorgestellten Bedingungen für das asynchrone Algorithmenmodell bei den hier entwickelten asynchronen Programmversionen kommen. Insbesondere gegen Bedingung (b) kann verstoßen werden, in der verlangt wird, daß zu alte Werte im Laufe der Berechnungszeit irgendwann nicht mehr benutzt werden dürfen, und durch neuere Werte ersetzt werden müssen. Im Multiprogramming-Betrieb kann es durch Unterbrechungen der ausführenden Tasks geschehen, daß ein aktueller Wert einer Komponente von einem aus früheren Zeitpunkten stammenden Wert überschrieben wird.

Als Beispiel dazu kann man sich den Fall vorstellen, daß eine Task zu einem frühen Zeitpunkt eine Komponente x_i bearbeitet, aber vor der Aktualisierung der gemeinsamen Variablen X(i) unterbrochen wird. Die anderen Tasks führen ihre Berechnungen fort, und auch die i-te Komponente wird weiter aktualisiert. Erwacht nun aber die unterbrochene Task wieder, nachdem ein Konvergenztest positiv ausgefallen ist, so wird der aktuelle Wert von X(i) mit dem alten Wert überschrieben, so daß das Ergebnis nicht mehr korrekt ist. Dieser Fall zeigt sich bei der Betrachtung der Werte des Lösungsvektors nach der Programmausführung dadurch, daß einige wenige Werte einzelner Komponenten relativ stark von der korrekten Lösung abweichen.

Ein weiterer ungünstiger Fall im Multiprogramming-Betrieb kann auftreten, wenn der Konvergenztest auf der Grundlage nicht aller Komponenten positiv ausfällt. Falls Tasks suspendiert werden, die Komponenten bearbeiten, welche einen negativen Ausgang des Tests bewirkt hätten, kann es zu einem vorzeitigen Beenden der Berechnungen kommen.

Dieser Fall kann sich in der Regel nur gegen Ende der Berechnungen einstellen, wenn viele Werte der Komponenten das Konvergenzkriterium schon erfüllen. Man erkennt das Eintreffen dieses Falls daran, daß die Werte der meisten Komponenten nur in der Größenordnung der Abbruchkonstante von den korrekten Werten abweichen.

Von der Berücksichtigung dieser Fälle bei der Programmentwicklung der asynchronen Versionen wurde abgesehen, weil dies eine nicht unerhebliche Verlängerung der Ausführungszeit zur Folge gehabt hätte. So hätten beispielsweise zusätzliche IF-Abfragen in kritischen Abschnitten eingefügt werden müssen, da nur in kritischen Abschnitten garantiert werden kann, daß der Prozessor nicht entzogen wird.

Bemerkt werden soll an dieser Stelle, daß die hier beschriebenen Fälle nicht sehr häufig auftreten, und daß die Anzahl der korrekt ausgeführten Programmläufe der asynchronen Versionen im Multiprogramming-Betrieb die Anzahl der fehlerhaften Läufe bei weitem übertrifft.
In einer dedizierten Umgebung kann der Entzug eines Prozessors während der Ausführung eines parallelen Abschnittes ausgeschlossen werden.

6.2 Leistungsverhalten im Multiprogramming-Betrieb

Trotz der obengenannten möglicherweise fehlerhaften Programmausführungen der asynchronen Programme im Multiprogramming-Betrieb soll auch auf Zeitmessungen in dieser Umgebung kurz eingegangen werden.
Dabei wurden zu verschiedenen Zeiten, in denen also verschiedene Hintergrundlasten der CRAY Y-MP vorlagen, jeweils zehn Messungen der synchronen und asynchronen Programmversionen unmittelbar hintereinander vorgenommen, und die Durchschnittszeit gebildet.
Diese Durchschnittszeiten sind repräsentativ für die SOR-Methode mit $n = 1024$ in Abb. 29 dargestellt.
Bei Zeitmessungen im Multiprogramming-Betrieb gestaltet es sich allerdings als schwierig algorithmische Einflüsse auf die Ausführungszeit zu erkennen, da bei parallelen Programmen ein starker Einfluß durch die Anzahl der an der Ausführung beteiligten Prozessoren besteht.

Bei der Betrachtung von Abb. 29 stellt man fest, daß für die meisten Messungen die Laufzeit des synchronen Programms höher liegt als die des asynchronen Programms. Auch sind die Unterschiede zwischen einzelnen Messungen bei der synchronen Version ausgeprägter als bei der asynchronen Version.

Dies liegt zum einen an der Arbeitsweise des Betriebssystems bei der Zuteilung und beim Entzug von Tasks, denn vor allem nach der Beendigung eines parallelen Abschnitts im Programm können Tasks anderen Programmen zugeteilt werden. Da bei den asynchronen Programmen lediglich ein großer paralleler Abschnitt realisiert ist, während mehrere kürzere parallele Schleifen bei den synchronen Versionen vorkommen, besteht bei den synchronen Programmen eher die Möglichkeit des Prozessorentzugs als bei den asynchronen Programmversionen.

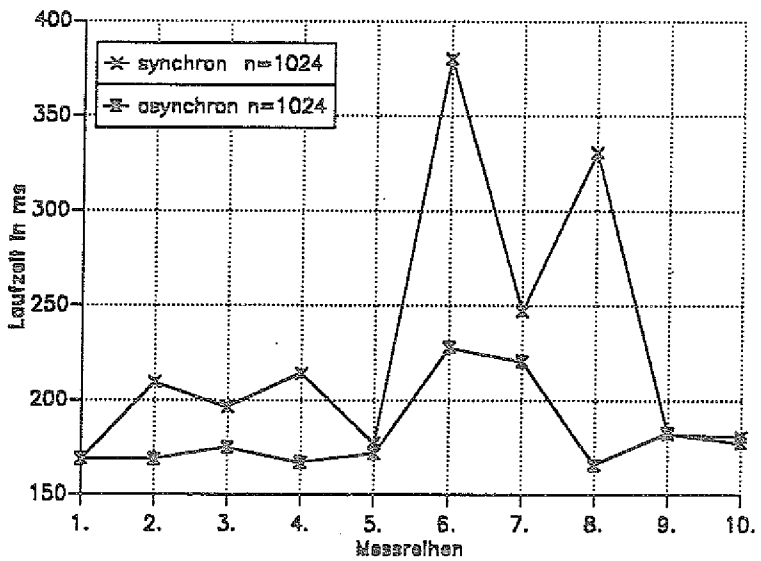


Abb. 29. Laufzeiten synchrones/asynchrones SOR im Multiprogramming-Betrieb

Zum anderen lassen die Werte in Abb. 29 aber auch den Schluß zu, daß bei der synchronen Programmversion durch die Synchronisation am Ende der parallelen Schleifen Wartezeiten eintreten, die sich verlängernd auf die Laufzeit auswirken. Die etwas geringeren Schwankungen bei der asynchronen SOR-Methode und die meist absolut niedrigeren Laufzeiten sind dann Folge der fehlenden Wartezeiten.

Diese bei dem größeren untersuchten Gleichungssystem für die SOR-Methode noch relativ gut zu beobachtende Tendenz prägt sich bei den beiden kleineren betrachteten Systemen nicht so aus, wie die Darstellung in Abb. 30 zeigt.

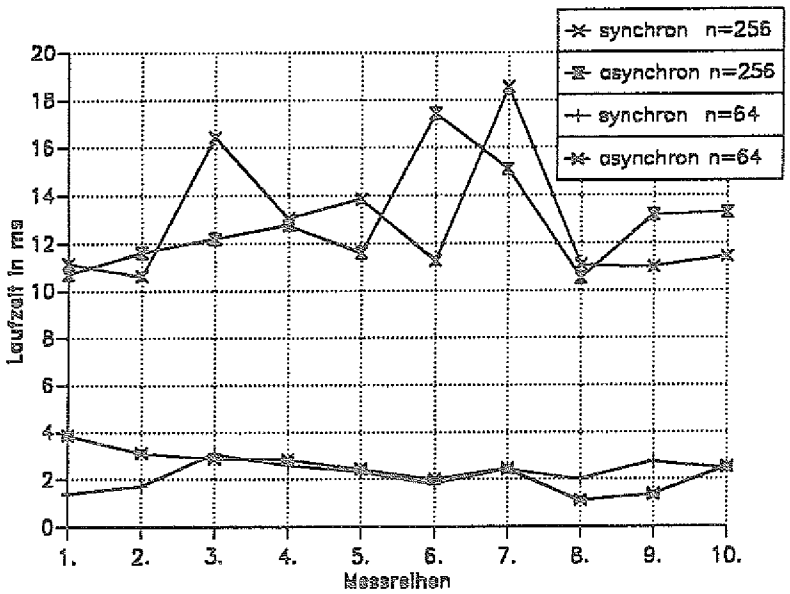


Abb. 30. Laufzeiten synchrones/asynchrones SOR im Multiprogramming-Betrieb

Hier wird wohl wegen den kürzeren Laufzeiten der Programme der Effekt der Wartezeiten von der Anzahl der Prozessoren stark überdeckt.

Zusammenfassend kann man sagen, daß die Laufzeiten der asynchronen Versionen durch das Fehlen der Synchronisation nach jedem Iterationsschritt im Multiprogramming-Betrieb meist geringer ausfallen als die der synchronen Versionen.

7.0 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurden synchrone und asynchrone Algorithmen anhand von Verfahren zur Lösung von linearen und nichtlinearen Gleichungssystemen auf CRAY-Rechnern untersucht.
Als Fazit dieser Untersuchungen wurde festgestellt, daß die asynchronen Verfahren für diese Anwendung auf CRAY-Mehrprozessorsystemen keine überragenden Vorteile gegenüber den synchronen Verfahren in Bezug auf das Laufzeitverhalten boten.
Eine Hauptursache dafür kann in den günstigen Eigenschaften gesehen werden, die das Autotasking-Konzept in Bezug auf Parallelisierung und Synchronisierung dem Anwender bietet. So ist in den Autotasking-Direktiven zur Parallelisierung von Schleifen jeweils eine implizite Synchronisation am Ende der Schleifen enthalten; so daß dem Anwender eine explizite Angabe zur Synchronisation von parallelen Prozessen erspart bleibt. Dies gestaltete sich als sehr vorteilhaft für die synchronen Programme, wohingegen für die asynchronen Versionen Zusatzaufwand an Programmiertechnik und Rechenzeit nötig war, um die implizite Synchronisation zu umgehen. Um den Rechenaufwand möglichst gering zu halten, wurden dabei Datenabhängigkeiten im parallelen Teil der asynchronen Programme in Kauf genommen, die im Multiprogramming-Betrieb zu Überschreibungen der aktuellen Werte des Lösungsvektors mit veralteten Werten führen können. Dies stellt für die Anwendung dieser Programmversionen einen großen Nachteil dar.

Da die Synchronisation bei der Verwendung von Autotasking bei CRAY-Rechnern mittels Semaphorregistern geschieht und daher sehr effizient realisiert ist, stellte sich der anfallende Synchronisations-Overhead für die synchronen Programmversionen als nicht sonderlich nachteilig auf die parallele Ausführungszeit heraus. Dadurch konnte bei den asynchronen Versionen die beabsichtigte Einsparung des Synchronisations-Overhead, außer bei kleinen Größen der Gleichungssysteme, nicht in einem großen Maße ausfallen. Bei anfallenden Wartezeiten für die synchronen Programme, entweder durch eine ungleiche Lastverteilung oder durch Einsatz im Multiprogramming-Betrieb verursacht, konnte jedoch für die asynchronen Programme ein günstigeres Laufzeitverhalten im Vergleich zu den synchronen Versionen beobachtet werden.
Für Verfahren zur Lösung von Gleichungssytemen, in denen eine Parallelisierung über die einzelnen Komponenten des Lösungsvektors vorgenommen werden kann, ist wegen der dadurch relativ kleinen Prozesse eine gleichmäßige Lastverteilung meist leicht realisierbar, so daß Wartezeiten nicht übermäßig stark auftreten. Es sind jedoch durchaus Anwendungen vorstellbar, bei denen eine gut balancierte Lastverteilung nicht möglich ist, so daß das Fehlen von Wartezeiten bei asynchronen Algorithmen auch bei gut zu

synchronisierenden Rechnersystemen im Hinblick auf die Ausführungszeit stärker zum Tragen kommt.

Ein interessanter Aspekt der asynchronen Algorithmen konnte bei der Abwandlung des Jacobi-Verfahrens zu einem asynchronen Jacobi-ähnlichen Verfahren beobachtet werden. Durch die asynchrone Vorgehensweise ergaben sich andere Berechnungsreihenfolgen und Aktualisierungsgrade für die Werte der Komponenten des Lösungsvektors, wodurch Iterationsschritte eingespart werden konnten und somit in einer kürzeren Laufzeit resultierten.

Die erhaltenen Ergebnisse für CRAY-Systeme mit einer relativ kleinen Anzahl von hochleistungsfähigen Prozessoren, die über den gemeinsamen Speicher miteinander gekoppelt sind, wecken das Interesse für eine Untersuchung von asynchronen Algorithmen in verteilten Rechnersystemen mit einer hohen Anzahl von Prozessoren und verteiltem Speicher. In einer solchen Umgebung ist die Synchronisation meist an mehr Programmieraufwand und Zeitverbrauch durch Kommunikation geknüpft, so daß vermutlich die asynchronen Algorithmen ein höheres Leistungspotential bergen.

8.0 Literaturverzeichnis

[B&E,82] Barlow, R.H. / Evans, D.J.
Parallel Algorithms for the Iterative Solution to Linear Systems
The Computer Journal 25 (1982), 56-60

[Bau,78] Baudet, G.M.
Asynchronous Iterative Methods for Multiprocessors
Journal of the ACM 25 (1978), 226-244

[BBK,80] Baudet, G.M. / Brent, R.P. / Kung, H.T.
Parallel Execution of a Sequence of Tasks on an Asynchronous Multiprocessor
Australian Computer Journal 12 (1980), 105-112

[BEK,90] Bönniger, T. / Esser, R. / Krekel, D.
Cray Y-MP, NEC SX-3 und Siemens VP-S
Vergleichende Darstellung von Höchsleistungsrechnern
Wirtschaftsinformatik 3 (1990), 273-286

[Ben,85] Ben-Ari, M.
Grundlagen der Parallelprogrammierung
Carl Hanser Verlag, München, 1985

[C&M,69] Chazan, D. / Miranker, W.
Chaotic Relaxation
Linear Algebra and Its Applications 2 (1969), 199-222

[CRA,87] CRAY
CRAY X-MP Multitasking Programmer's Reference Manual
CRAY Research Inc., Revision D, SN-0222, 1987

[CRA,88a] CRAY
CRAY System Programmer's Reference Manual
CRAY Research Inc., CSM-0400-000, 1988

[CRA,88b] CRAY
Autotasking Users Guide
CRAY Research Inc., SN-2088, 1988

[Fox&&,88] Fox, G.C. / Johnson, M. / Lyzenga, G. / Otto, S. / Salman, J. / Walker, D.
Solving problems on concurrent processors
Prentice-Hall, Englewood Cliffs, 1988

[HKN,89] Hoßfeld, F. / Knecht, R. / Nagel, W.E.
Multitasking: Experience with applications on a CRAY X-MP
Parallel Computing 12 (1989), 259-283

[H&J,88] Hockney, R.W. / Jesshope, C.R.
Parallel computers 2
Adam Hilger Ltd, Bristol, 1988

[Hoß,83] Hoßfeld, F.
Parallele Algorithmen
Springer Verlag, Berlin, 1983

[H&W,84] Hwang, K. / Briggs, F.A.
Computer Architecture and Parallel Processing
McGraw-Hill, New York, 1984

[Kun,76] Kung, H.T.
Synchronized and Asynchronous Parallel Algorithms for Multiprocessors
in: Algorithms and Complexity (J.F. Traub ed.), Academic Press, New York, 1976

[Lie,88] Liegmann, A.
Die Strategie des Microtasking als Mittel zur Beschleunigung von Programmen auf dem Vektorrechner CRAY X-MP
Spezielle Berichte der KFA Jülich, Jül-Spez-435, 1988

[Nag,88] Nagel, W.E.
Using multiple CPUs for problem solving: Experiences in multitasking on the CRAY X-MP/48
Parallel Computing 8 (1988), 223-230

[Nag,90] Nagel, W.E.
Exploiting autotasking on a CRAY Y-MP: An improved software interface to multitasking
Parallel Computing 13 (1990), 225-233

[Ort,70] Ortega, J.M. / Rheinboldt, W.C.
Iterative Solution of Nonlinear Equations in Several Variables
Academic Press, New York, 1970

[Qui,87] Quinn, M.J.
Algorithmenbau und Parallelcomputer
McGraw-Hill, Hamburg, 1987

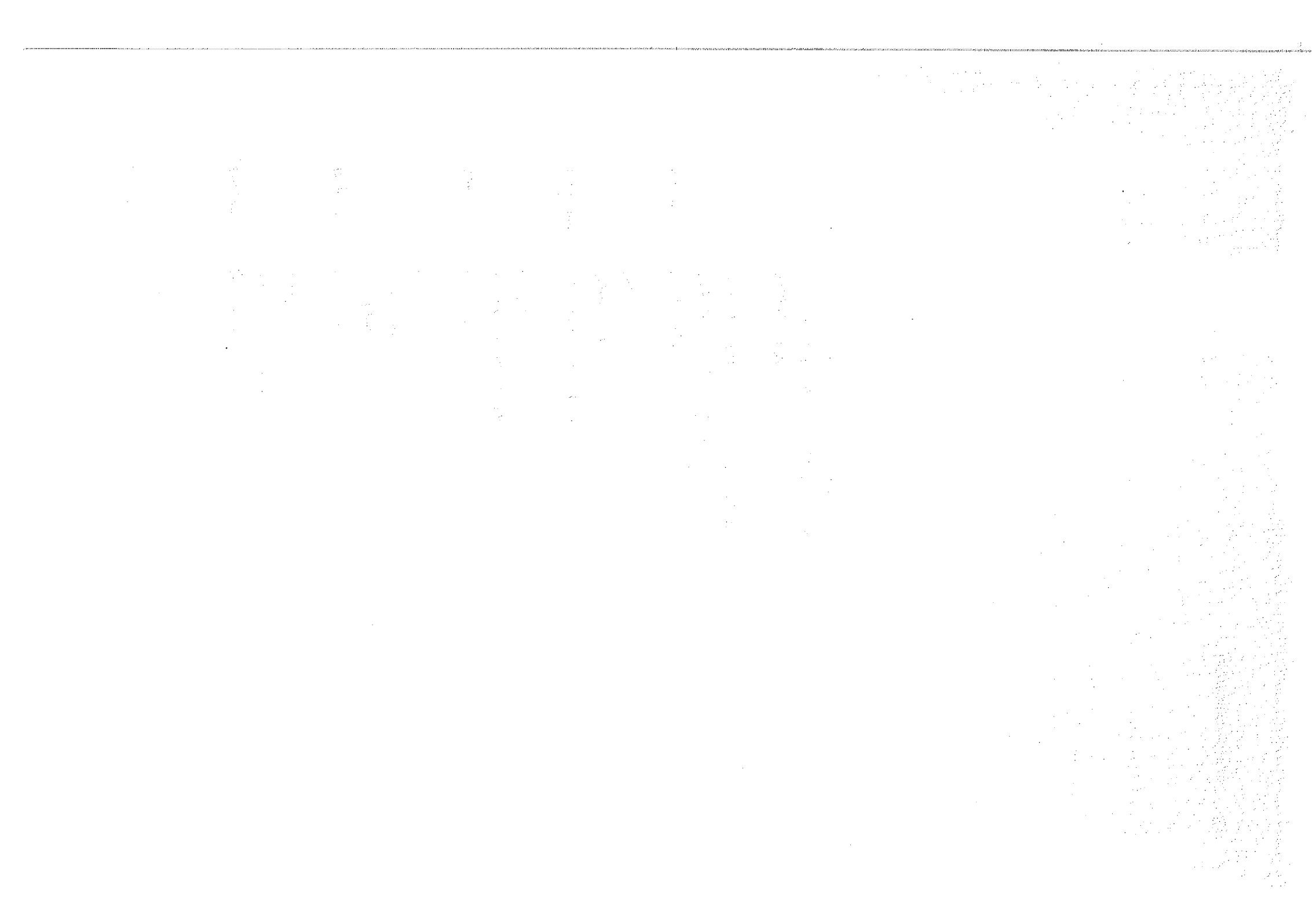
[Reg,89] Reger, H.
Ein Vergleich der Multitasking-Implementierungen auf CRAY X-MP und IBM 3090
Spezielle Berichte der KFA Jülich, Jül-Spez-542, 1989

[Sto,76] Stoer, J.
Einführung in die Numerische Mathematik I
Springer Verlag, Berlin, 1976

[T&B,89] Tsitsiklis, J.N. / Bertsekas, D.P.
Parallel and Distributed Computation: Numerical Methods
Prentice-Hall, New York, 1989

[Var,62] Varga, R.S.
Matrix Iterative Analysis
Prentice-Hall, Englewood Cliffs, 1962

[You,71] Young, D.M.
Iterative Solution of Large Linear Systems
Academic Press, London, 1971



Die vorliegende Arbeit wurde als Diplomarbeit am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen angefertigt.

Für die Möglichkeit, diese Arbeit am Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich zu erstellen, danke ich Herrn Prof. Dr. F. Hoßfeld sowie seinen Mitarbeitern. Insbesondere Herrn Dr. P. Weidner gilt mein herzlicher Dank für die Betreuung und hilfreiche Unterstützung.

FORSCHUNGSZENTRUM JÜLICH GmbH

KFA

Jül-2426
Januar 1991
ISSN 0366-0885