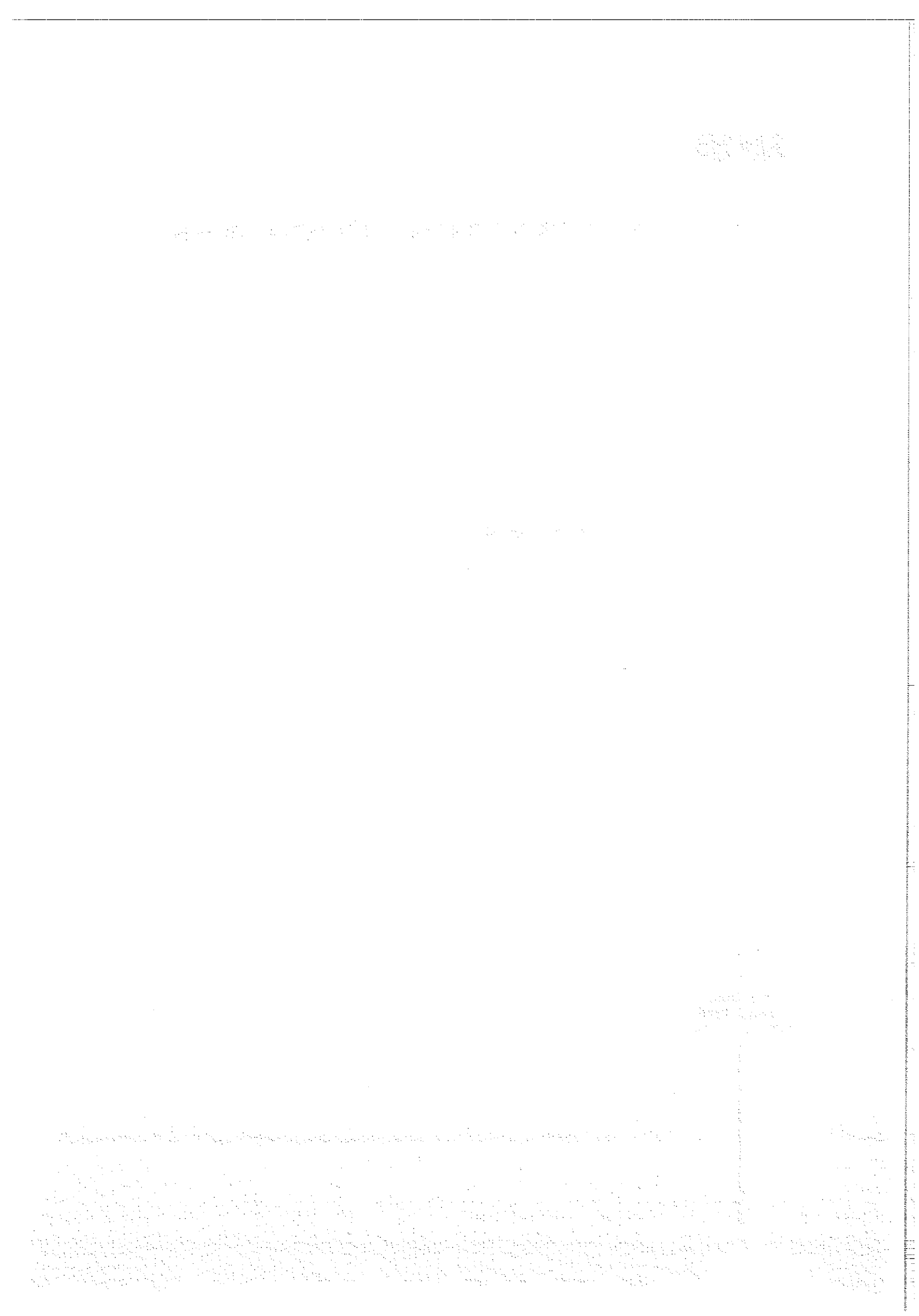
FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik

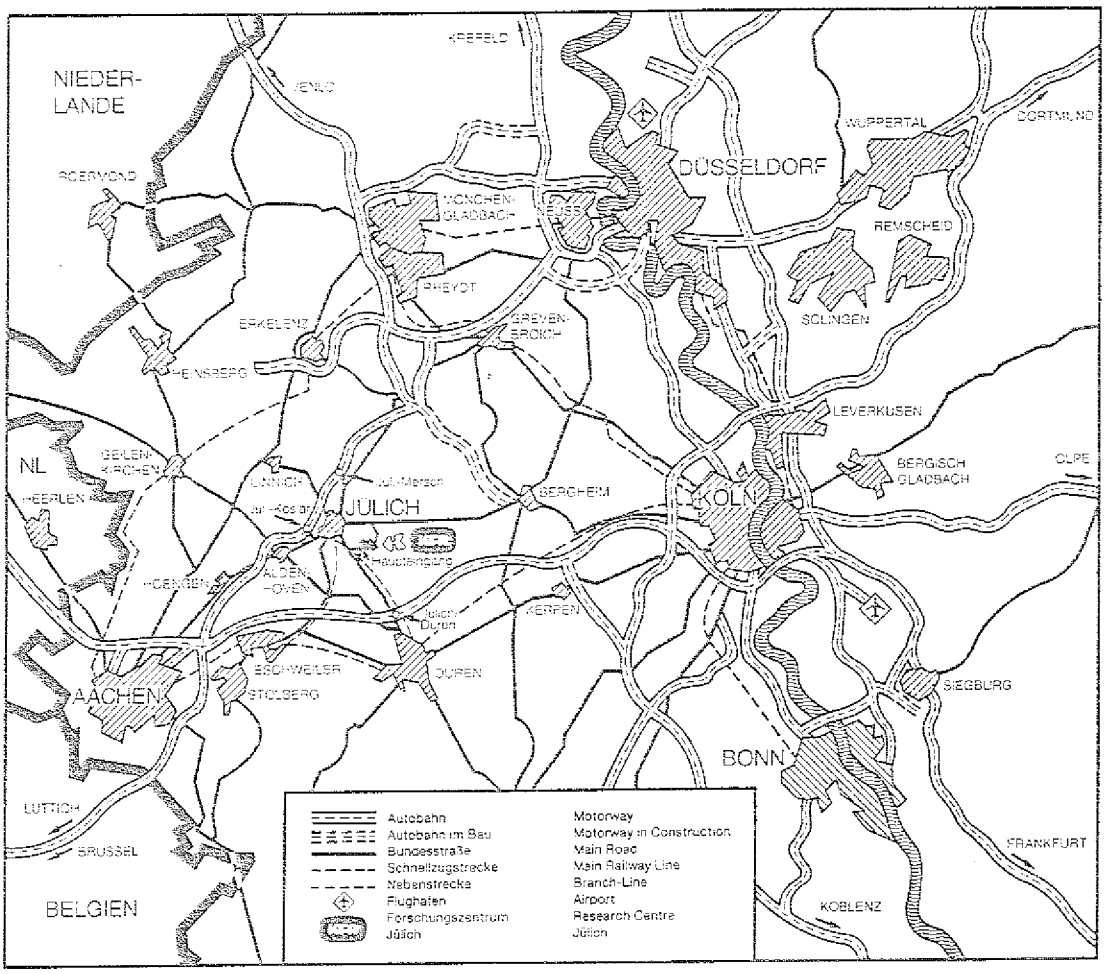
Lineare Algebra Software für SUPRENUM

von

I. Gutheil
W. Rönsch
H. Strauß

Jül-2345
Januar 1990
ISSN 0366-0885





Als Manuskript gedruckt

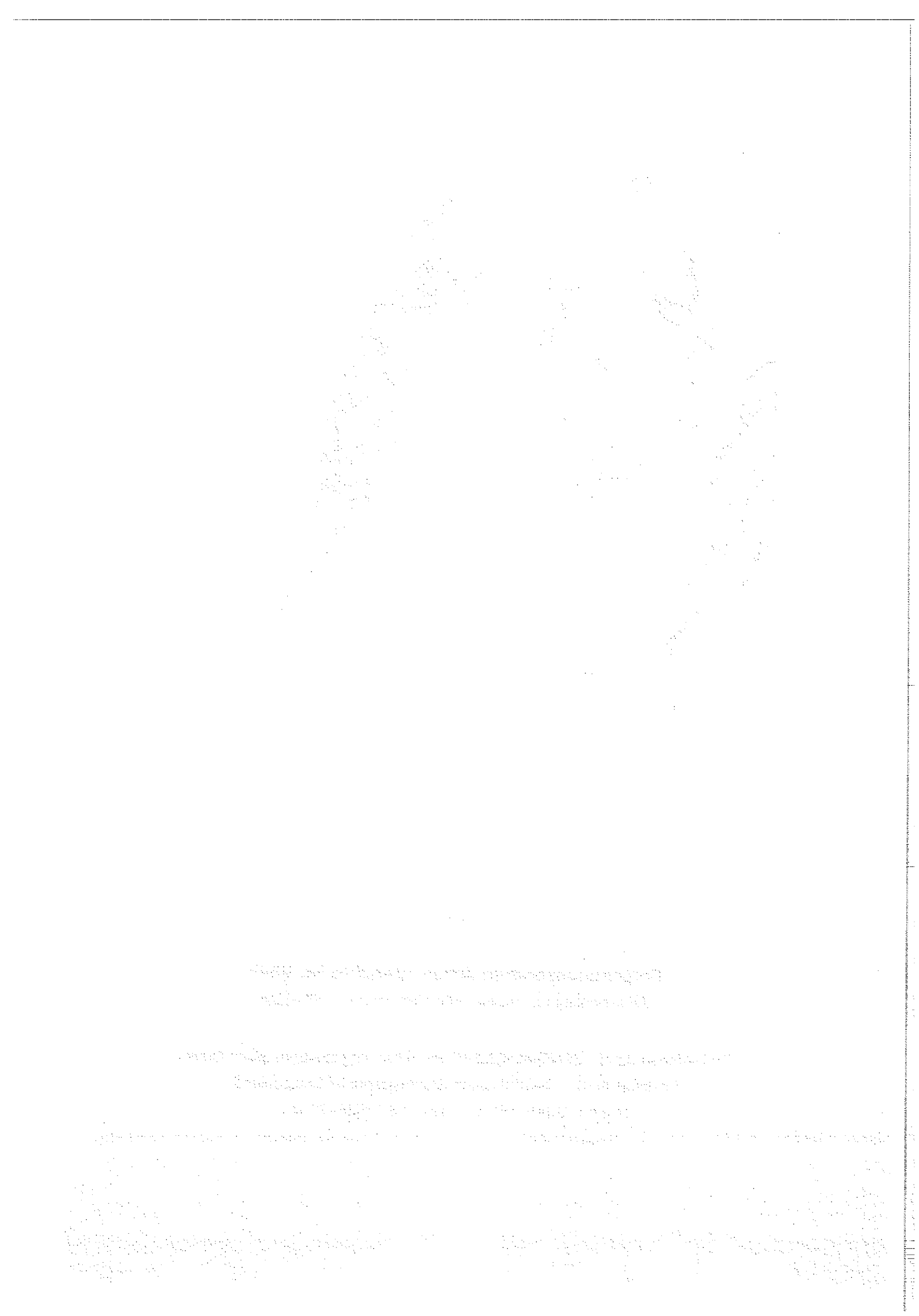
Forschungszentrum Jülich: Berichte Nr. 2345

Zentralinstitut für Angewandte Mathematik Jül-2345

Zu beziehen durch: ZENTRALBIBLIOTHEK Forschungszentrum Jülich GmbH

Postfach 1913 · D-5170 Jülich (Bundesrepublik Deutschland)

Telefon: 02461/61-0 · Telex: 833556-70 kf d



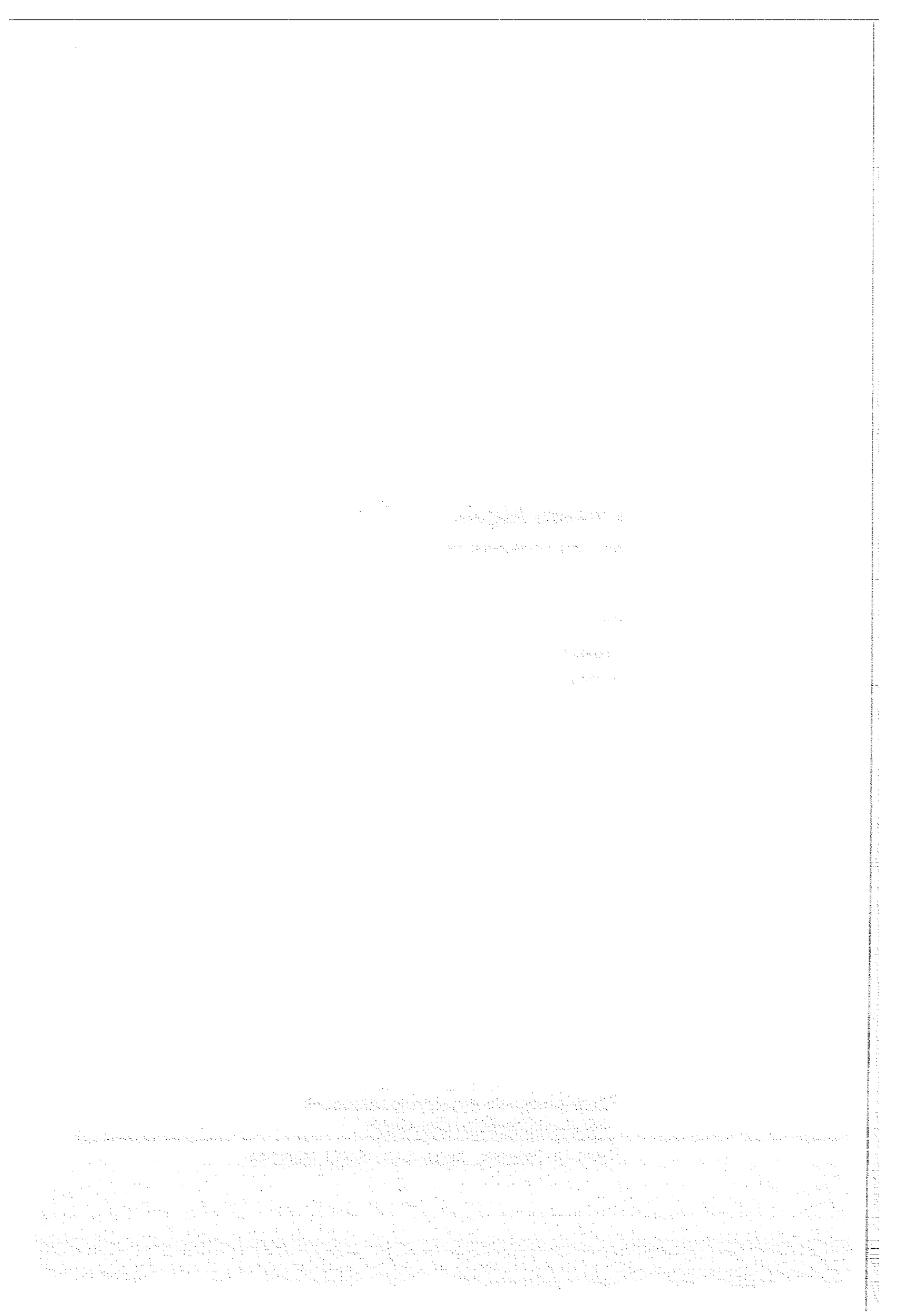
Lineare Algebra Software für SUPRENUM

von

I. Gutheil[1)]
W. Rönsch[2)]
H. Strauß[2)]

[1)] Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich GmbH

[2)] Prof. Dr. Feilmeier, Junker & Co. GmbH, München



Zusammenfassung

In einem Verbundprojekt mit 14 Partnern aus Großforschung, Industrie und Universitäten entsteht ein Parallelrechner, der SUPRENUM Rechner (SUPerREchner für NUMerische Anwendungen). Der Rechner besteht aus einem Hochleistungskern mit bis zu 256 Knoten und drei Steuerrechnern für Betrieb, Wartung und Programmierung.
Neben Hardware und Systemsoftware werden umfangreiche Anwendungssoftwarepakete für die Bereiche Lineare Algebra und numerische Verfahren für Strömungsdynamik, Meteorologie, elektromagnetische Felder etc. entwickelt. Diese werden als Programmbibliotheken zur Ausstattung des Rechners gehören.

Die KFA-Jülich ist in diesem Rahmen für das Programmpaket zur Lösung des reellen Eigenwertproblems und für ein Paket für Design und Optimierung von Teilchenbeschleunigern verantwortlich ([&Guth.],[&Buch.]). Außerdem wurde im Bereich Lineare Algebra ein Spezialprogramm zur Lösung linearer Gleichungssysteme mit Bandmatrix bei der KFA-Jülich entwickelt ([&Gut2.]).

Das gesamte Lineare Algebra Paket, SLAP (SUPRENUM Linear Algebra Package), entsteht in Zusammenarbeit zwischen der Prof. Dr. Feilmeier, Junker & Co. GmbH und der KFA-Jülich GmbH.

Im vorliegenden Dokument werden die Grundlagen (Kapitel 1-4) von SLAP dargestellt und einige Routinen exemplarisch detailliert beschrieben (Kapitel 5-7). Eine vollständige Beschreibung aller in SLAP implementierten Routinen wird im SLAP User's Guide, der endgültigen Dokumentation von SLAP, veröffentlicht.

Inhaltsverzeichnis

Verzeichnis der Abbildungen

Verzeichnis der Tabellen

Einleitung

In einem Verbundprojekt mit 14 Partnern aus Großforschung, Industrie und Universitäten entsteht ein Parallelrechner, der SUPRENUM Rechner (SUPerREchner für NUMerische Anwendungen). Der Rechner besteht aus einem Hochleistungskern mit bis zu 256 Knoten und drei Steuerrechnern für Betrieb, Wartung und Programmierung.
Neben Hardware und Systemsoftware werden umfangreiche Anwendungssoftwarepakete für die Bereiche Lineare Algebra und numerische Verfahren für Strömungsdynamik, Meteorologie, elektromagnetische Felder etc. entwickelt. Diese werden als Programmbibliotheken zur Ausstattung des Rechners gehören.

Die KFA-Jülich ist in diesem Rahmen für das Programmpaket zur Lösung des reellen Eigenwertproblems und für ein Paket für Design und Optimierung von Teilchenbeschleunigern verantwortlich ([22],[6]). Außerdem wurde im Bereich Lineare Algebra ein Spezialprogramm zur Lösung linearer Gleichungssysteme mit Bandmatrix bei der KFA-Jülich entwickelt ([21]).

Das gesamte Lineare Algebra Paket, SLAP (SUPRENUM Linear Algebra Package), entsteht in Zusammenarbeit zwischen der Prof. Dr. Feilmeier, Junker & Co. GmbH und der KFA-Jülich GmbH.

Im folgenden werden die Grundlagen (Kapitel 1-4) von SLAP dargestellt und einige Routinen exemplarisch detailliert beschrieben (Kapitel 5-7). Eine vollständige Beschreibung aller in SLAP implementierten Routinen wird im SLAP User's Guide, der endgültigen Dokumentation von SLAP, veröffentlicht.

1.0 Das SUPRENUM-Projekt

SUPRENUM ist ein vom BMFT gefördertes Verbundprojekt mit dem Ziel, einen marktreifen Parallelrechner der höchsten Leistungsklasse zu entwickeln.

Als Architektur wurde ein Parallelrechner mit verteiltem Speicher gewählt. Die einzelnen Prozessoren sind dabei leistungsfähige Vektorrechner, die durch ein zweistufiges Bussystem verbunden sind. Die Kommunikation zwischen den Prozessoren geschieht durch Botschaftenaustausch.

Das Projekt wird seit 1985 gefördert, und die geplante Laufzeit beträgt ca. vier Jahre.
Bis Ende 1989 sollen mehrere Entwicklungssysteme mit jeweils bis zu 16 Prozessoren fertiggestellt sein und an einzelne Projektpartner ausgeliefert werden. Der endgültige Prototyp mit 256 Prozessoren wird im Dezember 1989 vorgestellt.
Für die Entwicklung der Software stehen verschiedene Simulatoren auf CRAY und IBM-Großrechnern sowie auf SUN-Workstations zur Verfügung. Außerdem werden im sogenannten Preprototyping Programm die bereits fertiggestellten Hardwarekomponenten und die Betriebs- und Anwendungssoftware laufend getestet.

1.1 Organisation

Die beteiligten Partner aus Industrie, Hochschule und Großforschung arbeiten an drei wesentlichen Bereichen des Projekts:
Zum Bereich Hardware/System tragen vor allem die Abteilung FIRST der GMD Berlin, die Firmen Krupp Atlas Elektronik und Stollmann sowie die Hochschulen Braunschweig und Erlangen bei.
Im Bereich Sprachentwicklung arbeiten die GMD St. Augustin und Karlsruhe und die Hochschulen Bonn und Darmstadt.
Anwendungsprogramme werden bei den Firmen Dornier, Siemens (Bereich KWU) und Prof. Dr. Feilmeier, Junker & Co., bei der DLR Braunschweig, der GMD St. Augustin, der KFA, der KfK und an den Hochschulen Düsseldorf und Erlangen entwickelt.

Die Projektleitung und Organisation sowie der Vertrieb des Rechners liegen bei der im Januar 1986 gegründeten SUPRENUM Gesellschaft für numerische Superrechner mbH.

1.2 Die Hardware

Der Rechner besteht aus einem Hochleistungskern, dem eigentlichen Parallelrechner, und drei Steuerrechnern, je einem für Betrieb, Wartung und Programmierung.

Die drei Steuerrechner sind vom Typ MPR2300. Dies ist eine Entwicklung der Firma Krupp Atlas Elektronik auf der Basis des Mikroprozessors MC68020. Sie verfügen über periphere Schnittstellen für Ein- und Ausgabegeräte, über Plattenanschlüsse sowie über Netzwerkanschlüsse. Zusätzlich verfügt der MPR2300 über ein Hochleistungs-Graphik-Subsystem.

Der Hochleistungskern setzt sich aus 256 Prozessoren zusammen, den sogenannten Knoten. Jeder dieser Knoten besteht aus einem 32-Bit Mikroprozessor MC68020 mit

Gleitpunktkoprozessor 68881. Weiter verfügt jeder Knoten über einen Vektorkoprozessor WTL2264/65 sowie 8 Mbyte privaten Speicher. Unter Verwendung von Chaining erreicht damit jeder Knoten eine Maximalleistung von 20 Mflops.

Die Knoten sind in 16 Clustern zu je 16 Prozessoren angeordnet. Innerhalb eines Clusters sind die Knoten durch den sogenannten Clusterbus verbunden, der eine Übertragungsleistung von 40 Mbyte/s/Bus/Knotenpaar erreicht.
Jeder Cluster verfügt über eine Clusterplatte, einen Plattenanschlußknoten hierfür, einen Clusterdiagnoseknoten, sowie zwei Kommunikationsknoten mit je einem SUPRENUMBUS-Anschluß.

Die Cluster sind in einer 4×4-Matrix angeordnet, in der die Zeilen und die Spalten jeweils durch den sogenannten SUPRENUMBUS verbunden sind. Der SUPRENUMBUS wird auf der Basis des Taxichips realisiert und wird eine Übertragungsleistung von 125 Mbit/s erreichen.

1.3 Die Software

Im Gegensatz zu den meisten Konkurrenzprodukten wird der SUPRENUM-Parallelrechner mit einem umfangreichen Softwarepaket ausgestattet. Neben der Betriebssoftware für Steuerrechner und Hochleistungskern wird auch ein reichhaltiges Anwendungssoftwarepaket zusammen mit dem System ausgeliefert werden.

Auf den Steuerrechnern wird das Betriebssystem UNIX eingesetzt. Die Prozesse im Hochleistungskern sind dort als UNIX-Prozesse realisiert. Nur dieses Betriebssystem ist für den Benutzer sichtbar.
Auf dem Hochleistungskern läuft das im Rahmen des Projekts entwickelte Knotenbetriebssystem PEACE (Process Execution And Communication Environment), das insbesondere für die Kommunikation zwischen den Prozessen und die Verwaltung der lokalen Speicher und der Clusterplatten sorgt. Dieses Betriebssystem arbeitet vor dem Benutzer verborgen auf jedem Knoten.

Für die Programmierung steht ein Compiler für eine höhere Programmiersprache zur Verfügung, nämlich für FORTRAN. Diese Sprache wurden um Konstrukte für die Kommunikation und die Prozeßerzeugung erweitert.
Zusätzlich werden Entwicklungsumgebungen geschaffen, die durch Programmierwerkzeuge, „intelligente" Editoren und graphische Hilfsmittel dem Benutzer das Arbeiten mit einem hochparallelen System wesentlich erleichtern.

Im Bereich Anwendungssoftware werden neben der Bibliothek für Probleme aus der Linearen Algebra, SLAP, weitere Programmpakete entwickelt. Zum einen entstehen Basisroutinen zur Lösung elliptischer Differentialgleichungen, zum anderen Pakete zur Lösung spezieller Probleme z.B. aus der Strömungsdynamik (Navier-Stokes- und Euler-Gleichungen), für die Berechnung elektromagnetischer Felder und für Reaktorberechnungen. Für die meisten dieser Programme werden Mehrgitterverfahren verwendet, da diese für die Struktur des Rechners besonders gut geeignet sind. Alle Programme werden speziell an die Architektur des SUPRENUM-Rechners angepaßt, so daß eine möglichst effiziente Nutzung sichergestellt ist.

2.0 SUPRENUM aus der Sicht des Benutzers

2.1 Die abstrakte SUPRENUM Maschine

Der Benutzer ist nicht gezwungen, sich die in 1.2 beschriebene Hardware Struktur von SUPRENUM zu vergegenwärtigen, für ihn ist vielmehr die abstrakte SUPRENUM Maschine relevant. Diese beruht auf einem Prozeßmodell, bei dem verschiedene Prozesse unabhängig voneinander existieren und keinen gemeinsamen Speicher besitzen. Die Kommunikation zwischen Prozessen geschieht ausschließlich über expliziten Datenaustausch (message passing). Hierfür gibt es spezielle SEND und RECEIVE Befehle in den Programmiersprachen.

Die Prozeßstrukturen können unabhängig von der realen Hardware definiert werden, da grundsätzlich Nachrichtenaustausch zwischen beliebigen Prozessoren möglich ist. Außerdem wird es die Möglichkeit geben, mehrere Prozesse auf demselben Prozessor zu starten und ablaufen zu lassen.
Die Prozeßstruktur ist dynamisch: Prozesse können während der Laufzeit neue Prozesse kreieren, und sie können selbst terminieren; sie können jedoch nicht andere Prozesse terminieren.
Die Prozesse können asynchron kommunizieren, d.h. der sendende Prozeß kann weiterarbeiten, sobald die Nachricht abgeschickt ist. Er ist nicht so lange blockiert, bis der Empfängerprozeß die Nachricht empfangen hat.

2.2 Aufbau eines SUPRENUM FORTRAN Programmes

Ein SUPRENUM FORTRAN Programm besteht aus zwei wesentlichen Teilen.
Um eine Berechnung zu starten, ist genau eine **initiale Programmeinheit**, auch initialer Prozeß genannt, nötig. Diese ist ein FORTRAN Hauptprogramm, gekennzeichnet durch ein "INITIAL TASK PROGRAM name" Statement, mit dazugehörigen Unterprogrammen und läuft auf dem Steuerrechner. Nur in der initialen Programmeinheit ist Ein- und Ausgabe von und zur Peripherie möglich. Der initiale Prozeß muß durch Senden die Eingabedaten den Prozessen im Hochleistungskern bekannt machen und am Ende die Ergebnisse empfangen, um sie auszugeben.
Für die eigentliche Berechnung im Hochleistungskern gibt es mindestens eine **Task Programmeinheit**. Die Task Programmeinheit ist ebenfalls ein FORTRAN Hauptprogramm mit eventuellen Unterprogrammen, das durch ein "TASK PROGRAM name" Statement gekennzeichnet ist.
Zur Initialisierung eines neuen Prozesses enthält die initiale oder eine Task Programmeinheit folgende SUPRENUM FORTRAN spezifische Anweisungen:

```
TASKID proc                 /* Variablentyp
TASK EXTERNAL name          /* EXTERNAL Anweisung
proc=NEWTASK(name,i)        /* INTRINSIC FUNCTION
```

Hierdurch wird ein Prozeß mit der Prozeßkennung proc auf dem Prozessor mit Identifikationsnummer i gestartet, der den Code aus "TASK PROGRAM name" ausführt.
Im allgemeinen werden mehrere Prozesse mit demselben Programmcode gestartet, die diesen mit unterschiedlichen Daten ausführen.

Statt der expliziten Angabe einer Prozessornummer, also der benutzergesteuerten Zuordnung von Prozeß zu Prozessor, ist auch halbautomatische und vollautomatische Zuordnung möglich.
Bei der halbautomatischen Zuordnung kann der Benutzer angeben, wo der neu gestartete Prozeß in Relation zum ihn startenden Prozeß laufen soll. Es gibt die Möglichkeiten, daß er auf demselben Prozessor, auf einem beliebigen anderen, in demselben Cluster oder in einem anderen Cluster gestartet und abgearbeitet wird.
Bei der vollautomatischen Zuordnung überläßt es der Benutzer ganz dem Betriebssystem, auf welchem Prozessor der Prozeß ablaufen soll.

Zusätzlich gibt es Bibliotheksroutinen, die eine vom Benutzer gewünschte „Struktur" von Prozessen erzeugen. Mit diesen Routinen werden die Prozesse erzeugt und auf Prozessoren verteilt; zusätzlich bekommen sie Informationen über die Prozesse mitgeteilt, mit denen sie später kommunizieren sollen. Ein Beispiel ist die Prozedur "NEWRING", die eine gewünschte Anzahl von Prozessen kreiert und jedem Prozeß die Adresse seines linken und rechten Nachbarn mitteilt, so daß der Benutzer die Kommunikation so gestalten kann, als wären die Prozesse auf einem Ring von Prozessoren angeordnet.

3.0 Umfang des Lineare Algebra Pakets

3.1 Bereich Basisalgorithmen (BLAS)

Im Bereich der Basisalgorithmen sind parallele Versionen eines Teils der BLAS 2-Module und aller BLAS 3-Module realisiert worden. Die BLAS 2-Module umfassen verschiedene Varianten von Matrix-Vektor-Operationen (siehe 5.1.1), die BLAS 3-Module dagegen verschiedene Arten von Matrix-Matrix-Operationen (siehe 5.1.2).

Zum Bereich der realisierten parallelen Basisroutinen zählen auch Module zur parallelen Berechnung des Skalarproduktes zweier Vektoren und der Komponentensumme eines verteilten Vektors.

Alle Module sind so implementiert, daß sie sowohl auf einer Task Programmeinheit als auch auf der initialen Programmeinheit (siehe 2.2) aufgerufen werden könnnen und daß die Anzahl der an der Berechnung beteiligten Prozesse nur der Beschränkung unterliegt, daß sie mindestens eins ist. Alle Module sind in einfacher und doppelter Genauigkeit verfügbar, aber nicht in komplexer Arithmetik.

3.2 Bereich LINPACK

Es sind parallele Versionen der wichtigsten LINPACK-Routinen [11] implementiert worden. Dabei handelt es sich um Programme zur Lösung linearer Gleichungssysteme mit dichtbesetzten Matrizen.

Für allgemeine Matrizen gibt es Routinen zur Faktorisierung gemäß LU-Zerlegung und (im quadratischen Fall) zur anschließenden Lösung der linearen Gleichungssysteme mit Dreiecksmatrizen mittels Vorwärts- und Rückwärtseinsetzen.

Für symmetrische positiv-definite Matrizen sind Routinen zur Faktorisierung gemäß der Cholesky-Zerlegung und zur anschließenden Lösung der entstandenen linearen Gleichungssysteme mit Dreiecksmatrizen verfügbar.

Bei beiden Lösungsroutinen (LU- als auch Cholesky-Zerlegung) kann die rechte Seite der zu lösenden Gleichungssysteme aus einem oder mehreren Spaltenvektoren bestehen.

Für quadratische bzw. positiv-definite Matrizen sind Routinen vorhanden, die auf der Grundlage der Ergebnisse der LU- bzw. der Cholesky-Zerlegung sowohl die Inverse der Koeffizientenmatrix als auch die Determinante bestimmen. Alle mit der LU-Zerlegung im Zusammenhang stehenden Routinen gibt es sowohl in einer Ausführung ohne Pivotisierung als auch in einer mit teilweiser Pivotisierung. (Die Routinen ohne Pivotsuche können unter Umständen numerisch instabile Ergebnisse liefern. Falls bekannt ist, daß die Koeffizientenmatrix eine stabile Zerlegung ohne Pivotisierung erlaubt, bieten die Routinen ohne Pivotisierung den Vorteil erheblich verkürzter Laufzeiten.)

Der Bereich LINPACK umfaßt weiterhin Routinen zur QR-Zerlegung dichtbesetzter allgemeiner Matrizen sowie zur Lösung eines oder mehrerer linearer Ausgleichsprobleme.

Eine Spezialroutine zur Lösung linearer Gleichungssysteme mit diagonal-dominanter Bandmatrix wird zusätzlich angeboten. Hierbei sind Zerlegungs- und Lösungsteil nicht in zwei verschiedenen Routinen realisiert.
Die Bandmatrix muß in kompakter Form und in zeilenorientierter Aufteilung eingegeben werden. Diese Routine erfordert außerdem mindestens drei Prozesse und eine Gesamtbandbreite, die kleiner ist als die Anzahl der Zeilen der Bandmatrix, die jeder Prozeß bei der Aufteilung der Matrix bekommt.

Routinen für die Behandlung hermitescher und symmetrisch indefiniter Matrizen sowie von Matrizen in spezieller Speicherungsart, z.B. von symmetrischen Matrizen, deren Elemente spaltenweise in einem eindimensionalen Feld fortlaufend abgespeichert sind (wie es in LINPACK vorgesehen ist), wurden nicht implementiert.

Soweit bekannt wurden die Namen der parallelisierten Routinen und deren Aufruflisten an den neuen LAPACK-Standard [10] angepaßt.

Die hier beschriebenen Routinen mit Ausnahme der Spezialroutine für diagonaldominante Bandmatrizen (nur einfache Genauigkeit) gibt es sowohl in einfacher als auch in doppelter Genauigkeit.

3.3 Bereich EISPACK

Für einige der im EISPACK Guide [32] beschriebenen Verfahren für reelle symmetrische Matrizen sind parallele Versionen entwickelt worden. Es gibt Routinen zur Reduktion vollbesetzter reeller symmetrischer Matrizen auf Tridiagonalgestalt, Routinen zur Bestimmung der Eigenwerte und optional auch der Eigenvektoren symmetrischer Tridiagonalmatrizen sowie zur Rücktransformation der Eigenvektoren der Tridiagonalmatrizen auf die der ursprünglichen Matrix.

Vollbesetzte symmetrische Matrizen müssen komplett gegeben sein, also sowohl obere als auch untere Hälfte der Matrix sind besetzt, bei Bandmatrizen genügt ein zweidimensionales Feld, das die von Null verschiedenen Diagonalen enthält. Bei Tridiagonalmatrizen müssen zwei eindimensionale Felder mit der Diagonale und der Nebendiagonale gegeben sein.

Für die Reduktionsroutinen gibt es sowohl eine Version, die eine nach der Blockaufteilung verteilte Matrix voraussetzt als auch eine für block-zyklische und zyklische Verteilung der Matrix. Für jede dieser Reduktionsroutinen gibt es die entsprechende Routine zur Rücktransformation der Eigenvektoren.

Zur Berechnung der Eigenwerte und Eigenvektoren symmetrischer Tridiagonalmatrizen stehen zwei verschiedene Implementationen des Bisektionsverfahrens mit inverser Iteration und eine Implementation des "Divide-and-Conquer"-Verfahrens nach Cuppen zur Verfügung (siehe auch „Das reelle symmetrische Eigenwertproblem").

Mit Ausnahme der Routine, die das "Divide-and-Conquer"-Verfahren nach Cuppen verwendet, und die nur in doppelter Genauigkeit implementiert ist, stehen alle Routinen aus dem Bereich des symmetrischen Eigenwertproblems nur in einfacher Genauigkeit zur Verfügung. Dies soll jedoch in Kürze geändert werden, so daß dann auch alle Routinen

aus dem Bereich EISPACK in einfacher und doppelter Genauigkeit vorhanden sein werden. Im Bereich EISPACK gibt es keine Verfahren für komplexe Matrizen.

3.4 Bereich Datenverteilungsroutinen

Der Bereich der Datenverteilungsroutinen umfaßt Routinen zur Kommunikation zwischen initialer Programmeinheit und Task Programmeinheit bzw. der Task Programmeinheiten untereinander.

Bei der Verteilung allgemeiner Matrizen von der initialen Programmeinheit auf die Task Programmeinheiten wurden die Block-, die zyklische und die block-zyklische Aufteilung (siehe 4.3.3) sowohl in Spalten- als auch in Zeilenrichtung berücksichtigt. Entsprechend stehen Sammelroutinen zur Verfügung, die die aufgeteilten Matrizen auf der initialen Programmeinheit wieder zusammenführen.

Für die Kommunikation der Task Programmeinheiten untereinander gibt es Routinen, die allgemeine Matrizen oder Dreiecksmatrizen von einer Blockaufteilung in Spaltenrichtung in eine in Zeilenrichtung und umgekehrt überführen. Außerdem stehen Routinen zur Verfügung, die allgemeine Matrizen von einer spalten- bzw. zeilenorientierten Blockaufteilung in eine block-zyklische Aufteilung mit konstanter Blockbreite entlang Spalten bzw. Zeilen transformieren.

Es ist geplant, alle Datenverteilungsroutinen außer in einfacher auch in doppelter Genauigkeit verfügbar zu machen.

Einen Sonderfall im Bereich der Datenverteilungsroutinen stellt die Routine zur Synchronisation von parallel arbeitenden Task Programmeinheiten dar. Diese Routine ist notwendig zur Vermeidung von möglichen Botschaftsverwechslungen im Falle des mehrmaligen Aufrufs ein und derselben SLAP-Routine in Folge auf einer Task Programmeinheit.

4.0 Allgemeine Realisierungsstrategien

4.1 Algorithmen

Die implementierten Routinen beruhen nach Möglichkeit auf bekannten Algorithmen, wie sie z.B. in [36] beschrieben sind. Deshalb ist es nicht notwendig, numerische Eigenschaften der Algorithmen neu zu beweisen.
Wo es sinnvoll erschien, wurden statt der ursprüglichen Algorithmen aus [36] Blockversionen implementiert. Hierunter versteht man Verfahren, bei denen z.B. eine Zerlegung jeweils für einen kleinen Block, bestehend aus einigen Spalten, durchgeführt wird und danach erst auf den Rest der Matrix angewandt wird. Bei den ursprünglichen Algorithmen wird dagegen gleich nach der Bearbeitung einer Spalte auch der Rest der Matrix verändert. Durch diese Technik wird eine gröbere Granularität erreicht, die sowohl für die Vektorprozessoren Vorteile bringt als auch den Kommunikationsaufwand der implementierten Algorithmen senkt.

Auf Knotenebene werden serielle BLAS 3- und BLAS 2- Module in Assembler Code zur Verfügung stehen, die von den Lineare Algebra Routinen in der Task Programmeinheit aufgerufen werden. Dadurch wird die Granularität der Routinen gröber. Dies geschieht in Anlehnung an das Projekt LAPACK [10].

4.2 Aufrufkonzept

Die Lineare Algebra Routinen wurden als Programmbibliothek ähnlich LINPACK und EISPACK so implementiert, daß sie auf Task Programmebene aufgerufen werden können. Die Bibliotheksroutinen der Linearen Algebra gehen also grundsätzlich davon aus, daß die Daten bereits verteilt vorliegen.
Das Verteilen der Daten am Anfang einer Berechnung geschieht dabei mit Hilfe von Datenverteilungsroutinen, die in der initialen Programmeinheit, und durch Datenempfangsroutinen, die vor Beginn der Berechnungen in der Task Programmeinheit aufgerufen werden müssen.
Zusätzlich gibt es Rücksende- und Sammelroutinen, die die Ergebnisse der Rechnung an die initiale Programmeinheit zurücksenden bzw. dort sammeln, von wo sie auf Ausgabegeräte geschickt werden können (siehe auch 3.4).

Außerdem wird von dem oben bereits erwähnten Fall (s. 2.2) ausgegangen, daß der Benutzer, der die Bibliotheksroutinen aufrufen will, nur einen Task Programm Code schreiben muß, der von mehreren Prozessen mit unterschiedlichen Daten ausgeführt wird.
Die Aufrufe der Routinen im Task Programm sollen dabei möglichst kompatibel zu denen der LINPACK und EISPACK Routinen sein. Dies ist natürlich nicht ganz zu erreichen, da zusätzliche Parameter nötig sind, die die Anzahl und Identifikationen der übrigen beteiligten Prozesse sowie die Position des aktuellen Prozesses in der Gesamtheit angeben. Ein Prozeß muß z.B. aus den Übergabeparametern erkennen, ob er die ersten, ausschließlich mittlere oder die letzten Spalten der Matrix in seinem Speicher hat und daher bearbeiten muß, und welchen Prozessen er während der Berechnung Daten schicken muß.

4.2.1 Vorteile dieses Konzepts

Für den Benutzer sollte die Verwendung der Bibliothek damit so einfach wie möglich sein. Er muß selbst keine Kommunikation explizit programmieren, er muß beim Aufruf der Bibliotheksroutinen lediglich einige zusätzliche Übergabeparameter angeben. Da diese auch mit Hilfe von Bibliotheksroutinen erzeugt werden, sind Inhalt und Bedeutung dieser Variablen für den Benutzer nur dann wichtig, wenn er zusätzliche eigene verteilte Routinen schreiben will, die auf den verteilten Daten operieren.

Die verschiedenen Routinen können nacheinander auf Task Programmebene aufgerufen werden, ohne dazwischen die initiale Programmeinheit einzuschalten. Damit können die Daten auf den Knoten des Hochleistungskerns verteilt bleiben, ohne daß Kommunikation mit dem Steuerrechner erfolgen muß.

Bei einem alternativen Konzept, bei dem der Benutzer nur eine initiale Programmeinheit schreibt, in der er Bibliotheksroutinen aufruft, müßten bei jedem Aufruf im initialen Prozeß die Daten zuerst vom Steuerrechner zu den Knoten geschickt und am Ende der Prozedur wieder an den Steuerrechner zurückgesendet werden. Dies würde erhebliche zusätzliche Kommunikationszeit erfordern, da alle Prozessoren mit dem Steuerrechner kommunizieren müßten.

4.2.2 Nachteile dieses Konzepts

Bei dieser Art des Vorgehens ist es notwendig, eine Standardaufteilung der Matrizen und Vektoren auf die Prozesse festzulegen, die zu Beginn und Ende jedes Unterprogrammes gleich ist. Leider sind die verschiedenen Aufteilungsstrategien für die verschiedenen Problemstellungen unterschiedlich gut geeignet. Eine für die eine Aufgabenstellung hervorragend geeignete Aufteilung erweist sich unter Umständen für die nächste Aufgabe als besonders ungeschickt und umgekehrt. Andererseits kann eine ungünstige Aufteilung der Matrizen wiederum zu einer so schlechten Lastverteilung führen, daß die Umverteilung von Matrizen zu Beginn und Ende einer Routine angeraten sein könnte. Daher wird es auch Routinen geben, die verteilte Daten umverteilen.

4.3 Möglichkeiten für die Wahl einer Standardaufteilung

Aus den in [28] und [29] beschriebenen und im Kapitel „Das quadratische lineare Gleichungssystem" noch einmal aufgegriffenen Gründen sollen nur spaltenorientierte Aufteilungen betrachtet werden. Hierbei ergeben sich im wesentlichen drei Möglichkeiten, die im folgenden näher erläutert werden sollen.
Dabei sei n die Anzahl der Spalten einer zu verteilenden Matrix, p die Anzahl der Prozesse, auf die die Matrix zu verteilen ist und $\lceil n/p \rceil$ sei die größte ganze Zahl kleiner oder gleich n/p.

4.3.1 Die Blockaufteilung

Bei der Blockaufteilung bekommt jeder Prozeß einen zusammenhängenden Block von $k = n/p$ Spalten der Matrix. Falls p kein Teiler von n ist, sind die Blöcke der ersten

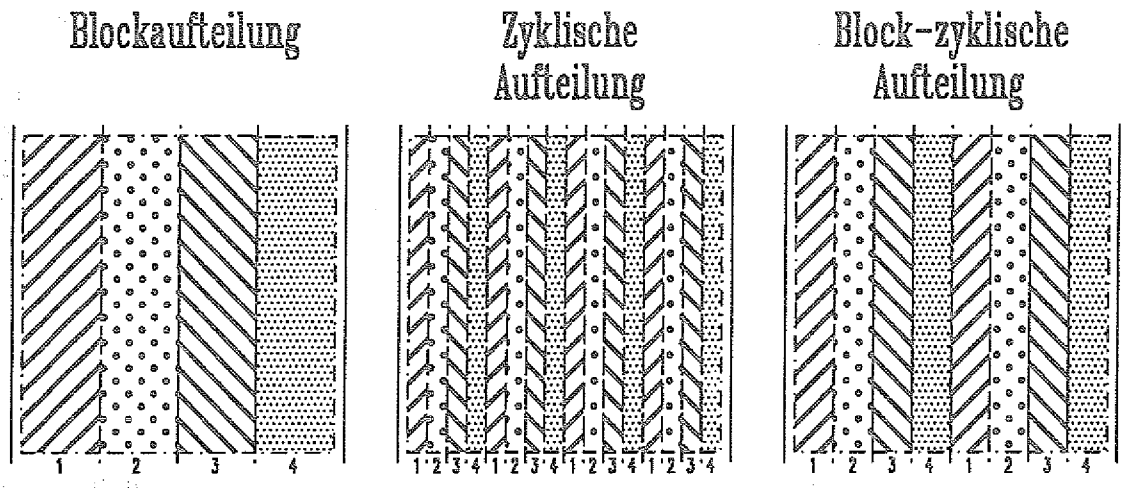


Abb. 1. Beispiele für Standardaufteilungen auf vier Prozesse: Die Zahlen unter den Spalten der Matrix geben jeweils die Nummer des Prozesses an, der diese Spalte(n) in seinem Speicher hat.

$n \bmod p$ Prozesse jeweils um eine Spalte größer als die der übrigen, also ist $k = \lceil n/p \rceil + 1$ für die ersten $n \bmod p$ Prozesse und $k = \lceil n/p \rceil$ für die restlichen Prozesse.

Der Vorteil dieser Aufteilung ist, daß sie sehr einfach und leicht zu verstehen und zu verwenden ist. Außerdem ist sie ideal für die Matrixmultiplikation, da auf Task Programmebene große Matrixblöcke in kontinuierlicher Adressierung multipliziert werden. Es können also im Task Programm BLAS 3-Module mit großen, zusammenhängenden Matrizen aufgerufen werden.

Der Nachteil dieser Aufteilung besteht darin, daß bei Zerlegungsroutinen nach k Schritten, wobei ein Schritt die Behandlung einer Spalte bedeutet, der erste Prozeß nichts mehr zu tun hat, nach $2 \cdot k$ Schritten der zweite usw., so daß eine sehr ungleichmäßige Lastverteilung entsteht.

4.3.2 Die zyklische Aufteilung

In der englischsprachigen Literatur ist dieses Aufteilungsschema unter dem Namen "Wrap-Mapping" bekannt geworden. Hierbei wird wie bei einem Kartenspiel den Prozessen reihum je eine Spalte der Matrix zugeteilt so, daß Prozeß j die Spalten i bekommt mit $i \bmod p = j, \quad j = 1, \ldots, p-1$ und daß Prozeß p die Spalten i mit $i \bmod p = 0$ bekommt. Auch hier haben, falls n kein Teiler von p ist, die ersten $n \bmod p$ Prozesse jeweils $\lceil n/p \rceil + 1$ Spalten und die übrigen haben $\lceil n/p \rceil$ Spalten in ihrem lokalen Speicher.

Diese Aufteilung ist ebenfalls leicht zu verstehen, aber nicht mehr ganz so leicht zu implementieren wie die Blockaufteilung. Für die Zerlegungsroutinen ist sie sehr günstig, da bis beinahe zum Schluß, genauer so lange noch mindestens p Spalten zu bearbeiten sind, alle Prozesse aktiv an der Berechnung beteiligt sind und ähnlich große Teile zu bearbeiten haben. Die Lastverteilung ist also fast ausgewogen.

Für die Matrixmultiplikation ist diese Aufteilung allerdings weniger günstig, da hier auf Task Programmebene die zweite Matrix jeweils mit Inkrement p angesprochen werden muß, wofür derzeit noch keine BLAS 3-Module vorgesehen sind (s. [12]).
Noch ungünstiger sieht dieses Verfahren im Zusammenhang mit Blockalgorithmen aus. Diese können bei einer zyklischen Aufteilung der Daten nicht effizient implementiert werden, da bei Blockalgorithmen immer zunächst ein kleiner Block zusammenhängender Spalten bearbeitet wird, ehe die übrige Matrix aktualisiert wird. Durch die zyklische Verteilung der Matrix hat jedoch kein Prozeß einen kleinen Block von zusammenhängenden Spalten in seinem lokalen Speicher. Zur Behandlung des kleinen Blockes wäre also bereits Kommunikation notwendig.

4.3.3 Die block-zyklische Aufteilung

Diese Aufteilung bildet eine Art von Kompromiß zwischen Blockaufteilung und zyklischer Aufteilung. Nicht einzelne Spalten sondern ganze Blöcke von *ncb* (number of columns per block, s. [28] und [29]) Spalten werden den einzelnen Prozessen reihum zugeteilt. Es gibt dann $\lceil n/ncb \rceil = nb$ Blöcke zu je *ncb* Spalten und eventuell noch einen kleineren Block. Jeder Prozeß hat $\lceil nb/p \rceil$ Blöcke in seinem Speicher, eventuell haben die ersten $nb \bmod p$ Prozesse wieder je einen Block mehr als die restlichen. Falls *ncb* kein Teiler von n ist, bekommt dann der $nb \bmod p + 1$ te Prozeß den unvollständigen Block. Die Anzahl der Spalten je Prozeß unterscheidet sich also um bis zu *ncb*.

Der Vorteil dieser Aufteilung ist, daß sie gut geeignet ist für die kommunikationssparenden Blockalgorithmen. Außerdem ist die Lastverteilung bei Zerlegungsroutinen fast so gut wie bei der zyklischen Aufteilung.

Andererseits ist die block-zyklische Aufteilung nicht so leicht zu überblicken und zu implementieren wie die beiden ersten. Bei der Matrixmultiplikation werden außerdem aus der zweiten Matrix jeweils Zeilenblöcke mit *ncb* Zeilen angesprochen, was sich nur schwer in BLAS 3-Modulen ausdrücken läßt. Schließlich muß eine einheitliche Blockgröße festgelegt werden, und hierfür gilt dasselbe wie für das Festlegen der Standardaufteilung überhaupt, keine Blockgröße ist für alle Problemstellungen gleich gut geeignet. Es müssen daher Kompromisse geschlossen werden.

4.3.4 Weitere Möglichkeiten

In [2] werden QR-Zerlegungsroutinen mit adaptiver Blockgröße und zyklischer Aufteilung der Matrix beschrieben, d.h. die Größe der Blöcke, die nacheinander reihum auf die Prozesse verteilt werden, richtet sich danach, wie weit die QR-Zerlegung schon fortgeschritten ist. Das Kriterium ist, die Zeit zu minimieren, während der ein Prozeß nicht arbeiten kann, weil ein anderer mit der Berechnung einer Block-Householder Matrix noch nicht fertig ist. Zunächst wird aus Maschinencharakteristiken und Komplexitätsanalysen ein kritischer Pfad bestimmt, mit dessen Hilfe sich die geeigneten sukzessiven Blockgrößen berechnen lassen. Dabei zeigt es sich, daß die ersten und letzten Blöcke nur eine Spalte breit sein sollten, während die Größe in der Mitte schnell zunehmen sollte. Der kritische Pfad und die Blockgrößen werden im voraus bestimmt und die Matrix entsprechend verteilt.

Etwas später wird in [2], bei der Zerlegung einer Matrix mit Pivotsuche, noch eine weitere Variante der Aufteilungsstrategie beschrieben. Hier wird die Matrix in Blockaufteilung den Prozessen gesendet, danach wählen die Prozesse reihum ihre lokale Pivotspalte und führen gemäß einem Rangkriterium eine Zerlegung durch oder nicht. Dabei wird nicht die ursprüngliche Matrix zerlegt, sondern eine Spaltenpermutation derselben.
Diese Strategie ist zwar auch recht kompliziert, aber eventuell am besten vereinbar mit einer Standardaufteilung. Die Matrizen werden in zusammenhängenden Blöcken auf die Prozesse verteilt, und dann wird die Aufteilung je nach Problem unterschiedlich interpretiert, was für den einfachsten Fall, wenn p Teiler von n und ncb Teiler von n/p ist, erläutert werden soll:
Bei der Matrixmultiplikation werden z.B. die Blöcke als Blöcke aufeinanderfolgender Spalten angesprochen. Dies bedeutet, daß Spalte i (oder kleiner Spaltenblock i mit ncb Spalten) von Prozeß j als Spalte (Spaltenblock) $(j-1)\cdot\frac{n}{p}+i$ der ursprünglichen Matrix betrachtet wird, das heißt, die Originalmatrix wird bearbeitet.
Bei Zerlegungen dagegen wird die Aufteilung als zyklische oder block-zyklische Aufteilung angesehen, was bedeutet, daß Spalte (Spaltenblock) i von Prozeß j als Spalte (Spaltenblock) $(i-1)\cdot p+j$ der spaltenpermutierten Originalmatrix angesehen wird (siehe auch 6.1).
Auch eine Betrachtung als zyklische Aufteilung mit adaptiver Blockgröße ist so möglich, jedoch mit der Einschränkung, daß die Gesamtzahl der Spalten, die jeder Prozeß besitzt, vorgegeben ist.

Leider ist es noch viel schwieriger, diese Strategie auf die Householdertransformation symmetrischer Matrizen zur Reduktion auf Tridiagonalgestalt zu übertragen, da hierbei nicht nur die Spalten, sondern auch die Zeilen permutiert werden müssen. Dies führt zu einer sehr komplizierten Adressierung der noch zu transformierenden Matrix.
Dieses Problem läßt sich umgehen, wenn man dafür in Kauf nimmt, daß mehr Speicherplatz verbraucht wird. Dann kann man Teile der Restmatrix einmal mit permutierten Zeilen und einmal mit der ursprünglichen Zeilenfolge speichern.

4.4 Weitere Optimierungsstrategien

Aufgrund der asynchronen Kommunikation beim SUPRENUM-Rechner, kann man sinnvollerweise auch Kommunikation auf einem Prozeß und Arithmetik auf einem anderen Prozeß parallel ausführen.
Eine wichtige Strategie hierbei ist das sogenannte "Compute-and-Send-Ahead". Das bedeutet, daß die zu sendenden Daten so früh wie möglich berechnet und gesendet werden, und daß alle anderen Berechnungen, die nicht unbedingt für das Senden notwendig sind, zurückgestellt werden.
Ein Beispiel für die Anwendung dieser Strategie sind die Zerlegungsroutinen, bei denen der Prozeß, der als nächster eine Teilzerlegung durchführen soll, nur die Spalten auf den aktuellen Stand bringt, die er sofort zerlegen muß. Nach deren Zerlegung und nach Senden der zerlegten Spalten werden dann erst die übrigen Spalten aktualisiert.

4.5 Kommunikationsstrategien

An zwei in der Linearen Algebra häufig auftretenden Kommunikationsproblemen soll hier dargestellt werden, welche Möglichkeiten es gibt, das gewünschte Ergebnis zu erlangen, und welcher Kommunikationsaufwand dabei zu erwarten ist.

Erläutert werden alle Verfahren zunächst unter der Annahme synchroner Kommunikation, wobei zusätzlich vorausgesetzt wird, daß sowohl Sender als auch Empfänger zur Kommunikation bereit sind. Anderenfalls muß der Sender bei synchroner Kommunikation warten bis der Empfänger bereit ist, ehe er seine Daten abschicken kann. Dies führt zu einem zusätzlichen Zeitaufwand für beide Prozesse, der nur von der Programmstruktur abhängt.

Unter den eben erläuterten Annahmen besteht eine Kommunikation aus einer Initialisierungs- und einer Übertragungsphase. Die Zeit für das Senden von n REAL-Zahlen von einem Prozeß zu einem anderen beträgt $T_{Kom} = t_\alpha + n\, t_\beta$, wobei t_α die sogenannte "Startup"-Zeit ist, also die Zeit, die nötig ist zum Aufbau der Verbindung, Herstellen von Sendebereitschaft auf der einen und Empfangsbereitschaft auf der anderen Seite und zur Vorbereitung der Nachricht; t_β ist die Zeit, die benötigt wird, um eine REAL-Zahl von einem Prozeß zu einem anderen zu übertragen. Während der gesamten Zeit $t_\alpha + nt_\beta$ sind beide Prozesse mit Kommunikation beschäftigt, danach können beide andere Befehle ausführen.
Bei vielen der existierenden Parallelrechnersystemen mit verteiltem Speicher ist $t_\beta \ll t_\alpha$, so daß für kleine n der zweite Term in erster Näherung vernachlässigbar ist.

Der SUPRENUM-Rechner sieht dagegen asynchrone Kommunikation vor, bei der das Senden unabhängig vom Empfangen geschehen kann. Der sendende Prozeß ist fertig mit der Kommunikation, sobald er eine Nachricht abgeschickt hat, auch wenn der empfangende Prozeß noch nicht bereit ist. Der Empfängerprozeß beginnt mit der Kommunikation, sobald er eine Nachricht benötigt. Dann muß er in seiner Mailbox nachsehen, ob diese vorliegt und sie von dort holen. Liegt sie nicht vor, muß er warten.
Ein Kommunikationsvorgang zerfällt also in drei Phasen, eine Phase der Initialisierung des Sendens (das ist die Phase zur Vorbereitung der Nachricht für den Bus), eine Übertragungsphase und eine Phase des Holens der Nachricht aus der Mailbox.
In der ersten Phase ist nur der sendende Prozeß mit Kommunikation beschäftigt, während der empfangende noch andere Befehle ausführen kann. Für die Dauer der dritten Phase ist nur noch der empfangende Prozeß mit der Kommunikation beschäftigt, wohingegen der sendende Prozeß schon neue Befehle ausführen kann. Insbesondere kann der Sender bereits ein neues Senden einleiten.
Die Zeit für das Senden von n REAL-Zahlen von einem Prozeß zu einem anderen würde damit $T_{Kom} = t_\gamma + nt_\beta + t_\delta$ betragen, wenn t_γ die Zeit für das Initialisieren des Sendens, t_β wie oben die Übertragungszeit für eine REAL-Zahl und t_δ die Zeit für die Initialisierung einer Empfangsoperation und das Empfangen bedeutet. Die Abhängigkeit der Zeit t_δ für das Empfangen der Nachricht von der Länge n der Nachricht wurde vereinfachend außer acht gelassen.
Der sendende Prozeß benötigt jedoch nur die Zeit t_γ , während der empfangende Prozeß im günstigsten Fall erst nach $t_\gamma + nt_\beta$ mit der Kommunikation beginnt und dann die Zeit t_δ benötigt.

Vergleicht man das kompliziertere asynchrone Modell mit dem synchronen Modell, so sieht man, daß man eine Abschätzung für den Kommunikationsaufwand bekommen kann, wenn man zur Vereinfachung das synchrone Modell mit kommunikationsbereitem Sender und Empfänger zugrunde legt und $t_\gamma + t_\delta$ für t_α einsetzt. Dabei läßt man die Möglichkeit außer acht, daß der sendende Prozeß schon nach der Zeit t_γ oder spätestens nach $t_\gamma + nt_\beta$, je nach Betriebssystem, neue Befehle ausführen kann. Die tatsächliche Zeit wird dabei höchstens überschätzt; Möglichkeiten, die kürzere Verzögerung des Senders auszunutzen, werden verschenkt.

Die im folgenden bestimmten Abschätzungen werden später im Kapitel „Das reelle symmetrische Eigenwertproblem" zur Abschätzung des Gesamtaufwandes verwendet. Ansonsten wird dort ein vereinfachtes asynchrones Modell zugrunde gelegt, bei dem Sender und Empfänger jeweils eine Initialisierungsphase für die Kommunikation durchlaufen. Die Initialisierungsphase des Empfängers kann zu einer beliebigen Zeit nach Beendigung der des Senders beginnen. Wartezeiten werden also nur dort betrachtet, wo der Empfänger früher bereit ist als der Sender. Die Übertragungszeit t_β wird vernachlässigt, da sie im Vergleich zur Initialisierungszeit klein ist.

4.5.1 Broadcast

Das Problem ist wie folgt gegeben:
Von p Prozessen, die mit Prozeß(1) bis Prozeß(p) numeriert seien, kennt ein Prozeß, oBdA Prozeß(1), ein Datum (einen Skalar oder Vektor etc. im Beispiel n REAL-Zahlen), das alle p Prozesse benötigen. Das Verschicken dieser Nachricht an die übrigen $p-1$ Prozesse läßt sich auf verschiedene Arten realisieren.
Da für den SUPRENUM-Rechner kein Hardware-Broadcast vorgesehen ist, muß eine Software-Lösung gefunden werden. Drei Beispiele sollen vorgestellt werden. Dabei wird immer davon ausgegangen, daß sowohl der Sender als auch alle Empfängerprozesse am Anfang des Broadcasts kommunikationsbereit sind.

4.5.1.1 Sukzessives Senden von Prozeß(1) an die übrigen Prozesse

Diese Strategie ist die einfachste: Prozeß(1) sendet nacheinander an Prozeß(2) bis Prozeß(p) die gewünschte Nachricht.
Setzt man synchrone Kommunikation voraus, so werden hier genau $p-1$ Kommunikationsschritte bestehend aus Initialisierung und Übertragung benötigt. Da alle Sendeoperationen auf demselben Prozeß initialisiert werden, können keine Kommunikationen parallel stattfinden. Die benötigte Zeit beträgt also $T^I_{Br} = (p-1)(t_\alpha + nt_\beta)$.
Diese Methode ist nur für kleine p sinnvoll. Wenn allerdings die Möglichkeit bestünde, daß das Initialisieren für das Senden derselben Nachricht an mehrere Prozesse nicht wesentlich länger dauert als das Initialisieren für das Senden an einen Prozeß, so wäre diese Methode wieder interessant.

4.5.1.2 Jeder, der die Nachricht empfangen hat, beteiligt sich am Senden

Diese Strategie wird in der Literatur für den Hypercube als „Verwendung spannender Bäume" beschrieben [26].

Hier geschieht das Senden in mehreren Stufen, wobei die Prozesse jeweils in kleinere Gruppen aufgeteilt werden. Prozeß(1) sendet die Nachricht zuerst an Prozeß($\frac{p}{2}+1$), dann gibt es zwei Gruppen von Prozessen, nämlich Prozeß(1) bis Prozeß($\frac{p}{2}$) und Prozeß($\frac{p}{2}+1$) bis Prozeß(p), in denen jeweils der erste Prozeß die gewünschte Nachricht kennt. In den beiden Teilmengen wird wieder wie oben beschrieben verfahren, bis nach $\lceil \log_2 p \rceil$ Stufen die Teilmengen einelementig sind. Dann kennt jeder Prozeß das Gewünschte. Mit $\lceil \log_2 p \rceil$ soll immer die kleinste ganze Zahl größer oder gleich $\log_2 p$ gemeint sein.
Bei dieser Methode wird in der ersten Stufe eine Initialisierung ausgeführt, in der zweiten zwei, die aber parallel stattfinden können, in der i-ten Stufe werden dann $2^{(i-1)}$ Initialisierungen für Kommunikation parallel ausgeführt.
Die gesamte benötigte Zeit beträgt somit bei synchroner Kommunikation $T_{Br}^{II} = \lceil \log_2 p \rceil (t_\alpha + n t_\beta)$, wobei auftretende Buskonflikte nicht berücksichtigt sind.

4.5.1.3 Senden über einen Binärbaum

Auch dies ist ein Vorgang, bei dem das Senden stufenweise abläuft, und auch die Menge der Prozesse wird auf jeder Stufe unterteilt. Im Gegensatz zum eben beschriebenen Fall führt jeder Prozeß hier jedoch maximal zwei Sendeoperationen aus, was zu einer besseren Lastverteilung führt.
Prozeß(1) sendet zuerst an Prozeß($1+(p+1)/2$) und an Prozeß(2). Damit ist die Menge der Prozesse, die die Nachricht empfangen sollen in zwei Teile geteilt, der erste enthält die Prozesse von 2 bis $(p+1)/2$, der zweite die Prozesse von $1+(p+1)/2$ bis p. In beiden Hälften kennt jeweils der erste Prozeß die Nachricht. Das Verfahren wird von vorn begonnen, bis auch hier am Ende alle Prozesse die gewünschte Nachricht kennen. Dies ist spätestens nach $\lceil \log_2(p+1) \rceil - 1$ Stufen erreicht. Auf jeder Stufe werden von den sendenden Prozessen jeweils zwei Nachrichten nacheinander verschickt, so daß im ungünstigsten Fall, wenn $p+1 = 2^\mu$, $\mu \in \mathbb{N}$, pro Stufe genau zwei Initialisierungsphasen anfallen.
Damit beträgt die Gesamtzeit für das Broadcasting einer Nachricht bei synchroner Kommunikation im ungünstigsten Fall $T_{Br}^{III} = 2(\lceil \log_2(p+1) \rceil - 1)(t_\alpha + n t_\beta)$, wobei aber jeder einzelne Prozeß nur für die Zeit $2(t_\alpha + n t_\beta)$ mit Kommunikation beschäftigt ist. Auch hier sind möglicherweise auftretende Buskonflikte nicht berücksichtigt.

4.5.1.4 Vergleich der Strategien

In Abb. 2 werden die drei möglichen Realisierungen des Broadcastings von Prozeß 1 an die Prozesse 2 bis 8 unter der Annahme synchroner Kommunikation und unter Vernachlässigung der Übertragungszeiten t_β dargestellt. Die Zeit wird in "Startups" gemessen, wobei ein "Startup" = t_α ist.
Man sieht, daß im Fall $p = 8$ und bei synchroner Kommunikation die zweite Möglichkeit am schnellsten ist. Die dritte Möglichkeit ist dennoch vorzuziehen, wenn es wichtig ist, daß Prozeß(1) möglichst schnell weiter arbeiten kann.

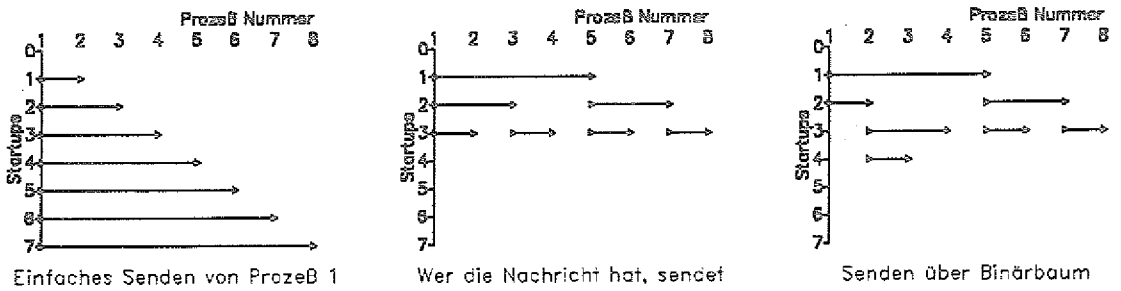


Abb. 2. Broadcast bei synchroner Kommunikation: Die drei vorgestellten Möglichkeiten jeweils bei 8 Prozessen und synchroner Kommunikation. Ein "Startup" ist die Zeit zur Initialisierung eines Kommunikationsvorgangs, im Text t_α genannt. Die Übertragungszeit wird in der Abbildung durch Setzen von $t_\beta = 0$ vernachlässigt.

In den Programmen zur Lösung des symmetrischen Eigenwertproblems ist die zweite Methode implementiert. Auch bei der Abschätzung des Gesamtaufwandes zur Berechnung von Eigenwerten und Eigenvektoren liegt dieses Modell zugrunde.

In Kombination mit der "Compute-and-Send-Ahead"-Strategie erweisen sich die zweite und dritte Version des Broadcastings als weniger günstig, da sie eine künstliche Synchronisation aller Prozesse einführen. Es genügt bei den baumartigen Broadcastversionen nicht, daß der Prozeß, der die Information berechnet hat, sie so schnell wie möglich abschickt, die Nachricht muß auch so schnell wie möglich von den anderen Prozessen weitergesendet werden, damit alle Prozesse sofort mit dieser neuen Information weiter arbeiten können. Anderenfalls beginnt für die Prozesse, die die Botschaft indirekt erhalten, die Kommunikationsphase erst, wenn die zwischengeschalteten Prozesse mit ihren Berechnungen fertig sind. Zu diesem Zeitpunkt sind auch die nachfolgenden Empfängerprozesse eventuell schon mit ihren Berechnungen fertig oder fast fertig, und müssen daher warten.
Um dies zu vermeiden, wäre es wünschenswert, daß ein Prozeß seine Berechnungen an beliebiger Stelle unterbrechen kann, wenn eine Nachricht bei ihm angekommen ist. Dies ist jedoch nur eingeschränkt und unter erheblichem zusätzlichem Zeitaufwand möglich, so daß ein solches Vorgehen nicht sinnvoll ist.
Bei allen Verfahren, bei denen reihum abwechselnd alle Prozesse einmal eine Nachricht an alle anderen Prozesse zu senden haben, zeigt sich außerdem, daß die Anzahl der Sende- und Empfangsoperationen pro Prozeß im Durchschnitt einer Runde bei allen Broadcastversionen etwa gleich ist. Einsparungen ergeben sich also nur durch Minimierung der Wartezeiten; dies kann zum Beispiel durch "Compute-and-Send-Ahead" geschehen und wird später noch am Beispiel der Rücktransformation von Eigenvektoren im Kapitel „Das reelle symmetrische Eigenwertproblem" erläutert.

4.5.1.5 Vergleich mit asynchroner Kommunikation

Setzt man das oben beschriebene asynchrone Modell mit $T_{Kom} = t_\gamma + nt_\beta + t_\delta$ voraus, so verschieben sich bei den Verfahren, die in Stufen ablaufen, die Stufen, da ein Prozeß, der gerade gesendet hat, sofort wieder senden kann, während der empfangende Prozeß zuerst noch die Nachricht annehmen muß.

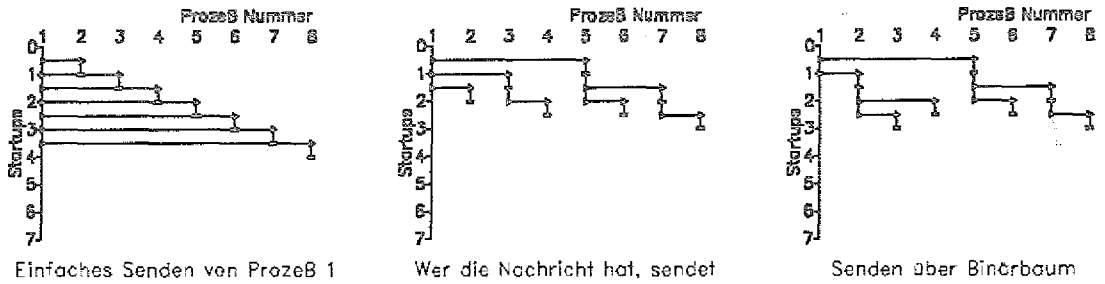


Abb. 3. Broadcast bei asynchroner Kommunikation: Die drei vorgestellten Möglichkeiten jeweils bei 8 Prozessen und asynchroner Kommunikation. Ein "Startup" besteht in diesem Beispiel je zur Hälfte aus der Initalisierung für das Senden, t_γ, und für das Empfangen der Nachricht, t_δ(also $t_\gamma = t_\delta = 1/2t_\alpha$). Auch in dieser Abbildung ist $t_\beta = 0$ gesetzt.

Beim sukzessiven Senden von Prozeß(1) an die anderen ergibt sich bei asynchroner Kommunikation eine Gesamtzeit für das Versenden der Nachricht an alle von $T^I_{Br} = (p-1)t_\gamma + nt_\beta + t_\delta$. Nach $(p-1)t_\gamma$ hat Prozeß(1) das letzte Senden initialisiert, bis die Nachricht dann beim letzten Empfänger angekommen ist, dauert es noch zusätzlich $nt_\beta + t_\delta$.
Im Gegensatz zum synchronen Fall, wo alle Kommunikationen sequentiell durchgeführt werden müssen, da immer Prozeß(1) beteiligt ist, kann hier das Empfangen einer Nachricht parallel zum Initialisieren eines neuen Sendevorgangs geschehen. Die Gesamtzeit für das Broadcasting reduziert sich dadurch gegenüber der Abschätzung aus dem synchronen Modell um $(p-2)t_\delta$, wenn man $t_\alpha = t_\gamma + t_\delta$ setzt.
Außerdem läst sich dieses Verfahren sehr gut mit der "Compute-and-Send-Ahead"-Technik kombinieren, da es für den Kommunikationsaufwand keine Rolle spielt, daß möglicherweise der Empfang der Nachricht erst nach Abschluß des Sendevorgangs erfolgt, wenn der sendende Prozeß bereits wieder mit Arithmetik beschäftigt ist.

Bei der zweiten Methode, bei der jeder Prozeß sendet, sobald er die Nachricht empfangen hat, wirkt sich die asynchrone Kommunikation auf die Gesamtzeit weniger aus. Es existiert nämlich ein Prozeß auf der letzten Stufe, bei dem die Nachricht auf jeder vorhergehenden Stufe empfangen und dann gesendet worden war. Bis dieser Prozeß die Nachricht empfangen hat, ist also die Zeit $T^{II}_{Br} = \lceil \log_2 p \rceil (t_\gamma + nt_\beta + t_\delta)$ vergangen. Dies entspricht genau der Gesamtzeit bei synchroner Kommunikation, wenn $t_\alpha = t_\gamma + t_\delta$. Möglicherweise treten aber weniger Buskonflikte auf als bei synchroner Kommunikation, da die Prozesse bei asynchroner Kommunikation nicht exakt gleichzeitig ihre Initialisierungen für das Senden durchführen.

Anders sieht es beim Senden mit Binärbaum aus. Zwar bekommt auch hier der letzte Prozeß die Nachricht wieder erst nach $\lceil \log_2 p \rceil$ Stufen, aber da jeder Prozeß auf jeder Stufe eine Initialisierung für Empfangen und zwei für Senden ausführt, ergibt sich die Gesamtzeit pro Stufe zu $2t_\gamma + t_\delta$ gegenüber 2 t_α im Fall synchroner Kommunikation. Dies ist gleichzeitig auch die Gesamtzeit, während der ein Prozeß mit Kommunikation beschäftigt ist. Die Gesamtzeit für das Broadcasting von n REAL-Zahlen ergibt sich somit im ungünstigsten Fall, wenn $p+1$ eine Zweierpotenz ist, zu

$T_{Br}^{III} = (2t_\gamma + t_\delta + nt_\beta)(\lceil \log_2(p+1) \rceil - 1)$ unter Vernachlässigung von Buskonflikten. Dies ist um $t_\delta(\lceil \log_2(p+1) \rceil - 1)$ weniger als bei dem synchronen Modell unter der Annahme, daß $t_\alpha = t_\gamma + t_\delta$ ist.

Beim Einsatz der "Compute-and-Send-Ahead"-Technik ist bei den beiden Broadcastversionen über Bäume die Zeit vom Beginn des Sendens bis zu dem Zeitpunkt, an dem der letzte Prozeß die Nachricht aus seiner Mailbox holt, sogar noch länger als in der Skizze gezeigt. Dies liegt daran, daß die Prozesse, die als erste empfangen sollen, eventuell noch nicht empfangsbereit sind, wenn die Nachricht bei ihnen eintrifft. Alle Prozesse außer dem Sender haben ja in der Regel noch Berechnungen durchzuführen. Die Zeit, die zusätzlich zu der in der Skizze angezeigten verbraucht wird, ist daher keine Wartezeit, sondern Rechenzeit.
Da der Prozeß, der auf diese Weise als letzter die Botschaft in seiner Mailbox vorfindet, aber nicht immer auch der ist, der sie als letzter benötigt, entstehen fast die gleichen Wartezeiten wie in der Skizze, sie treten nur später auf.

In Abb. 3 ist der Ablauf der drei Verfahren mit asynchroner Kommunikation bei $p = 8$, $t_\gamma = t_\delta$ und $t_\beta = 0$ dargestellt. Dabei bedeuten die Balken, daß zu diesem Zeitpunkt die Nachricht von dem betreffenden Prozeß aus seiner Mailbox entnommen worden ist. Die Zeiteinheit ist ein "Startup", wobei ein "Startup" $= t_\gamma + t_\delta$ gesetzt ist. Vereinfachend geht die Abbildung wie auch schon bei der synchronen Kommunikation davon aus, daß alle Empfängerprozesse von Anfang an auf die Nachricht warten, diese also sofort nach Eintreffen in der Mailbox aus dieser abrufen können.

4.5.2 Sammeln eines verteilten Vektors

Ein auf p Prozesse verteilter Vektor der Länge $n \cdot p$ soll komplett allen p Prozessen bekannt gemacht werden. Am Anfang kennt jeder Prozeß demnach einen Teilvektor der Länge n, am Ende soll er den gesamten Vektor kennen. Auch hierfür gibt es mehrere Möglichkeiten, von denen zwei vorgestellt werden sollen.

4.5.2.1 Jeder Prozeß sendet seinen Teilvektor an alle anderen Prozesse

Bei dieser einfachen Version können jeweils alle p Prozesse parallel die Initialisierung eines Sendevorganges und auch eines Empfangsvorganges durchführen. Da jeder Prozeß an $p-1$ Prozesse seinen Teil des Vektors sendet, ergibt sich bei synchroner Kommunikation unter Vernachlässigung der dabei auftretenden Buskonflikte eine Gesamtzeit von $T_{Sc}^{I} = (p-1)(t_\alpha + nt_\beta)$.
Bei asynchroner Kommunikation werden zuerst alle Sendevorgänge gestartet, danach die Empfangsvorgänge. Bis die Prozesse mit dem Empfangen starten, sollten alle schon wenigstens eine Nachricht in ihrer Mailbox haben, sodaß sich keine weiteren Verzögerungen beim Empfangen ergeben. Die Gesamtzeit ergibt sich analog der bei synchroner Kommunikation zu $T_{Sc}^{I} = (p-1)(t_\gamma + nt_\beta + t_\delta)$.

4.5.2.2 Senden immer größerer Stücke an jeweils andere Prozesse

Bei dieser Methode wird die Anzahl der zu sendenden Nachrichten dadurch verkleinert, daß die Nachrichten bei jedem weiteren Senden länger werden.

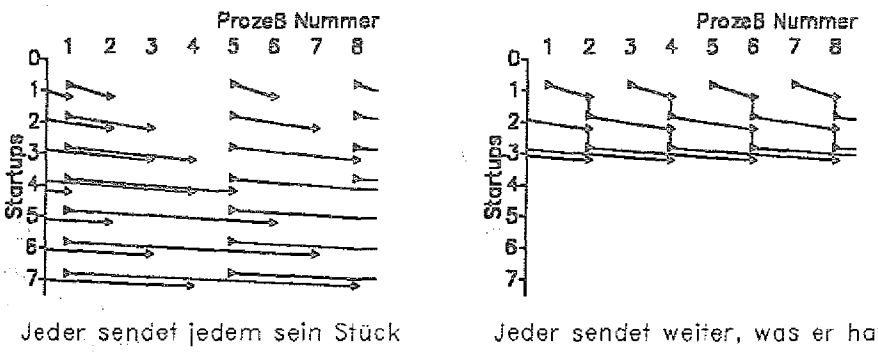


Abb. 4. Sammeln eines verteilten Vektors in 8 Prozessen: In der linken Skizze sind nur die Sendeaktionen der Prozesse 1, 5 und teilweise auch 8 dargestellt. In der rechten Skizze ist am Beispiel der von den Prozessen 1, 3, 5 und 7 gestarteten Nachrichten gezeigt, wie diese dann an jeweils andere Prozesse weitergesendet werden. Die über Prozeß(8) hinaus gehenden Pfeile sind durch die vor Prozeß(1) startenden Pfeile fortzusetzen. Wegen der Überschneidungen mußten die Pfeile schräg gezeichnet werden, was aussieht, als ob Übertragungszeiten t_β berücksichtigt wären. Dies ist jedoch nicht der Fall.

Prozeß(j), $j = 1, \ldots, p$, sendet zunächst an Prozeß($j + 1 \bmod p$) [1] seinen Anteil der Länge n des Vektors. Dann kennt jeder Prozeß einen Teil der Länge $2n$ des Vektors, nämlich den eigenen und den, den Prozeß($j - 1 \bmod p$) [1] besessen hatte. Diesen gesamten Anteil sendet er jetzt an Prozeß($j + 2 \bmod p$) [1]. Damit kennt Prozeß(j) jetzt die Anteile von Prozeß($j - 3 \bmod p$)[1] bis Prozeß(j), die er nun wieder an Prozeß($j + 4 \bmod p$)[1] sendet. Falls p eine Zweierpotenz ist, sendet somit im i-ten Schritt, $i = 1, \ldots, \log_2 p$ Prozeß(j) den gesamten Anteil von 2^{i-1} Teilstücken, den er bis dahin kennt, an Prozeß $(j + 2^{i-1} \bmod p)$ [1], so daß nach diesem Schritt Prozeß(j) die 2^i Anteile der Prozesse $(j - 2^i \bmod p)$[1] bis j kennt. Nach $\log_2 p$ Schritten kennt somit jeder Prozeß $2^{\log_2 p} = p$ Anteile des Vektors, also den ganzen Vektor.
Falls p keine Zweierpotenz ist, werden beim $\lceil \log_2 p \rceil$ -ten Senden nur noch die Anteile gesendet, die der Empfänger noch nicht kennt.

Unter der Annahme synchroner Kommunikation und unter Vernachlässigung von Buskonflikten beträgt die Gesamtzeit für das Sammeln des Vektors bei allen Prozessen

$$T_{Sa}^{II} = \lceil \log_2 p \rceil \, t_\alpha + \sum_{i=1}^{\lceil \log_2 p \rceil} 2^{i-1} n t_\beta \simeq \lceil \log_2 p \rceil \, t_\alpha + n \cdot p t_\beta .$$

1 Falls $j \bmod p = 0$, soll mit $(j \bmod p)$ p gemeint sein.

Asynchrone Kommunikation ändert am Ablauf prinzipiell nichts, da für das nächste Senden jeweils das Empfangen der vorigen Nachricht Voraussetzung ist. Die Kommunikation erfolgt also analog und die benötigte Zeit beträgt ebenfalls analog

$$T_{Sa}^{II} = \lceil \log_2 p \rceil (t_\gamma + t_\delta) + \sum_{i=1}^{\lceil \log_2 p \rceil} 2^{i-1} n t_\beta \simeq \lceil \log_2 p \rceil (t_\gamma + t_\delta) + n \cdot p t_\beta.$$

In Abb. 4 werden die beiden Methoden bei synchroner Kommunikation graphisch dargestellt. Die Zeit ist wieder in "Startups" gemessen, ein "Startup" $= t_\alpha$. Die Zeit für asynchrone Kommunikation ist dieselbe, wenn dort ein "Startup" $= t_\gamma + t_\delta$ gerechnet wird. Es zeigt sich, daß die zweite Methode in weniger Schritten und dadurch mit weniger "Startups" zum Ziel führt. Daher wurde dieses Verfahren im SUPRENUM Lineare Algebra Paket im Bereich Eigenwerte und bei der Realisierung der parallelen BLAS 2-Module verwendet. Die im Kapitel „Das reelle symmetrische Eigenwertproblem" erwähnten Zeiten beziehen sich auf diese Methode.

5.0 Die parallelen BLAS-Module

Ein Ziel der Erstellung von SLAP war es, möglichst für jedes der seriellen BLAS 2- und BLAS 3-Module [13], [12] eine parallele Version zur Verfügung zu stellen. Während für alle seriellen BLAS 3-Module parallele Versionen realisiert wurden, konnte das Ziel für die BLAS 2-Module nur zum Teil verwirklicht werden. Auf die Gründe wird in 5.1.1 näher eingegangen.

In ihrer ursprünglichen Form führen die BLAS 2- und BLAS 3-Module Matrix-Vektor- bzw. Matrix-Matrix-Operationen für serielle Rechner aus. Die BLAS 1-Module, die am Anfang der BLAS-Entwicklung standen, waren dagegen Programmkerne mit Routinen für Skalar-Vektor-Operationen und bilden heute das Fundament der BLAS-Hierarchie. Während die Operationsanzahl der BLAS 1-Module nur von der Ordnung $O(n)$ ist, steigt sie über $O(n^2)$ bei BLAS 2 bis auf $O(n^3)$ bei BLAS 3. Dies ist insbesondere für Rechner mit hierarchischer Speicherarchitektur und für Multiprozessoren mit lokalem Speicher, deren Knotenrechner auf der Basis des Botschaftsaustausches miteinander kommunizieren, von Vorteil. Da Botschaftsinitialisierungen auf Rechnern wie SUPRENUM im Vergleich zur Ausführung einer einzelnen Arithmetikoperation in der Regel relativ viel Zeit kosten, kann in numerischen Anwendungen durch die Verwendung von BLAS-Modulen gröberer Granularität der Kommunikationsaufwand, d.h. die Anzahl der zu übertragenden Botschaften, gesenkt werden, so daß effizientere Implementierungen als beim Einsatz von BLAS-Modulen feinerer Granularität möglich sind.

5.1 Umfang der parallelen BLAS-Module

Der erste Buchstabe des Namens gibt an, mit welcher Genauigkeit die BLAS-Module arbeiten. 'S' steht dabei für 'single precision'; 'double precision'-Modulnamen beginnen dagegen mit einem 'D'. Um eine Unterscheidung mit den seriellen BLAS-Modulen zu gewährleisten, verfügen alle parallelen BLAS-Module am Namensende über ein 'P'. Ansonsten stimmen die Namen der parallelen BLAS 2- und BLAS 3-Module mit denen der seriellen BLAS 2- und BLAS 3-Module überein.

5.1.1 Die parallelen BLAS 2-Module

SGEMVP: $y \leftarrow \alpha \mathbf{A} x + \beta y$

SGBMVP: $y \leftarrow \alpha \mathbf{B} x + \beta y$

SSYMVP: $y \leftarrow \alpha S x + \beta y$

SGERP: $A \leftarrow \alpha x y^T + A$

SSYRP: $S \leftarrow \alpha x x^T + S$

SSYR2P: $S \leftarrow \alpha x y^T + \alpha y x^T + S$

STRMVP: $x \leftarrow \mathbf{T} x$

STBMVP: $x \leftarrow \mathbf{TB} x$

STRSVP: löst die Vektorgleichung $\mathbf{T}\,x = y$

STBSVP: löst die Vektorgleichung $\mathbf{TB}\,x = y$

wobei	A	eine Rechteckmatrix ist,
	B	eine Rechteckmatrix mit Bandstruktur ist,
	S	eine symmetrische Matrix ist,
	$\mathbf{M}$	M oder M^T ist,
	M^T	die transponierte Matrix von M bezeichnet,
	T	eine Dreiecksmatrix bezeichnet,
	TB	eine Dreiecksmatrix mit Bandstruktur bezeichnet,
	x, y	Vektoren bezeichnen,
	x^T	den transponierten Vektor von x bezeichnet und
	α, β	Skalare sind.

BLAS 2-Module, die symmetrische Matrizen in gepackter Speicherung (SSPMV, SSPR, SSPR2, STPMV, STPSV) oder symmetrische Bandmatrizen (SSBMV) verwenden, konnten im Rahmen der hier gewählten Aufteilungsstrategie nicht effizient umgesetzt werden. Die gepackte Speicherung symmetrischer $n \times n$-Matrizen besteht in der spaltenweisen Anordnung der Matrixelemente der oberen (oder unteren) Hälfte einschließlich der Hauptdiagonalen in Form eines Vektors der Länge $(n^2 + n)/2$. Diese gepackte Speicherform symmetrischer Matrizen ist leider nicht verträglich mit der für SLAP gewählten Standardaufteilung gewöhnlicher Matrizen.

Die BLAS 2-Module, die an der dritten Stelle ihres Namens ein 'B' haben, operieren auf Matrizen mit Bandstruktur, die bei Aufruf in kompakter Speicherung vorliegen müssen. Die seriellen BLAS 2-Module SGBMV, STBMV und STBSV gehen davon aus, daß die Diagonalen der Bandmatrix, wie in Abb. 5 dargestellt, zeilenweise in einem zweidimensionalen Feld abgespeichert sind.

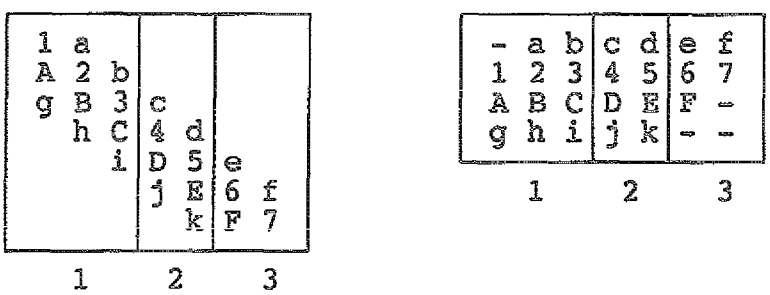


Abb. 5. Matrix mit Bandstruktur: in gewöhnlicher Speicherung (links) und in kompakter Speicherung (rechts) bei Standardaufteilung

Im Gegensatz zu der gepackten Speicherung für symmetrische Matrizen ist die kompakte Speicherung für Matrizen mit Bandstruktur mit der gewählten Standardaufteilung für die parallelen BLAS-Module verträglich, da die Standardaufteilung einer Matrix mit Bandstruktur, die in gewöhnlicher Speicherform vorliegt, sich in natürlicher Weise auf ihre Standardaufteilung in kompakter Speicherung überträgt (siehe Abb. 5). Dabei ist wichtig festzustellen, daß der Übergang von der einen Speicherung zur anderen, die Zu-

gehörigkeit der Matrixelemente zu den Spaltenblöcken nicht verändert. Dies wiederum hat zur Folge, daß ein derartiger Übergang ohne Kommunikation durchgeführt werden kann. Bei einer Standardaufteilung, die auf Zeilen- anstatt auf Spaltenblöcken basieren würde, ginge diese wünschenswerte Eigenschaft verloren.

5.1.2 Die parallelen BLAS 3-Module

SGEMMP: $C \leftarrow \alpha\,\mathbf{A}\,\mathbf{B} + \beta\,C$

SSYMMP: $C \leftarrow \alpha\,S\,\mathbf{B} + \beta\,C$

oder $C \leftarrow \alpha\,\mathbf{B}\,S + \beta\,C$

SSYRKP: $S \leftarrow \alpha\,A\,A^T + \beta\,S$

oder $S \leftarrow \alpha\,A^T A + \beta\,S$

SSYR2KP: $S \leftarrow \alpha\,(A\,B^T + B\,A^T) + \beta\,S$

oder $S \leftarrow \alpha\,(A^T B + B^T A) + \beta\,S$

STRMMP: $B \leftarrow \alpha\,\mathbf{T}\,B$

oder $B \leftarrow \alpha\,B\,\mathbf{T}$

STRSMP: löst die Matrixgleichung $\mathbf{T}\,X = \alpha\,B$

oder $X\,\mathbf{T} = \alpha\,B$

wobei

A, B, C, X	rechteckige Matrizen sind,
S	eine symmetrische Matrix ist,
$\mathbf{M}$	M oder M^T bezeichnet,
M^T	die transponierte Matrix von M bezeichnet,
T	eine Dreiecksmatrix bezeichnet,
α, β	Skalare sind.

Bei den parallelen BLAS 3-Modulen gibt es das Problem unterschiedlicher Speicherungen für die betreffenden Matrizen nicht, so daß für alle seriellen BLAS 3-Module entsprechende parallele Versionen geschrieben werden konnten.

5.2 Entwurfsprinzipien für die parallelen BLAS-Module

Im folgenden sollen die wichtigsten Prinzipien erläutert werden, auf denen die Implementierungen der parallelen BLAS 2- und BLAS 3-Module beruhen.

(a) Alle parallelen BLAS-Module weisen weitgehende Aufrufkompatibilität mit den entsprechenden seriellen BLAS-Modulen auf und verhalten sich funktional wie diese. Die parallelen BLAS-Module sind so konzipiert, daß sie auf der Task Programmeinheit (siehe 2.2) aufgerufen werden können. Es war jedoch notwendig, die Aufruflisten der

seriellen BLAS-Module um Parameter zu ergänzen, die die Umgebung beschreiben, in der die parallelen BLAS-Module ablaufen sollen. Zu diesen Parametern gehören

NP die Gesamtzahl der an der Berechnung beteiligten parallelen Prozesse,

PROC das sogenante Prozeßidentifikationsfeld, das die beteiligten Prozesse beschreibt, und

IMY die Nummer desjenigen Elementes des Prozeßidentifikationsfeldes, das das entsprechende Modul aufruft.

Die parallelen BLAS-Module sind außerdem so implementiert worden, daß bei entsprechender Besetzung dieser drei Parameter auch ein Aufruf auf der initialen Programmeinheit möglich ist. Das parallele BLAS-Modul verhält sich dann funktional genau wie das entsprechende serielle BLAS-Modul. Die parallelen BLAS-Module können daher als Verallgemeinerungen der seriellen BLAS-Module aufgefaßt werden.

Beispiel für die Aufrufkompatibiltät von SGEMVP mit SGEMV:

Aufruf von SGEMV (serielles BLAS 2-Modul)

CALL SGEMV (TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)

wobei

TRANS angibt, ob A oder A^T für die Matrix-Vektor-Multiplikation verwendet wird,

M die Zeilenanzahl von A angibt,

N die Spaltenanzahl von A angibt,

LDA die führende Dimension von A angibt,

INCX eine Schrittweite für den Vektor X angibt (die Matrix A bzw. A^T wird mit denjenigen Komponenten des Vektors X multipliziert, auf die beim Durchlauf mit der Schrittweite INCX zugegriffen wird),

INCY eine Schrittweite für den Vektor Y angibt (diejenigen Komponenten des Vektors Y werden zum Vektor-Matrix-Produkt addiert, auf die beim Durchlauf mit der Schrittweite INCY zugegriffen wird).

Die übrigen Parameter erklären sich selbst (siehe auch 5.1.1).

Aufruf von SGEMVP (paralleles BLAS 2-Modul)

CALL SGEMVP (TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY, NP, PROC, IMY, IHM, IHN, ITMP1, ITMP2, ITMP3, TEMP1, TEMP2, TEMP3, TEMP4, ERROR)

wobei

IHM die Aufteilung des Vektors Y beschreibt (siehe auch 5.4.1),

IHN die Spaltenblockaufteilung der Matrix A angibt (siehe auch 5.4.1),

ITMP1, ITMP2, ITMP3
temporäre Felder der Länge NP sind,

TEMP1, TEMP2
temporäre Felder der Länge M sind,

TEMP3, TEMP4
temporäre Felder der Länge N sind.

Die übrigen Parameter erklären sich selbst (siehe auch 5.1.1).

Im Falle des Aufrufs von SGEMVP auf der initialen Programmeinheit sind die Parameter NP, PROC und IMY folgendermaßen zu besetzen:

NP = 1, PROC = MASTER() und IMY = 1.

(b) Die parallelen BLAS-Module sollen effizient sein.

Um dieser Anforderung gerecht zu werden, wurden folgende Grundsätze befolgt:

- der Kommunikationsaufwand wurde unter Berücksichtigung der Speicherplatzanforderung so gering wie möglich gehalten,
- die parallelen BLAS-Module verwenden, wann immer möglich, lokal die optimierten seriellen BLAS-Module als Arithmetikbausteine,
- die parallelen BLAS-Module wurden für die SUPRENUM- Architektur sehr grob granularisiert,
- um die Vektoreinheiten der Prozessoren optimal ausnutzen zu können, wurde Wert auf möglichst große Vektorlängen gelegt,
- eine gleichmäßige Auslastung der Prozessoren wird weitestgehend gewährleistet,
- die "Compute-and-Send-Ahead"-Strategie wurde verwendet, um unnötiges Warten auf Botschaften zu vermeiden und um die zeitliche Überlappung von Arithmetik und Kommunikation zu ermöglichen (siehe auch 5.4.2).

Der Einsatz der "Compute-and-Send-Ahead"-Strategie zusammen mit der alternierenden Ausführung von Kommunikation und Arithmetik gestattet es, die Kommunikation mit der Arithmetik sowohl bei den parallelen BLAS 2- als auch bei den parallelen BLAS 3-Modulen zu überlappen.

Vereinfacht gesagt bedeutet die "Compute-and-Send-Ahead"-Strategie im Zusammenhang mit den parallelen BLAS-Realisierungen: Jeder Prozeß sendet seine Daten an andere Prozesse so früh wie möglich und empfängt selbst Daten so spät wie möglich. Weitere Einzelheiten zur Anwendung dieser Strategie bei dem parallelen BLAS 3-Modul SGEMMP finden sich in 5.4.2.

(c) Die parallelen BLAS-Module wurden für die Verarbeitung großer Datenobjekte (Matrizen und Vektoren) konzipiert.

Diese Forderung hat zur Konsequenz, daß der in der Regel bestehende Zielkonflikt zwischen Kommunikationsreduktion und Speicherplatzbedarf zugunsten der Minimierung des Speicherplatzbedarfs und damit zugunsten der Möglichkeit der Verarbeitung größerer Matrizen bzw. Vektoren entschieden worden ist. Diese Überlegungen spielten bei der Implementierung der Kommunikationsstruktur für die parallelen BLAS 3-Module eine wichtige Rolle (siehe 5.3).

Andererseits sollte man aber auch bedenken, daß eine Reduzierung der Kommunikation nur dann angestrebt werden sollte, wenn die eingesparte Kommunikationszeit auch zu einer Verringerung der gesamten Ausführungszeit führt. Dies ist z.B. dann nicht der Fall, wenn die anfallende Kommunikationszeit vollständig von Arithmetik überlappt wird. Sehr vereinfacht ausgedrückt heißt das: man kann sich dann relativ viel Kommunikation leisten, ohne die Gesamtzeit zu erhöhen, wenn auch viel Arithmetik anfällt. Wenn die aufgewendete Kommunikation dagegen nicht in einem ausgewogenen Verhältnis zur Arithmetik bleibt, wird sie sehr schnell zeitdominant.

(d) Ein- und Ausgabe aller parallelen BLAS-Module müssen bzgl. der Aufteilung der Datenobjekte konsistent sein, d.h. die Ausgabe eines parallelen BLAS-Moduls muß so auf die parallelen Prozesse aufgeteilt sein, daß ein nachfolgend aufgerufenes paralleles BLAS-Modul die vorliegende Ausgabe als Eingabe übernehmen kann, ohne die Aufteilung der Daten aufwendig ändern zu müssen.

Die Konsequenz dieser Forderung ist die Verwendung eines einheitlichen Datenaufteilungskonzepts für alle parallelen BLAS-Module, damit eine Aufteilungskonsistenz zwischen allen parallelen BLAS-Modulen gewährleistet ist. Gewählt wurde hierfür die Blockaufteilung wie sie in 4.3.1 eingeführt wurde. Allerdings können die Blockbreiten, um einen flexiblen Einsatz der Module zu ermöglichen, beim Aufruf der BLAS-Module variabel gewählt werden. Dies bedeutet, daß von dem in 4.3.1 beschriebenen starren Schema der Blockbreitenwahl abgewichen werden kann. So wird davon ausgegangen, daß Prozeß 1 den Block der ersten k_1 zusammenhängenden Spalten $(0 \le k_1 \le n)$ der $n \times n$-Matrix, Prozeß 2 den unmittelbar anschließenden Block von k_2 zusammenhängenden Spalten $(0 \le k_2 \le n)$ besitzt usw., wobei die Bedingung $\sum k_i = n$ zu erfüllen ist. Die Blockbreiten werden in Form eines Vektors dem aufgerufenen parallelen BLAS-Modul übergeben (siehe IHM und IHN in 5.2 (a)). Die konsequente Umsetzung dieses Prinzips bedeutet, daß die für die Erzeugung der Aufteilung der Ausgabeobjekte notwendige Kommunikation in den Algorithmusablauf und damit in das Modul selbst integriert werden kann und somit dem Benutzer verborgen bleibt.

5.3 Überlegungen zur Wahl der Kommunikationsstruktur bei den parallelen BLAS 2- und BLAS 3-Modulen

Die Überlegungen, die bei der Wahl der Kommunikationsstruktur der BLAS 2- und BLAS 3-Module die entscheidende Rolle gespielt haben, werden im folgenden exemplarisch an SGEMVP für BLAS 2 und an SGEMMP für BLAS 3 dargestellt. Zur Erläuterung einiger wichtiger Implementierungsdetails wird vereinfachend angenommen,

daß alle Vektoren die Länge n haben, die betrachteten Matrizen nicht-transponierte $n \times n$-Matrizen sind, p, die Anzahl der Prozesse, ein Teiler von n und $\beta = 0$ ist.

Bei der Implementierung der BLAS 3-Module liegt allen Modulen als Kommunikationsstruktur ein Ring von Prozessen zugrunde, den die Daten zyklisch durchlaufen. Aufgrund der Wahl dieser Kommunikationsstruktur ist es daher nicht notwendig, auf einem Prozeß temporären Speicherplatz von der Größe einer globalen Matrix, d.h. von n^2 Speicherplätzen, zur Verfügung zu stellen. Der Bedarf an temporärem Speicher ist daher auf jedem Prozeß nicht höher als die Größe einer lokalen Matrix, d.h. n^2/p Elemente. Speziell bei SGEMMP wird der Prozeßring von den Daten nur genau einmal durchlaufen, so daß jeder der p Prozesse während der gesamten Berechnung $p - 1$ Botschaften sendet und genau so viele empfängt. Der Kommunikationsaufwand pro Prozeß ist damit von der Ordnung $O(p)$. Der Arithmetikaufwand (Anzahl der Gleitpunktoperationen) beträgt bei Standardaufteilung (n/p aufeinanderfolgende Spalten auf jedem Prozeß) pro Prozeß rund $2n^3/p$ Operationen, ist also von der Ordnung $O(n^3)$. Im Falle der Multiplikation zweier nicht-transponierter Matrizen beträgt die maximale Länge einer Botschaft n^2/p, ist also von der Ordnung $O(n^2)$. Die maximale Botschaftslänge ist also um eine Größenordnung geringer als der Arithmetikaufwand.

Die Ergebnisse im Überblick:

SGEMMP:	$C \leftarrow \alpha A * B$	(BLAS 3-Modul)
Arithmetik pro Prozeß	$2 \cdot \frac{(n^3)}{p}$	$(O(n^3))$
max. Länge einer Botschaft	$\frac{n^2}{p}$	$(O(n^2))$
Botschaftsanzahl pro Prozeß	$p - 1$	$(O(p))$

Bei dem BLAS 2-Modul SGEMVP ($y \leftarrow \alpha A x$) stellen sich die Aufwände dagegen ganz anders dar. Hier ist der Arithmetikaufwand aufgrund der Matrix-Vektor-Multiplikation im Vergleich zu SGEMMP um eine Größenordnung geringer; er beträgt pro Prozeß nur $(2 \cdot n^2)/p$ Operationen, ist also von der Ordnung $O(n^2)$. Zwar ist auch hier die Botschaftslänge um eine Größenordnung geringer als der Arithmetikaufwand, jedoch erscheint es sinnvoll, bei SGEMVP den Kommunikationsaufwand, d.h. die Anzahl der zu verschickenden und zu empfangenden Botschaften, aus folgenden Gründen gegenüber SGEMMP zu senken:

(a) Um bei SGEMVP die Verhältnisse von Arithmetikaufwand, maximaler Botschaftslänge und Botschaftsanzahl ähnlich günstig wie bei SGEMMP zu gestalten (siehe Tabelle 1), sollte bei SGEMVP die Botschaftsanzahl von $O(p)$ auf $O(\log_2 p)$ gesenkt werden.

(b) Da man es bei SGEMVP in bezug auf die Kommunikation mit Vektoren und nicht mit Matrizen zu tun hat, kann man es sich leisten, für die Kommunikation temporären Speicherplatz von der Größe eines globalen Vektors zur Verfügung zu stellen, also dafür n Speicherplätze zu reservieren.

Insbesondere (b) erlaubt es, bei SGEMVP (und bei fast allen anderen BLAS 2-Modulen) eine Kommunikationsstruktur zu wählen, die mit einer Botschaftsanzahl von $\log_2 p$ pro Prozeß auskommt. Obwohl auch hier - wie bei SGEMMP - jeder Prozeß mit jedem anderen Prozeß Daten austauschen muß, kommt er mit weniger als $p-1$ Botschaften aus, falls $p > 3$ ist. Der Trick besteht darin, daß jeder Prozeß nur mit $\log_2 p$ Prozessen direkt und mit den anderen indirekt kommuniziert. Dies hat allerdings zur Folge, daß Botschaften auf anderen Prozessen kurzfristig zwischengespeichert werden müssen, bevor sie an ihren eigentlichen Adressaten weitergeleitet werden können. Dafür benötigt jeder Prozeß temporären Speicherplatz, der bei SGEMVP weniger stark ins Gewicht fällt als bei SGEMMP, da es sich nur um Vektoren und nicht wie bei SGEMMP um Matrizen handelt. Weitere Einzelheiten können dem in 5.4.1 angegebenen Beispiel entnommen werden.

Die Ergebnisse im Überblick:

SGEMVP:	$y \leftarrow \alpha A x$	(BLAS 2-Modul)
Arithmetik pro Prozeß	$2 \cdot \frac{n^2}{p}$	($O(n^2)$)
max. Länge einer Botschaft	$\frac{n}{2}$	($O(n)$)
Botschaftsanzahl pro Prozeß	$\log_2 p$	($O(\log_2 p)$)

Tabelle 1 gibt einen Überblick über die Verhältnisse von Arithmetik- und Kommunikationsaufwand bei den parallellen BLAS 2- und BLAS 3-Modulen. Dabei wurde vereinfachend angenommen, daß alle Matrizen $n \times n$-Matrizen sind. Die in Tabelle 1 angegebenen Werte treffen nicht auf diejenigen Module zu, die Matrix- oder Vektorgleichungen lösen.

	BLAS 3	BLAS 2
$O(n^3)$	ATM	
$O(n^2)$	MBL	ATM
$O(n)$		MBL
$O(p)$	BAZ	
$O(\log_2 p)$		BAZ

ATM = Arithmetikaufwand pro Prozeß in Anzahl Gleitpunktoperationen
MBL = maximale Botschaftslänge
BAZ = Botschaftsanzahl pro Prozeß
n = Matrixordnung
p = Prozeßanzahl.

Tab. 1. Überblick über Arithmetik- und Kommunikationsaufwand: parallele BLAS-Module mit Ausnahme der Module STRSVP, STBSVP und STRSMP

5.4 Die parallelen Algorithmen für SGEMVP und SGEMMP

In diesem Abschnitt wird anhand von Beispielen der parallele Algorithmus für die Matrix-Vektor- und die Matrix-Matrix-Multiplikation dargestellt. Dabei wird beson-

derer Wert auf die anschauliche Darstellung des Ablaufs der Kommunikation bei beiden Algorithmen gelegt. Aus diesem Grunde wurde für SGEMVP ein Beispiel mit Prozeßanzahl $p = 4$ und für SGEMMP eines mit Prozeßanzahl $p = 3$ gewählt. Die Verallgemeinerungen für andere Werte von p sind offensichtlich.

5.4.1 Der parallele Algorithmus für SGEMVP

Am Beispiel SGEMVP mit $\alpha = 1$ und $\beta = 0$, also der Berechnung von $y \leftarrow A\,x$ auf vier Prozessen, wird der Ablauf des Algorithmus für das parallele BLAS 2-Modul SGEMVP in Abb. 6 dargestellt. Die Eingabedaten, Vektor $x = (x_1, x_2, x_3, x_4)$ und Matrix $A = (A_{ij})$, $(i = 1, \ldots, 4;\ j = 1, \ldots, 4)$ sind zu Beginn gemäß der Standardaufteilung auf die vier Prozesse verteilt. Der Ausgabevektor $y = (y_1, y_2, y_3, y_4)$ muß so berechnet werden, daß er am Ende des Algorithmus in Standardaufteilung vorliegt. (Die x_k bzw. y_k bezeichnen Teilvektoren von x bzw. y und die A_{ij} Teilblöcke der Matrix A.)

Prozeß i $(i = 1, 2, 3, 4)$ verfügt in seinem lokalen Speicher über folgende Daten: Bei Algorithmusbeginn über x_i und $A_{\cdot i}$ und nach Algorithmusende zusätzlich über y_i. In drei Rechen- und zwei Kommunikationsschritten ist der Vektor $y = (y_1, y_2, y_3, y_4)$ bestimmt.

Die zwei Kommunikationsschritte sind in Abb. 6 dargestellt. Die Berechnung der einzelnen Komponenten von y erfolgt folgendermaßen:

$$y_1 = (A_{11}\,x_1 + A_{12}\,x_2) + (A_{13}\,x_3 + A_{14}\,x_4)$$

$$y_2 = (A_{22}\,x_2 + A_{23}\,x_3) + (A_{24}\,x_4 + A_{21}\,x_1)$$

$$y_3 = (A_{33}\,x_3 + A_{34}\,x_4) + (A_{31}\,x_1 + A_{32}\,x_2)$$

$$y_4 = (A_{44}\,x_4 + A_{41}\,x_1) + (A_{42}\,x_2 + A_{43}\,x_3)$$

Der Algorithmus ist in diesem Beispiel für den Fall beschrieben worden, daß p eine Zweierpotenz ist. Unter Inkaufnahme zusätzlichen Aufwandes wurde der Algorithmus jedoch so implementiert, daß der Ablauf auch für den Fall möglich ist, daß p keine Zweierpotenz ist.

Aus dem Algorithmusablauf geht hervor, daß es möglich ist, lokal das serielle BLAS 2-Modul SGEMV als Arithmetikbaustein einzusetzen sowie durch alternierende Ausführung von Kommunikation und Arithmetik die Überlappung beider Operationensarten zu ermöglichen (siehe auch 5.4.2).

5.4.2 Der parallele Algorithmus für SGEMMP

Am Beispiel SGEMMP mit $\alpha = 1$ und $\beta = 0$, also der Berechnung von $C \leftarrow A * B$ auf drei Prozessen, wird der Ablauf des Algorithmus für das parallele BLAS 3-Modul SGEMMP in Abb. 7 dargestellt. Die Eingabedaten, Matrix $A = (A_i)$, $(i = 1, 2, 3)$ und Matrix $B = (B_{ij})$, $(i = 1, 2, 3;\ j = 1, 2, 3)$ sind zu Beginn gemäß der Standardaufteilung auf die drei Prozesse verteilt. Die Ausgabematrix $C = (C_1, C_2, C_3)$ muß so berechnet werden, daß sie am Ende des Algorithmus in Standardaufteilung vorliegt.

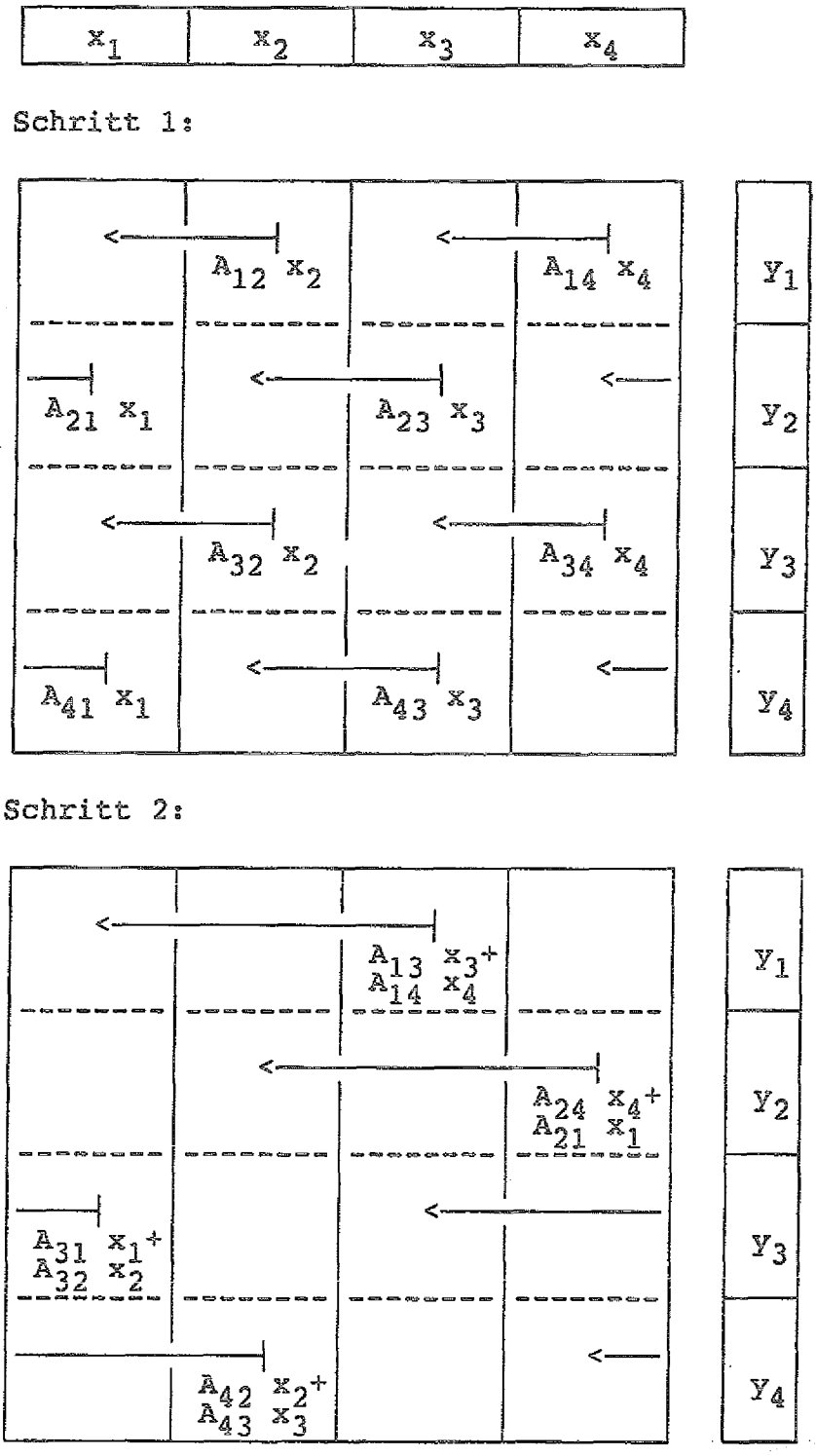


Abb. 6. Kommunikationsablauf für SGEMVP bei 4 Prozessen

Abb. 7 ist zu entnehmen, daß die Spaltenblöcke der Matrix A genau einmal durch den Prozeßring rotieren. Hierzu sind genau zwei Kommunikationsschritte und drei Rechenschritte notwendig. Eine Verallgemeinerung auf $p \neq 3$ Prozesse ist offensichtlich. Es ist festzuhalten, daß nur Daten von Matrix A in den Kommunikationsprozeß einbezogen

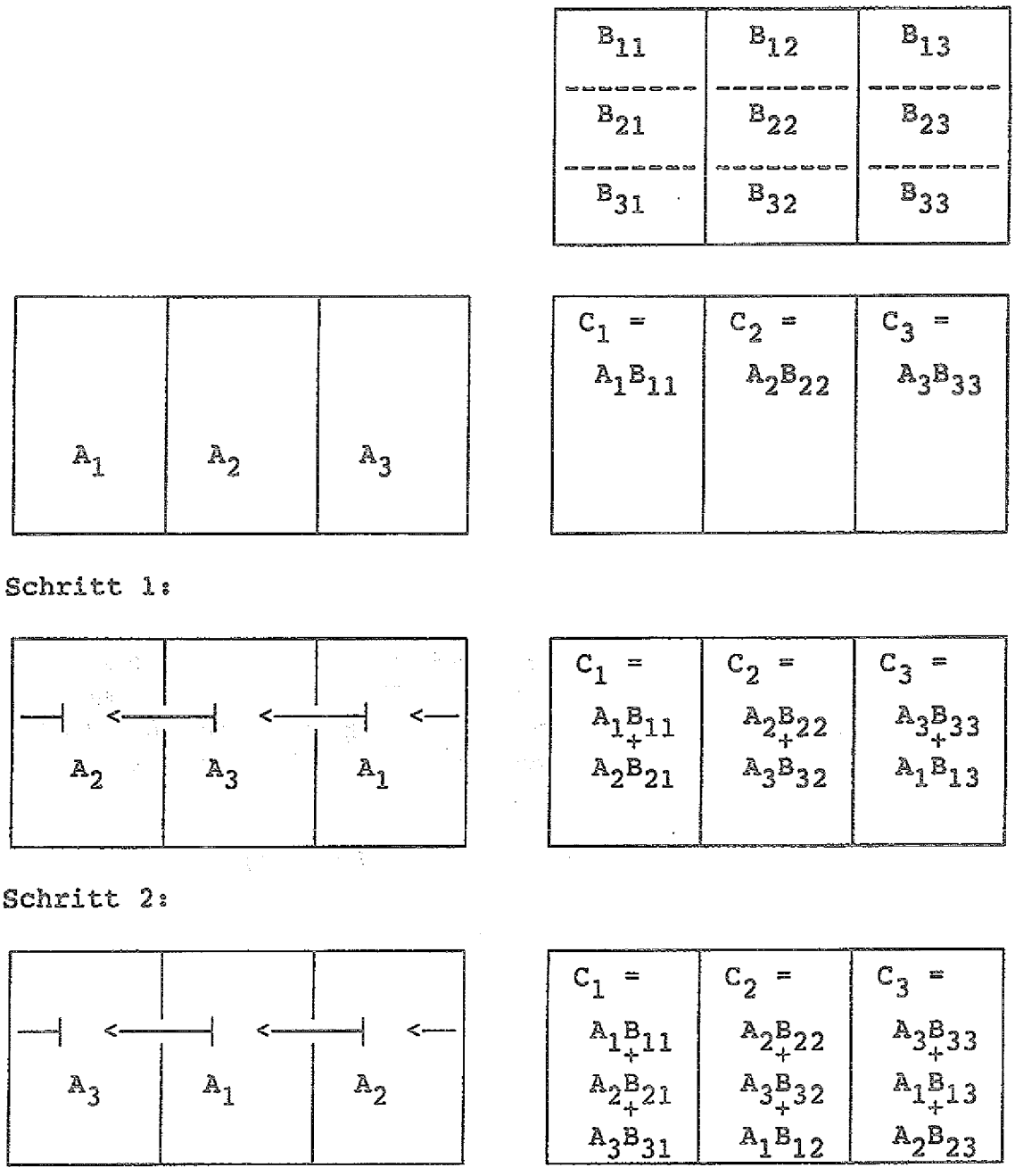


Abb. 7. Kommunikationsablauf für SGEMMP bei 3 Prozessen

werden; die Daten von Matrix *B* bzw. die schon berechneten Teile der Matrix *C* bleiben dagegen stationär.

Aus dem Algorithmusablauf geht hervor, daß es möglich ist, lokal das serielle BLAS 3-Modul SGEMM als Arithmetikbaustein einzusetzen sowie durch alternierende Ausführung von Kommunikation und Arithmetik die Überlappung beider Operationensarten zu ermöglichen. Wie dies im Detail ausssieht, soll am Beispiel des parallelen Algorithmus für SGEMMP erläutert werden. Es werden dabei nur diejenigen Schritte betrachtet, die Prozeß 1 im Algorithmus für SGEMMP ausführt. (Die beiden anderen

Prozesse führen parallel dazu analoge Schritte in der sich wiederholenden Reihenfolge 'Berechne, Sende, Empfange' durch.)

Die naive Implementierung von SGEMMP am Beispiel von Prozeß 1:

(1) Berechne $C_1 = A_1 * B_{11}$

(2) Sende A_1 an Prozeß 3

(3) Empfange A_2 von Prozeß 2

(4) Berechne $C_1 = C_1 + A_2 * B_{21}$

(5) Sende A_2 an Prozeß 3

(6) Empfange A_3 von Prozeß 2

(7) Berechne $C_1 = C_1 + A_3 * B_{31}$

(8) Ende

Diese naive Implementierung berücksichtigt nicht die Möglichkeit, so früh wie möglich zu senden. Schon im Schritt (1) ist es möglich, die Teilmatrix A_1 abzuschicken. Das unmittelbare Aufeinanderfolgen von Senden und Empfangen in den Schritten (2), (3), und (5), (6) gestattet keine Überlappung von Arithmetik und Kommunikation. Dies liegt daran, daß in Schritt (3) die Information empfangen wird, die in Schritt (2) von einem anderen Prozeß verschickt worden ist. Gleiches gilt auch für die Schritte (5) und (6). Ordnet man dagegen die Kommunikationsschritte durch Vertauschen von Schritt (1) und (2) sowie von Schritt (4) und (5) um, so verschwindet dieser Nachteil.

Implementierung von SGEMMP mit Überlappung von Kommunikation und Arithmetik am Beispiel von Prozeß 1:

(1) Sende A_1 an Prozeß 3

(2) Berechne $C_1 = A_1 * B_{11}$

(3) Empfange A_2 von Prozeß 2

(4) Sende A_2 an Prozeß 3

(5) Berechne $C_1 = C_1 + A_2 * B_{21}$

(6) Empfange A_3 von Prozeß 2

(7) Berechne $C_1 = C_1 + A_3 * B_{31}$

(8) Ende

Die Zeit, während der die Botschaften zwischen den Schritten (1) und (3) bzw. zwischen den Schritten (4) und (6) unterwegs sind, kann jetzt genutzt werden, um Arithmetik

auszuführen (Schritt (2) bzw. Schritt (5)). Damit wird eine Überlappung von Kommunikation und Arithmetik ermöglicht, was erheblich zur Effizienzsteigerung beiträgt.

Ähnliche Überlegungen, wie die eben skizzierten, können auch auf die parallele Implementierung des BLAS 2-Moduls SGEMVP angewendet werden. Durch entsprechende Anordnung von Senden, Rechnen und Empfangen kann auch hier die Effizienz der Implementierung erhöht werden.

5.5 Einige Bemerkungen zur Qualität der numerischen Ergebnisse

Wie die Beispiele in 5.4.1 und 5.4.2 gezeigt haben, löst ein paralleles BLAS-Modul das Gesamtproblem dadurch, daß es Teilergebnisse, deren Anzahl von der Prozeßanzahl abhängt, zum Gesamtergebnis zusammensetzt. Bei der Berechnung des Gesamtergebnisses - häufig durch Addition von Vektoren oder Matrizen - wird ausgiebig das Assoziativgesetz der Addition angewendet. Da dieses aber für Maschinenarithmetik keine Gültigkeit hat, kann ein paralleles BLAS-Modul numerisch betrachtet andere Ergebnisse als das entsprechende serielle BLAS-Modul liefern. Darüberhinaus kann der Fall eintreten, daß ein paralleles BLAS-Modul unterschiedliche numerische Werte bei Verwendung verschiedener Prozeßanzahlen p erzeugt. Das liegt daran, daß die Zerlegung des Gesamtproblems in Teilprobleme - also die Blockung - von p abhängig ist.

Um diesen aus der Theorie stammenden kritischen Anmerkungen praktische Resultate gegenüberzustellen, wurden einige numerische Tests mit Hilfe der SUPRENUM-Simulatoren durchgeführt. Die Ergebnisse lassen sich wie folgt zusammenfassen:

Die Rundungsfehler der parallelen BLAS-Module liegen in der Größenordnung der seriellen BLAS-Module, unabhängig von der verwendeten Prozeßanzahl p. Dies gilt auch für die BLAS-Module STRSVP, STBSVP und STRSMP, die Vektor- bzw. Matrixgleichungen lösen und mathematisch betrachtet sogar völlig andere Algorithmen als die entsprechenden seriellen BLAS-Module verwenden. (Die parallelen Versionen dieser 3 BLAS-Module basieren auf den in 6.3 beschriebenen Algorithmen von Li und Coleman [25]).

6.0 Das quadratische lineare Gleichungssystem

Ein Algorithmus zur Lösung eines linearen Gleichungssystems zerfällt i.d.R. in zwei Teile: zunächst wird die Koeffizientenmatrix faktorisiert, d.h. in einige Faktoren von einfacher Gestalt (oft Dreiecksmatrizen) zerlegt; anschließend kann mit Hilfe dieser Zerlegung das Gleichungssystem in einfacher Weise gelöst werden (im Falle von Dreicksmatrizen durch Vorwärts- bzw. Rückwärtseinsetzen).

Konkret seien A eine (reelle oder komplexe) quadratische Matrix der Ordnung n und b ein Vektor der Länge n; zu lösen sei $A\,x = b$. Dann läßt sich A auf jeden Fall, d.h. auch wenn die Matrix singulär ist, zunächst zerlegen in eine Permutationsmatrix P, eine untere Dreiecksmatrix L mit Einsen auf der Diagonalen und eine obere Dreiecksmatrix U: $A = PLU$. Falls U regulär ist, kann diese Faktorisierung benutzt werden, um das System $Ax = b$ zu lösen. Dies geschieht durch sukzessives Lösen von $L(Ux) = P^T b$, d.h. zunächst wird $b' = P^T b$ gebildet, dann $Ly = b'$ durch Vorwärts- und abschließend $Ux = y$ durch Rückwärtseinsetzen gelöst.

6.1 Datenaufteilung und allgemeine Strategie

Bei der Behandlung von linearen Gleichungssystemen auf einem Distributed-Memory-Rechner wie SUPRENUM ist zunächst die Frage der Datenaufteilung und deren Integration bzw. Umsetzung in einen Algorithmus zu klären, d.h. wie müssen die Daten in Form der Matrix A und des Vektors b (bzw. x) auf eine Anzahl p von Prozessen verteilt werden, damit sich bei geeigneter Gestaltung des Instruktionsflusses ein ausgewogener und effizienter Ablauf des parallelen Algorithmus ergibt.

An dieser Stelle sollen noch einmal die Vor- und Nachteile der verschiedenen Alternativen der Datenaufteilung anhand des Problems der Lösung linearer Gleichungssysteme detailliert erläutert werden. Zunächst besteht die Möglichkeit, eine Matrix zeilen- oder spaltenweise oder auch schachbrettartig auf eine Anzahl von Prozessen aufzuteilen (siehe Abb. 8). Zieht man in Betracht, daß bei der Lösung von linearen Gleichungssystemen der größte Teil des Rechenaufwandes bei der Faktorisierung der Koeffizientenmatrix anfällt, so erscheint es ratsam, die Datenaufteilung hinsichtlich dieses Teilproblems zu optimieren und dafür evtl. auch einige Effizienzeinbußen bei der Einsetzungsprozedur hinzunehmen.

Eine zeilenorientierte Aufteilung bringt Probleme mit sich, wenn bei der Zerlegung eine Teilpivotsuche durchgeführt werden soll: für jede Spalte wäre dann der betragsmäßig größte Wert der auf die Prozesse verteilten Elemente ab der Diagonalen zu finden; alleine diese Maximumsbildung würde über den gesamten Algorithmus gesehen etwa $n \cdot p$ Kommunikationen erforden - dieser Aufwand ist nur bei Multiprozessorsystemen mit sehr feiner Granularität und gegenüber der Arithmetik sehr schneller Kommunikation zu rechtfertigen, kommt also für SUPRENUM nicht in Frage. Für eine spaltenorientierte Aufteilung sprechen hingegen mehrere Gründe:

- Die Pivotsuche wird stark vereinfacht.

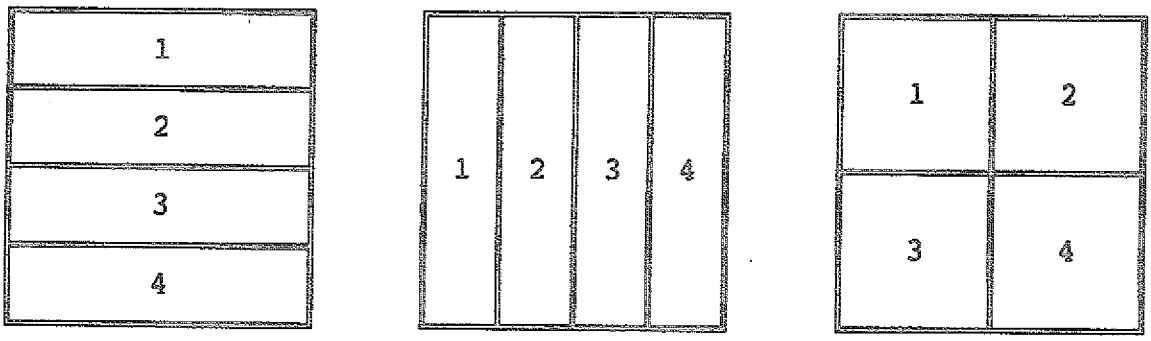


Abb. 8. Verschiedene Arten der Matrixaufteilung

- In Fortran werden bekannterweise die Elemente einer Matrix spaltenweise abgespeichert, spaltenweiser Zugriff bedeutet also Zugriff auf im Speicher aufeinanderfolgende Elemente (i.d.R. optimal).
- Auch LINPACK [11] bzw. die zugrundeliegenden Algorithmen arbeiten spaltenorientiert.
- Im Gegensatz zu früheren Annahmen in der Literatur [18] gibt es auch effiziente parallele Verfahren zur Lösung von Dreieckssystemen, deren Koeffizientenmatrizen spaltenweise verteilt sind.
- Bei einer schachbrettartigen Aufteilung tauchen (in abgeschwächter Form) ähnliche Probleme bzgl. der Pivotsuche wie bei einer Zeilenorientierung auf.
- Zudem ergibt sich bei schachbrettartiger Aufteilung eine schlechte Lastverteilung, weil z.B. die Prozesse 1, 2 und 3 in Abb. 8(c) nichts mehr zu tun haben, sobald die erste Hälfte der Spalten bearbeitet worden ist.

Aus diesen Gründen empfiehlt es sich, von einer Spaltenorientierung auszugehen. Hierbei stellt sich dann die Frage, welche der drei in 4.3 vorgestellten Aufteilungen (oder welche andere) für die Lösung von linearen Gleichungssystemen am günstigsten ist. Dazu muß im Vorgriff auf die eigentliche Beschreibung des parallelen Algorithmus kurz auf die Kommunikations- und Algorithmusstruktur eingegangen werden, die sich bei der Gaußelimination unabhängig von der Art der Datenaufteilung ergibt.

Der Ablauf des Verfahrens ist dadurch gekennzeichnet, daß in jedem Schritt eine Spalte oder eine Anzahl von aufeinanderfolgenden Spalten (genauer: die Elemente dieser Spalte(n) ab der Diagonalen) Pivotspalte(n) ist (sind). Derjenige Prozeß, der diese Spalte(n) in seinem Speicher hält, fungiert in diesem Schritt als zentrale Task, die das Ergebnis ihrer lokalen Berechnungen an alle anderen Prozesse schickt, damit diese ihre global gesehen rechts von der (den) Pivotspalte(n) liegenden Daten aktualisieren. Die Pivotspalte(n) durchläuft (durchlaufen) die Gesamtheit der Spalten der Matrix von links nach rechts. Somit besteht die Kommunikationsstruktur aus einer Folge von Broadcasts, deren Sender die Menge der Prozesse (bei zyklischer Aufteilung evtl. mehrmals) durchlaufen; die Anzahl der Broadcasts ist gleich der Anzahl der Blöcke. Der Großteil des Rechenaufwandes fällt bei der Aktualisierung der lokalen Teile der jeweiligen Restmatrix an.

Bei einer Blockaufteilung der Matrix wie in 4.3.1 beschrieben ist die Anzahl der Sends und Receives minimal, da die Anzahl der Blöcke minimal ist; dies ist angesichts der zu erwartenden relativ großen Startup-Zeiten bei der Kommunikation auf SUPRENUM positiv. Nicht akzeptabel ist dagegen der Umstand, daß bei der Blockaufteilung der erste Prozeß bereits nach der Bearbeitung von n/p Spalten und die folgenden auch recht bald „arbeitslos" werden; d.h. die Lastverteilung ist - über den gesamten Algorithmus gesehen - sehr schlecht.

Durch die zyklische Aufteilung (siehe 4.3.2) wird dieser Nachteil fast vollkommen aufgehoben. Den Preis, den man dafür zu zahlen hat, ist ein Ansteigen der Broadcasts auf n. Dies ist wiederum auf SUPRENUM wegen der relativ „teuren" Kommunikation nicht hinzunehmen.

Als Kompromiß bietet sich also die block-zyklische Aufteilung an (siehe 4.3.3) - am besten mit variablen Blockbreiten (4.3.4), um lange Wartezeiten der letzten Prozesse in der Anlaufphase des Algorithmus zu verhindern. Diese schwierig zu überblickende (im übrigen allgemeinste, d.h. alle anderen spaltenorientierten umfassende) Aufteilung ist dem Benutzer jedoch kaum zuzumuten. Für die Gleichungssystemroutinen in SLAP wurde daher folgende Lösung gefunden:

Ausgangspunkt ist die für den Benutzer am einfachsten zu erfassende Form des Mapping, die Blockaufteilung; d.h. vor dem Aufruf der Faktorisierungs- und Lösungsroutinen müssen die Matrix A und der Vektor b entsprechend auf die Prozesse verteilt werden. Intern in den SLAP-Routinen werden diese „Makroblöcke" jedoch noch einmal unterteilt in kleinere Blöcke, die als Einheiten bzgl. der implementierten Blockalgorithmen dienen. Diese „Mikroblöcke" sind von variabler Breite, die adaptiv aufgrund der Lage der Blöcke innerhalb der Gesamtmatrix sowie aufgrund von Problemgrößen wie n und p sowie von Maschinenparametern (z.B. Knotenprozessorleistung und Übertragungsgeschwindigkeit) automatisch bestimmt wird. Dabei ist zu bemerken, daß die Mikroblöcke nicht in ihrer „natürlichen" Reihenfolge (Mikroblock 1, 2, ... von Prozeß 1, Mikroblock 1, 2, ... von Prozeß 2, ...) bearbeitet werden, denn dies würde im Vergleich zur Blockaufteilung keinen Vorteil im Sinne einer besseren Lastverteilung sondern nur eine feinere Granularität und damit auf SUPRENUM sogar einen Nachteil bringen. Vielmehr werden diese kleineren Blöcke der einzelnen Prozesse in einer Reihenfolge bearbeitet, die der zyklischen Aufteilung entspricht: Mikroblock 1 von Prozeß 1, Mikroblock 1 von Prozeß 2, ..., Mikroblock 1 von Prozeß p, Mikroblock 2 von Prozeß 1,. (siehe Abb. 9). Dies bedeutet nun aber, daß nicht die ursprüngliche Matrix A sondern eine spaltenpermutierte $A' = AP'$ bearbeitet wird. Auf lineare Gleichungssysteme bezogen heißt das, daß die Unbekannten vertauscht worden sind. Berücksichtigt man, daß bei der Lösungsprozedur aufgrund der Pivotisierung sowieso zu Anfang die rechte Seite permutiert werden muß und sich diese beiden Vertauschungsoperationen (ohne Mehraufwand an Kommunkation!) zusammenfassen lassen, so erweist sich dieses Vorgehen als optimal:

- der Benutzer „sieht" nur die einfache Blockaufteilung,
- durch die intern zyklische Bearbeitung der Mikroblöcke wird nahezu über die gesamte Dauer des Algorithmus eine gleichmäßige Lastverteilung erreicht,
- die Anzahl der Kommunikationsbefehle hält sich aufgrund der (Mikro-)Blockung in Grenzen,

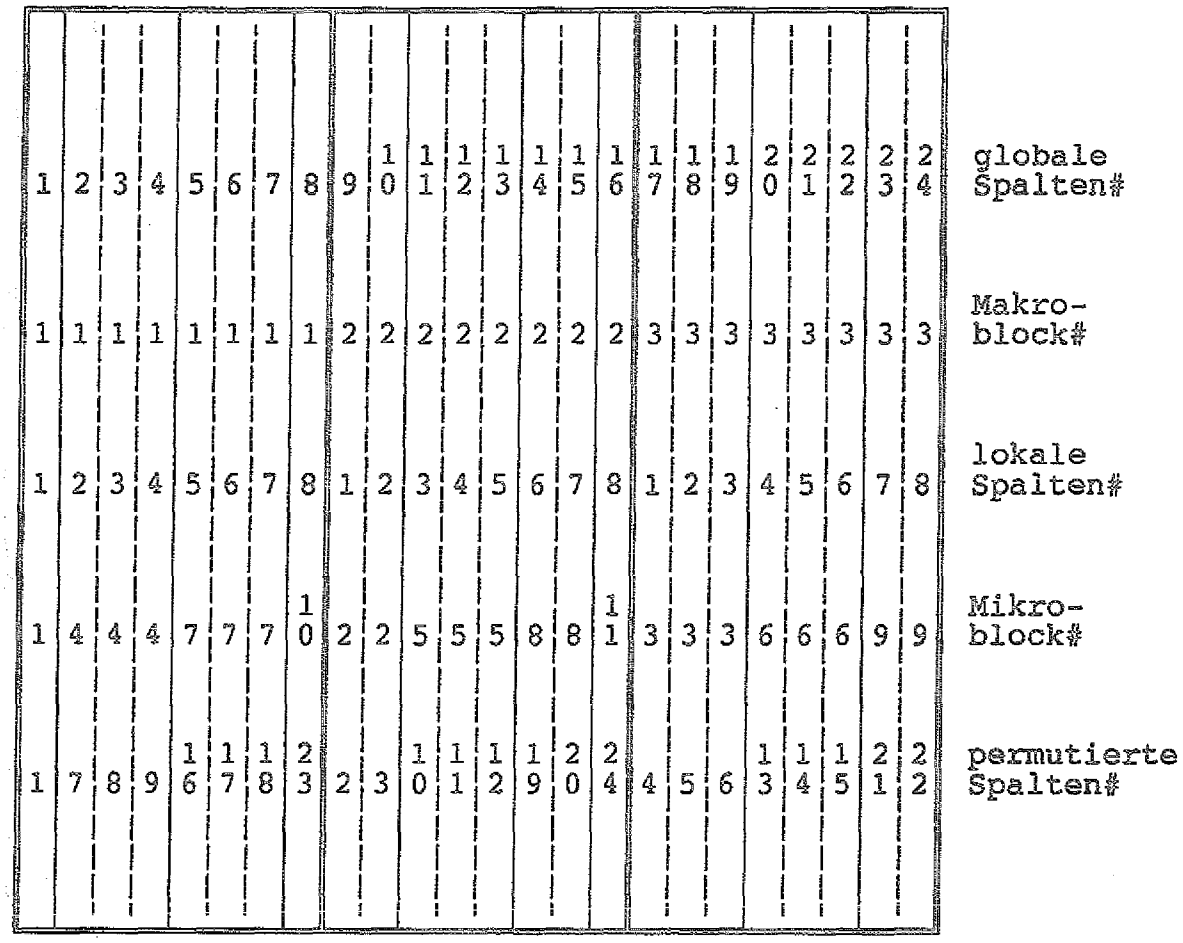


Abb. 9. Makro- und Mikroblockung

- der Gebrauch von BLAS 3-Modulen ist möglich.

Zu beachten ist hierbei allerdings, daß das Ergebnis der Faktorisierung wegen der Spaltenpermutation von A nicht mit dem der entsprechenden LINPACK-Routine übereinstimmt. In aller Regel dürfte den Benutzer dieses (Zwischen-)Ergebnis allerdings auch weniger interessieren als die Lösung des Gleichungssystems, die nicht prinzipiell von der von LINPACK berechneten abweicht. Durch die andere Faktorisierung können sich andere Rundungsfehler ergeben, die aber nicht generell größer sind als bei LINPACK.

6.2 Faktorisierung

Der vorbereitende Schritt zur Lösung eines linearen Gleichungssystems mit vollbesetzter Koeffizientenmatrix A besteht also darin, die Matrix wie folgt zu zerlegen:

$A = PLU$ mit $P = \prod_{j=1}^{n} P_j$ (wobei P_j durch die Vertauschung der j-ten und der p_j-ten Zeile der Einheitsmatrix entsteht mit $j \leq p_j \leq n$), $l_{ii} = 1$ für $i = 1, \ldots, n$, $l_{ij} = 0$ für $i < j$ und $u_{ij} = 0$ für $i > j$.
Bei dem dazu in SLAP verwendeten Algorithmus handelt es sich um eine Blockversion der Gauß-Elimination mit Teilpivotsuche. Das Verfahren arbeitet also (mikro-) blockweise die (globale) Diagonale entlang, d.h. die durch die oben beschriebene virtuelle zyklisch-variable Blockaufteilung entstandenen Blöcke werden der Reihe nach von dem jeweiligen Prozeß in einem Schritt bearbeitet. (Aus Gründen der Übersichtlichkeit wird im folgenden davon ausgegangen, daß die Matrix A zyklisch-variabel blockaufgeteilt vorliegt; die virtuelle Permutation P' wird vorausgesetzt, d.h. A ist eigentlich A'.)

Genauer: der aktuelle Spaltenblock ab der Diagonalen wird von dem zuständigen Prozeß faktorisiert (d.h. im k-ten Blockschritt wird eine LU-Zerlegung der in Abb. 10 schraffierten Teilmatrix ausgeführt); das Ergebnis wird benutzt, um die Restmatrix zu aktualisieren; dazu müssen $P^{(k)} := \prod_{j=j_2}^{j_1} P_j$, L_{kk} und $L_{\bullet k}$ an die anderen Prozesse geschickt werden (siehe 5.1). Es handelt sich also um einen Algorithmus mit sog. "Immediate Update": die noch zu verarbeitende Restmatrix wird sobald als möglich aktualisiert (im Gegensatz dazu wird in LINPACK und LAPACK das "Delayed Update" verwendet). Der Grund für die Wahl dieser Variante liegt darin, daß wegen des verteilten Speichers nur so ein hochgradig paralleles Arbeiten der verschiedenen Prozesse an der verteilten Matrix möglich ist.

Läßt man die zur Effizienzsteigerung integrierte und weiter unten beschriebene sog. "Compute-and-Send-Ahead"-Technik zunächst außer acht, so läßt sich der Faktorisierungsalgorithmus wie folgt formulieren:

```
FOR k = 1,nb DO
    IF (k-ter Block gehört mir) THEN
        berechne P^(k), L_kk, L_•k und U_kk
        sende P^(k), L_kk und L_•k an alle anderen Prozesse
    ELSE
        empfange P^(k), L_kk und L_•k
    ENDIF
    berechne den lokalen Teil von U_k• und A^(k)
ENDFOR
```

Die oben erwähnte "Compute-and-Send-Ahead"-Strategie hat den Zweck, unnötig lange Wartezeiten der Prozesse auf Zwischenergebnisse desjenigen Prozesses zu vermeiden, der den aktuellen Block bearbeitet. Dies wird erreicht, indem der aktuelle Prozeß zunächst nur die Operationen ausführt, die zur Berechnung der zu verschickenden Daten notwendig sind. D.h. das hier verwendete Verfahren arbeitet nach dem Prinzip „sende so

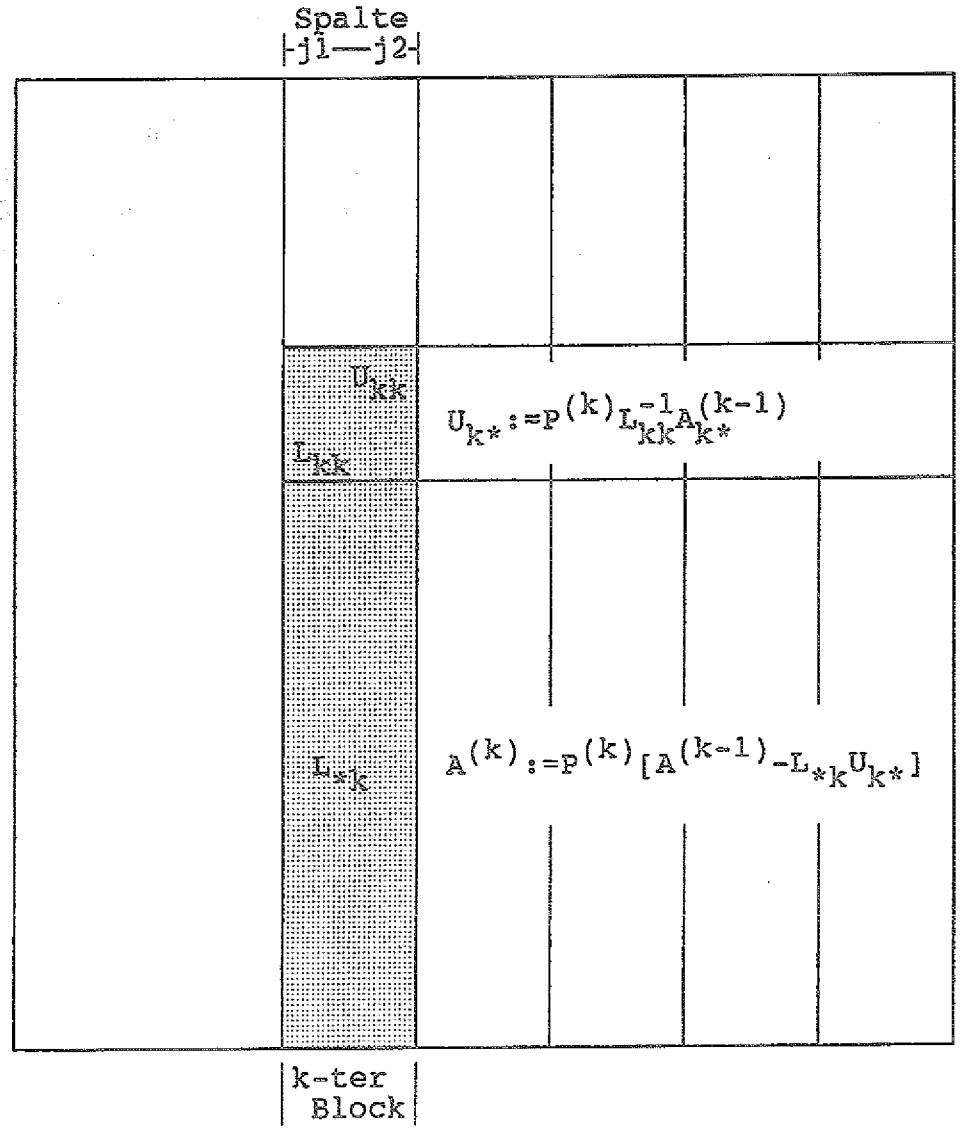


Abb. 10. k-ter Blockschritt der Gaußelimination: Die unteren Indices bezeichnen die Blocknummern; ein Stern deutet an, daß alle Blöcke angesprochen werden, die nach dem Index in der anderen Dimension liegen: z.B. bedeutet $L_{*k} = L_{k+1:nb,k}$. Die in Klammern gesetzten oberen Indices beziehen sich auf die Schritte des Algorithmus.

früh wie möglich". Dies hat zur Folge, daß sich die Schritte des obigen Algorithmus in dem Sinne überlappen, daß zunächst nur ein Teil der Berechnungen des Schrittes k ausgeführt wird, anschließend ein Teil von Schritt $k+1$ (lokale LU-Zerlegung und Broadcast) und die restlichen Operationen aus Schritt k und $k+1$ dann nachgeholt werden.

Mit anderen Worten: die starre DO-Loop-Struktur, bei der in jedem Schleifendurchlauf genau die Operationen des Schrittes k ausgeführt werden, wird aufgebrochen. Stattdessen wird jetzt eine WHILE-Struktur verwendet, mit der sich der modifizierte Algorithmus auch programmsprachlich sauber ausdrücken läßt:

$k = 1$
nb = number_of_blocks
ip = #MYPROCESS
IF $ip = 1$ THEN
lokale LU-Zerlegung von Block 1 :

$$A^{(0)} = P^{(1)} \begin{bmatrix} L_{1,1} \\ L_{*,1} \end{bmatrix} U_{11}$$

BROADCAST $P^{(1)}$, $L_{1,1}$, $L_{*,1}$
Permutation der Zeilen von A ohne Block 1 (Schritt 1) :

$$A^{(1)} = P^{(1)} A^{(0)}_{1,p+1:nb:p}$$

Aktualisierung der lokalen Restmatrix (Schritt 1) :

$$U_{1,p+1:nb:p} = L_{1,1}^{-1} A^{(1)}_{1,p+1:nb:p}$$
$$A^{(1)}_{*,p+1:nb:p} = A^{(1)}_{*,p+1:nb:p} - L_{*,1} U_{1,p+1:nb:p}$$

$k = 2$
ENDIF
WHILE $k \leq nb$ DO
RECEIVE $P^{(k)}$, $L_{k,k}$, $L_{*,k}$ FROM PROCESS MOD$(k-1,p)+1$
IF $k = nb$ THEN
Permutation der Zeilen von A:

$$A^{(nb)}_{*,nb} = P^{(nb)} A^{(nb-1)}_{*,nb}$$

EXIT
ENDIF
IF MOD$(k,p)+1 = ip$ THEN
Permutation der Zeilen von Block $k+1$ (Schritt k):

$$A^{(k)}_{*,k+1} = P^{(k)} A^{(k-1)}_{*,k+1}$$

Aktualisierung von Block $k+1$ (Schritt k):

$$U^{-1}_{k,k+1} = L^{(k)}_{k,k} A^{(k)}_{k,k+1}$$
$$A^{(k)}_{*,k+1} = A^{(k)}_{*,k+1} - L_{*,k} U_{k,k+1}$$

lokale LU-Zerlegung von Block $k+1$:

$$A^{(k)}_{*,k+1} = P^{(k+1)} \begin{bmatrix} L_{k+1,k+1} \\ L_{*,k+1} \end{bmatrix} U_{k+1,k+1}$$

BROADCAST $P^{(k+1)}$, $L_{k+1,k+1}$, $L_{*,k+1}$
Permutation der Zeilen von A ohne Block $k+1$ (Schritt k):

$$A^{(k)}_{*,ip:k:p} = P^{(k)} A^{(k-1)}_{*,ip:k:p}$$
$$A^{(k)}_{*,k+1+p:nb:p} = P^{(k)} A^{(k-1)}_{*,k+1+p:nb:p}$$

Aktualisierung ab Block $k+1+p$ (Schritt k):

$$U_{k,k+1+p:nb:p} = L^{-1}_{k,k} A^{(k)}_{k,k+1+p:nb:p}$$
$$A^{(k)}_{*,k+1+p:nb:p} = A^{(k)}_{*,k+1+p} - L_{*,k} U_{k,k+1+p:nb:p}$$

Permutation der Zeilen von A ohne Block $k+1$ (Schritt $k+1$):

$$A^{(k+1)}_{*,ip:k:p} = P^{(k+1)} A^{(k)}_{*,ip:k:p}$$

$$A^{(k+1)}_{*,k+1+p:nb:p} = P^{(k+1)} A^{(k)}_{*,k+1+p:nb:p}$$

Aktualisierung ab Block $k+1+p$ (Schritt $k+1$):

$$U_{k+1,k+1+p:nb:p} = L^{-1}_{k+1,k+1} A^{(k+1)}_{k+1,k+1+p:nb:p}$$

$$A^{(k+1)}_{*,k+1+p:nb:p} = A^{(k)}_{*,k+1+p:nb:p} - L_{*,k+1} U_{k+1,k+1+p:nb:p}$$

$k = k + 2$

ELSE

Permutation der Zeilen von A (Schritt k):

$$A^{(k)}_{ip:nb:p} = P^{(k)} A^{(k-1)}_{ip:nb:p}$$

Aktualisierung der lokalen Restmatrix (Schritt k):

$kp = \lceil k/p \rceil \cdot p + ip$

$$U_{k,kp:nb:p} = L^{-1}_{k,k} A^{(k)}_{k,kp:nb:p}$$

$$A^{(k)}_{*,kp:nb:p} = A^{(k)}_{*,kp:nb:p} - L_{*,k} U_{k,kp:nb:p}$$

$k = k + 1$

ENDIF

ENDWHILE

6.3 Lösungsprozedur

Mit der Zerlegung von A in P, L und U ist das lineare Gleichungssystem $Ax = b$ äquivalent zu den beiden Dreieckssystemen $Ly = P^T b$ (es ist $P^T = \prod_{k=nb}^{1} P^{(k)} = \prod_{j=n}^{1} P_j$ und $Ux = y$. Wenn bei der Faktorisierung eine Pivotsuche durchgeführt wurde, ist also zunächst $P^T b$ zu bilden, d.h. die rechte Seite b gemäß den Ergebnissen der Pivotsuche zu permutieren (dazu ist auf einem Local-Memory-Rechner wie SUPRENUM Kommunikation notwendig, wenn der Vektor b wie hier als verteilt vorausgesetzt wird). Anschließend können die beiden Dreieckssysteme nacheinander durch Vorwärts- bzw. Rückwärtseinsetzen gelöst werden; hierzu wird jeweils der Cyclic-Vectorsum-Algorithmus von Li und Coleman [25] in der passenden Variante benutzt. Berücksichtigt man die zyklisch- variable Spaltenblockaufteilung, so läßt sich der Algorithmus zur Lösung von $PLUx = b$ informell wie folgt beschreiben:

```
permutiere die rechte Seite
FOR k=1,nb DO
    IF (k-ter Block gehört mir) THEN
        empfange das Segment
        berechne die Lösungskomponenten des k-ten Blocks
        entferne den aktuellen Teil vom Segmentanfang
        aktualisiere das Segment
        berechne den neuen Teil des Segments
        füge den neuen Teil an das Segmentende an
```

```
            sende das Segment an den rechten Nachbarprozeß
            aktualisiere den Rest des lokalen Update-Vektors
        ENDIF
    ENDFOR
    FOR k = nb,1,-1 DO
        IF (k-ter Block gehört mir) THEN
            empfange das Segment
            berechne die Lösungskomponenten des k-ten Blocks
            entferne den aktuellen Teil vom Segmentende
            aktualisiere das Segment
            berechne den neuen Teil des Segments
            füge den neuen Teil an den Segmentanfang an
            sende das Segment an den linken Nachbarprozeß
            aktualisiere den Rest des lokalen Update-Vektors
        ENDIF
    ENDFOR
```

Das Segment ist ein Vektor, der so lang ist wie die nächsten $p-1$ Blöcke zusammen, und in dem die Update-Werte zur Berechnung der nächsten Lösungskomponenten akkumuliert werden. Dieses Segment wird in dem Prozeßring zyklisch herumgeschickt; dabei schneidet jeder Prozeß einen nicht mehr benötigten Teil ab und hängt einen neuen an. Daneben gibt es noch den lokalen Update-Vektor, in dem Werte aufsummiert werden, die erst bei späteren Durchläufen des Segments benötigt werden.

Soll statt $Ax = b$ das System mit transponierter Koeffizientenmatrix gelöst werden, also $A^T x = b$, so ist nach der Faktorisierung zunächst $U^T y = b$, dann $L^T x = y$ mit dem Cyclic-Scalarproduct-Algorithmus zu lösen und anschließend x gemäß den Ergebnissen der Pivotsuche zu permutieren, wobei die Transpositionen in umgekehrter Reihenfolge auszuführen sind (d.h. es ist Px zu bilden). Dieser Cyclic-Scalarproduct-Algorithmus ist in gewisser Weise dual zum Cyclic-Vectorsum-Algorithmus; allerdings entfällt hier der lokale Update-Vektor, und das Segment besteht jeweils aus den zuletzt berechneten Lösungskomponenten der vorhergehenden $p-1$ Blöcke.

Sollen mehrere lineare Gleichungssysteme mit gleicher Koeffizientenmatrix simultan gelöst werden, d.h. sind Matrizengleichungen $AX = B$ bzw. $A^T X = B$ zu lösen, so brauchen in den eben beschriebenen Algorithmen b und x nur durch B und X ersetzt werden, wobei letztere als zeilenverteilte Matrizen aufzufassen sind.

Die Permutation der zeilenverteilten rechten Seite(n) stellt auf einem System mit verteiltem Speicher ein Problem dar, weil zum einen dafür Kommunikation notwendig ist und zum anderen die notwendigen Vertauschungen datenabhängig und damit die angemessene Kommunikationsstruktur nicht vorhersagbar ist. In SLAP wurde dieses Pro-

blem so gelöst, daß die rechte(n) Seite(n) zunächst komplett auf einer Task - oBdA der letzten - eingesammelt wird (werden); diese Task Nr. p nimmt dann alle Vertauschungen vor, die sich bei der Faktorisierung durch die Pivotisierung ergeben haben (diese sind jedem Prozeß bekannt). Anschließend verschickt der letzte Prozeß die gesamte(n) permutierte(n) rechte(n) Seite(n) an die anderen Tasks; jede Task „vergißt" dann denjenigen Teil, der ihr nicht gehört. Dieses - wenig elegante - Vorgehen hat den Vorteil, daß es immer anwendbar und relativ einfach zu implementieren ist. Seine Nachteile liegen in der Möglichkeit, daß (viel) zu viele Daten verschickt werden, und in dem möglichen Engpaß beim letzten Prozeß. Um diese Nachteile etwas abzuschwächen, erfolgt das Einsammeln und Verteilen der rechten Seite(n) baumartig (siehe 4.5.2). Die eben beschriebene Permutationsroutine wird bei transponiertem Gleichungssystem nach dem Rückwärtseinsetzen aufgerufen, bei „normalem" Gleichungssystem jedoch vor dem Vorwärtseinsetzen.

Die Lösungsroutinen arbeiten also nach den bisher nur informell vorgestellten Algorithmen von Li und Coleman. Der folgende Pseudocode gibt den Cyclic-Vectorsum-Algorithmus (Vorwärtseinsetzen, eine rechte Seite) detaillierter wieder:

```
FOR k = 1,nb DO
    IF MOD(k − 1,p) + 1 = ip THEN
        kend = MIN(k + p − 2,nb)
        IF k ≠ 1 THEN
            RECEIVE s(k)k:kend FROM PROCESS# MOD(ip − 2,p) + 1
        ENDIF
        bk = Lk^-1 (bk − s(k)k − tk)
        IF k ≠ nb THEN
            s(k)k+1:kend = s(k)k+1:kend + tk−1:kend + Lk+1:kend,k bk
            IF k + p − 1 ≤ nb AND p > 1 THEN
                s(k)k+p−1 = tk+p−1 + Lk−p−1,k bk
            ENDIF
            SEND s(k)k+1:k−p−1 TO PROCESS# MOD(ip,p) + 1
            tk−p:nb = tk+p:nb + Lk+p:nb,k bk
        ENDIF
    ENDIF
ENDFOR
```

Hierbei wird b durch x überschrieben. Sollen mehrere rechte Seiten simultan behandelt werden, so sind nur die Felder b, s und ggf. t zu expandieren, d.h. als zweidimensional zu behandeln. Der Algorithmus für das Rückwärtseinsetzen ergibt sich analog.

6.4 Arithmetik- und Kommunikationsaufwand

Die Anzahl F der Gleitkommaoperationen für die übliche LU- Zerlegung einer $n \times n$-Matrix (Verwendung von BLAS 1-Modulen) ist bekannterweise

$$F_{fac} = \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n = O(n^3).$$

Geht man zu dem oben beschriebenen Blockalgorithmus basierend auf BLAS 3-Modulen über, so erhöht sich die Anzahl der erforderlichen Rechenoperationen nicht (bei anderen Blockalgorithmen kann der Rechenaufwand leicht ansteigen).

Auch bei der Lösungsprozedur stimmt die Anzahl der Rechenoperationen mit der des entsprechenden seriellen Algorithmus überein:

$$F_{sol} = 2n^2 - n = O(n^2).$$

(Um ein einzelnes Dreieckssystem zu lösen, werden n^2 Gleitkommaoperationen benötigt. Da die Matrix L auf der Diagonalen Einsen enthält, brauchen n Divisionen nicht ausgeführt zu werden.)

Der angegebene Wert für F_{sol} bezieht sich auf die Lösung eines „einfachen" Gleichungssystems. Bei mehreren rechten Seiten ist F_{sol} mit deren Anzahl zu multiplizieren.

Die Anzahl der erforderlichen SEND- und RECEIVE-Befehle für gegebene Werte von n und p exakt zu zählen, ist wesentlich schwieriger als die der arithmetischen Operationen, da der Kommunikationsaufwand stark von der Datenaufteilung abhängt und noch keine Angaben zu den Ergebnissen der angestrebten automatischen Verteilung (adaptive Blockung) gemacht werden können. Die im folgenden angegebenen Formeln beziehen sich daher auf die Anzahl p der eingesetzten Prozesse sowie auf die Anzahl nb der Blöcke, die später automatisch bestimmt werden wird.

Bei der Matrixfaktorisierung muß in jedem Blockschritt der jeweils zuständige Prozeß eine Nachricht bestehend aus den entsprechenden Teilen von P und L an seine $p-1$ „Kollegen" verschicken. Dies gilt auch für den letzten Block, da der zuständige Prozeß die anderen Prozesse über das Ergebnis der Pivotsuche informieren muß. Daher beträgt die (globale) Anzahl der Nachrichten

$$M_{fac} = nb(p-1).$$

Bei der Lösungsroutine müssen zunächst die Kommunikationsbefehle gezählt werden, die für die Permutation der rechten Seite anfallen. Das hierbei angewandte baumartige Vorgehen erfordert

$$M_{per} = 2(p-1)$$

Nachrichten.

Für das Vorwärts- und Rückwärtseinsetzen nach dem Cyclic-Vectorsum- oder Cyclic-Scalarproduct-Algorithmus beträgt die Anzahl der Messages jeweils

$$M_{sub} = nb - 1.$$

Damit ergibt sich für die gesamte Lösungsprozedur ein Kommunikationsaufwand von

$$M_{sol} = 2(p + nb - 2)$$

Nachrichten.

Anzumerken ist in diesem Zusammenhang, daß bei der Verwendung der Mehrfach-Lösungsroutine die Anzahl der verschickten Messages nicht ansteigt.

7.0 Das reelle symmetrische Eigenwertproblem

Zu einer reellen, vollbesetzten, symmetrischen $n \times n$-Matrix A sollen die Lösungen der Gleichung $Ax = \lambda x$, die sogenannten Eigenwerte λ_i und die zugehörigen Eigenvektoren x_i, $i = 1, \ldots, n$ gefunden werden.

Je nachdem, ob alle oder nur einige Eigenwerte und je nachdem, ob zusätzlich die zugehörigen Eigenvektoren zu bestimmen sind oder nicht, empfiehlt EISPACK [32] verschiedene Folgen von Unterprogrammaufrufen. Alle beginnen mit der Reduktion der Matrix auf Tridiagonalgestalt durch Ähnlichkeitstransformationen.
Danach werden entweder alle Eigenwerte und eventuell Eigenvektoren der Tridiagonalmatrix durch QL-Verfahren bestimmt, wobei die Eigenvektoren schon direkt mit der Transformationsmatrix aus dem ersten Schritt multipliziert werden, oder es werden einige Eigenwerte der Tridiagonalmatrix mit Bisektion bestimmt, danach durch inverse Iteration die Eigenvektoren der Tridiagonalmatrix, und schließlich werden diese Eigenvektoren auf die Eigenvektoren der ursprünglichen Matrix zurücktransformiert.

Für den Fall eines Parallelrechners sieht das Problem wie folgt aus:
Gegeben sei die reelle, vollbesetzte, symmetrische $n \times n$ Matrix A, deren Eigenwerte mit p Prozessen berechnet werden sollen. Bei einer spaltenorientierten Blockaufteilung oder bei zyklischer Aufteilung bedeutet dies, daß jeder Prozeß $k = n/p$ Spalten von A in seinem lokalen Speicher besitzt. Bei block-zyklischer Verteilung mit Blockgröße q gibt es insgesamt $\ell = n/q$ Spaltenblöcke, und somit besitzt jeder Prozeß $s = \ell/p = k/q$ Spaltenblöcke in seinem lokalen Speicher. Auch die parallele Version soll mit der Reduktion der Matrix auf Tridiagonalgestalt beginnen.

Wegen der einfacheren Parallelisierbarkeit des Bisektionsverfahrens gegenüber dem QL-Verfahren wurde auch für das Problem der Suche aller Eigenwerte und Eigenvektoren zunächst ein Bisektionsverfahren mit inverser Iteration und Rücktransformation der Eigenvektoren implementiert.
Zusätzlich wurde für die Berechnung aller Eigenwerte und Eigenvektoren reller symmetrischer Tridiagonalmatrizen das Divide and Conquer Verfahren von Cuppen [9] implementiert. Es beruht auf einem QL-Verfahren. Dieses kann unter bestimmten, später noch genauer erklärten Umständen, anstelle von Bisektion und inverser Iteration zwischen der Transformation auf Tridiagonalgestalt und der Rücktransformation der Eigenvektoren verwendet werden. Eine Rücktransformation der Eigenvektoren wird immer erforderlich sein, da eine explizite Berechnung der Transformationsmatrix nicht vorgesehen ist.

7.1 Reduktion einer vollbesetzten Matrix auf Tridiagonalgestalt

Im ersten Teil soll die Matrix A durch orthogonale Ähnlichkeitstransformationen auf Tridiagonalgestalt reduziert werden, d.h. es wird eine orthogonale Matrix Q gesucht, so daß $Q^TAQ = T$, wobei T eine symmetrische Tridiagonalmatrix ist.

7.1.1 Householdertransformation

Ein häufig verwendetes Verfahren zur Reduktion vollbesetzter Matrizen auf Tridiagonalgestalt ist die sogenannte Householdertransformation (s. z.B. [27]). Die Matrix wird hierbei spaltenweise Schritt für Schritt reduziert. Im i -ten Schritt, $i = 1, \ldots, n-2$, ist eine $(n-i+1) \times (n-i+1)$ -Untermatrix $\tilde{M}_i$ von A_i noch nicht tridiagonalisiert, während in den ersten $i-1$ Spalten und Zeilen von A_i nur noch auf der Diagonalen und der ersten Nebendiagonalen von Null verschiedene Einträge stehen.
Nun wird ein Vektor w_i der Länge $n-i$ und der Norm 1 bestimmt, so daß $Q_i c_i = (I - 2w_i w_i^T)c_i = \gamma_i \tilde{e}_i$, wobei c_i der Teil des ersten Spaltenvektors von $\tilde{M}_i$ ist, der in der zweiten Zeile beginnt, $\tilde{e}_i$ ist der erste Einheitsvektor e_1^{n-i} der Länge $n-i$, und γ_i ist ein Skalar.
Um A_{i+1} zu erhalten, muß die Transformation noch auf die restliche Matrix M_i, die untere $(n-i) \times (n-i)$ -Teilmatrix von A_i angewandt werden. Dabei ist folgendes zu berechnen:

$$\begin{aligned}
\tilde{M}_{i+1} &= (I - 2w_i w_i^T)^T M_i (I - 2w_i w_i^T) = M_i - (2w_i w_i^T)^T M_i - 2M_i w_i w_i^T + 4 (w_i w_i^T)^T M_i w_i w_i^T \\
&= M_i - 2w_i w_i^T M_i + \frac{1}{2} 4 w_i w_i^T M_i w_i w_i^T - 2M_i w_i w_i^T + \frac{1}{2} 4 w_i w_i^T M_i w_i w_i^T \\
&= M_i - \sqrt{2}\, w_i v_i^T - \sqrt{2}\, v_i w_i^T
\end{aligned}$$

mit $\quad v_i^T = z_i^T - \frac{1}{2}(z_i^T \sqrt{2}\, w_i)\sqrt{2}\, w_i^T \quad$ und $\quad z_i^T = \sqrt{2}\, w_i^T M_i$

Der bereits tridiagonalisierte Teil der Matrix bleibt unverändert, ebenso wie das i-te Diagonalelement, und das i-te Nebendiagonalelement wird zu γ_i.

Es gilt dann $Q_i^T A_i Q_i = A_{i+1}$ mit $A_1 = A$ und $A_n = T$. Die Gesamttransformationsmatrix Q berechnet sich als $Q = Q_1 \cdot Q_2 \cdot \ldots \cdot Q_n$.

7.1.2 Block-Householdertransformation

Bei der Block-Householdertransformation wird die Matrix A zuerst in eine $\ell \times \ell$ Blockmatrix mit Blöcken der Dimension $q \times q$ aufgeteilt. Sie wird dann blockspaltenweise zu einer Bandmatrix mit Bandbreite q reduziert. Diese muß schließlich noch durch ein anderes Verfahren, z.B. das Verfahren von Schwarz (s. [31]), tridiagonalisiert werden.

Die Block-Householdertransformation erfolgt analog zur gewöhnlichen Householdertransformation.
Im l-ten Blockschritt, $l = 1, \ldots, \ell-2$, ist noch eine $(\ell - l + 1) \times (\ell - l + 1)$ -Blockmatrix $\tilde{M}_\ell$ zu reduzieren. Hierfür wird eine QR-Zerlegung der Blockspalte unterhalb des ersten

Diagonalblocks der Restmatrix vorgenommen. Die orthogonalen Matrizen Q_l dieser Zerlegung, welche Produkte von q Householdermatrizen sind, werden in der WY-Form dargestellt, die in [4] beschrieben ist, d.h. $Q_l = I + W_l Y_l^T$, wobei W_l und Y_l Matrizen der Dimension $(\ell - l)q \times q$ sind. Die Matrix $I + W_l Y_l^T$ muß dann auf die untere $(\ell - l) \times (\ell - l)$ Blockteilmatrix M_l von A_l angewandt werden, und es wird analog der Householdertransformation

$$\tilde{M}_{l+1} = (I + W_l Y_l^T)^T M_l (I + W_l Y_l^T) = M_l + Y_l V_l^T + V_l Y_l^T$$

mit $\quad V_l^T = Z_l^T - \frac{1}{2} Z_l^T W_l Y_l^T \quad$ und $\quad Z_l^T = W_l^T M_l$

berechnet. Der erste Diagonalblock von $\tilde{M}_l$ bleibt als l-ter Diagonalblock in A_{l+1} erhalten, die Matrix R von der QR-Zerlegung wird zum l-ten Nebendiagonalblock.

Es gilt dann analog zum Fall der gewöhnlichen Houscholdertransformation $Q_l^T A_l Q_l = A_{l+1}$ mit $A_1 = A$ und $A_e = T$. Die Gesamttransformationsmatrix Q berechnet sich als $Q = Q_1 \cdot Q_2 \cdot \ldots \cdot Q_e$.

7.1.3 Tridiagonalisierung einer Bandmatrix mit dem Schwarz-Verfahren

Das Schwarz-Verfahren arbeitet so, daß durch Givens-Rotationen nacheinander die Nebendiagonalelemente bis auf die der ersten Nebendiagonale eliminiert werden. Dabei entsteht jeweils ein Fill-in, das wiederum durch Givens-Rotation „weitergeschoben" wird, bis es „über den Rand der Matrix hinaus geschoben" ist (s. Abb. 11).

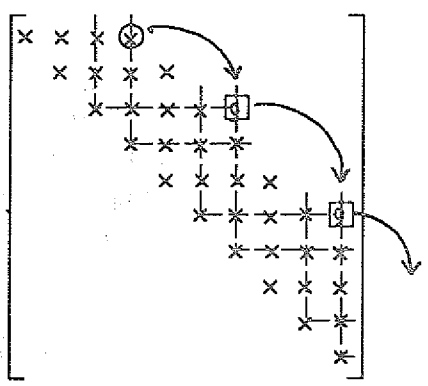


Abb. 11. Das Verfahren von Schwarz

Man kann entweder zeilenweise von außen nach innen die Nebendiagonalelemente eliminieren oder diagonalenweise von oben nach unten. Die erste Methode benötigt weniger Rechenoperationen, da die Fill-ins jeweils um so viele Zeilen nach unten wandern, wie die aktuelle Anzahl der Nebendiagonalen beträgt. Diese bleibt aber im ersten Fall konstant für das noch zu bearbeitende Stück der Matrix, während sie im zweiten Fall sukzessive abnimmt.

Leider wurde für Parallelrechner mit verteiltem Speicher noch keine effiziente Parallelisierung des Verfahrens von Schwarz gefunden. Wollte man die Rotationen auf

die spaltenblockweise verteilte Matrix anwenden, so müßte immer dort Kommunikation erfolgen, wo ein Fill-in über eine Blockgrenze hinweg geschoben werden muß. Selbst wenn es, wie bei der sukzessiven Reduktion der Bandbreite, möglich ist, mehrere Fill-ins zugleich zu behandeln und weiterzureichen, so sind doch Kommunikationsaufrufe in der Größenordnung von $n \cdot p$ erforderlich. Diese können zwar teilweise parallel zu anderen Kommunikationsaufrufen oder zu Berechnungen stattfinden, der Gesamtgrad an Parallelität ist jedoch sehr gering.

7.1.4 Parallelisierung der Householder- und Block-Householdertransformation

In diesem Abschnitt werden die Parallelisierungsmöglichkeiten der beiden Verfahren bei verschiedenen Aufteilungen der Matrix beschrieben und der jeweilige Kommunikations- und Rechenaufwand abgeschätzt. In einer tabellarischen Übersicht werden die verschiedenen Methoden schließlich miteinander verglichen.

Anders als bei Zerlegungsroutinen, bei denen die Restmatrix nur von links mit einer orthogonalen Matrix multipliziert wird, muß bei der Transformation die Restmatrix von links und rechts multipliziert werden. Dies kann nicht lokal in jedem Prozeß geschehen, da für die Multiplikation von rechts vollständige Zeilen der Matrix gebraucht werden, diese aber über alle Prozesse verteilt sind. Es ist auch nicht so einfach wie bei der Zerlegung, die Behandlung der Spalte, aus der im nächsten Schritt der Householdervektor berechnet werden soll, vorzuziehen. Ansätze für eine solche Vorgehensweise sind in [15] beschrieben. Im folgenden soll jedoch immer ein Schritt bzw. Blockschritt zu Ende geführt werden, ehe der nächste beginnt.

Bei der Kommunikation auf dem SUPRENUM Rechner ist zu beachten, daß die Übertragungszeiten für einzelne Daten (t_β) sehr klein sind im Vergleich zu den Initialisierungszeiten für einen SEND und RECEIVE Befehl (t_γ und t_δ). Daher ist in erster Näherung die Zahl der zu sendenden Daten vernachlässigbar gegenüber der Initialisierung einer Kommunikation, sofern die Größenordnung nicht im Bereich von einigen Tausend Daten liegt. Dies ist bei den Problemen aus der Linearen Algebra nicht der Fall, daher wird im folgenden bei der Abschätzung des Kommunikationsaufwands nur die Anzahl der Kommunikationsaufrufe gezählt, nicht aber die Menge der verschickten Daten.
Wie im Kapitel „Kommunikationsstrategien" bereits erwähnt, wird außerdem meist ein vereinfachtes asynchrones Kommunikationsmodell angenommen mit $t_\gamma = t_\delta = t_\alpha/2$ und $t_\beta = 0$.

7.1.4.1 Parallelisierung bei Blockaufteilung der Matrix

Hier ergibt sich, wie bereits bei Zerlegungsroutinen erwähnt, der Nachteil einer schlechten Lastverteilung. Nachdem die ersten k Spalten oder die ersten s Blockspalten bearbeitet sind, wird an diesen nichts mehr verändert, und der erste Prozeß hat keine Arbeit mehr.
Außerdem bleibt die Anzahl der parallel ausgeführten Operationen pro Schritt bzw. Blockschritt bis zu den letzten k Schritten konstant, da immer wenigstens ein Prozeß noch k Spalten bearbeiten muß. Die Anzahl der arithmetischen Operationen, die parallel ausgeführt werden, wird in Tabelle 4 verglichen.

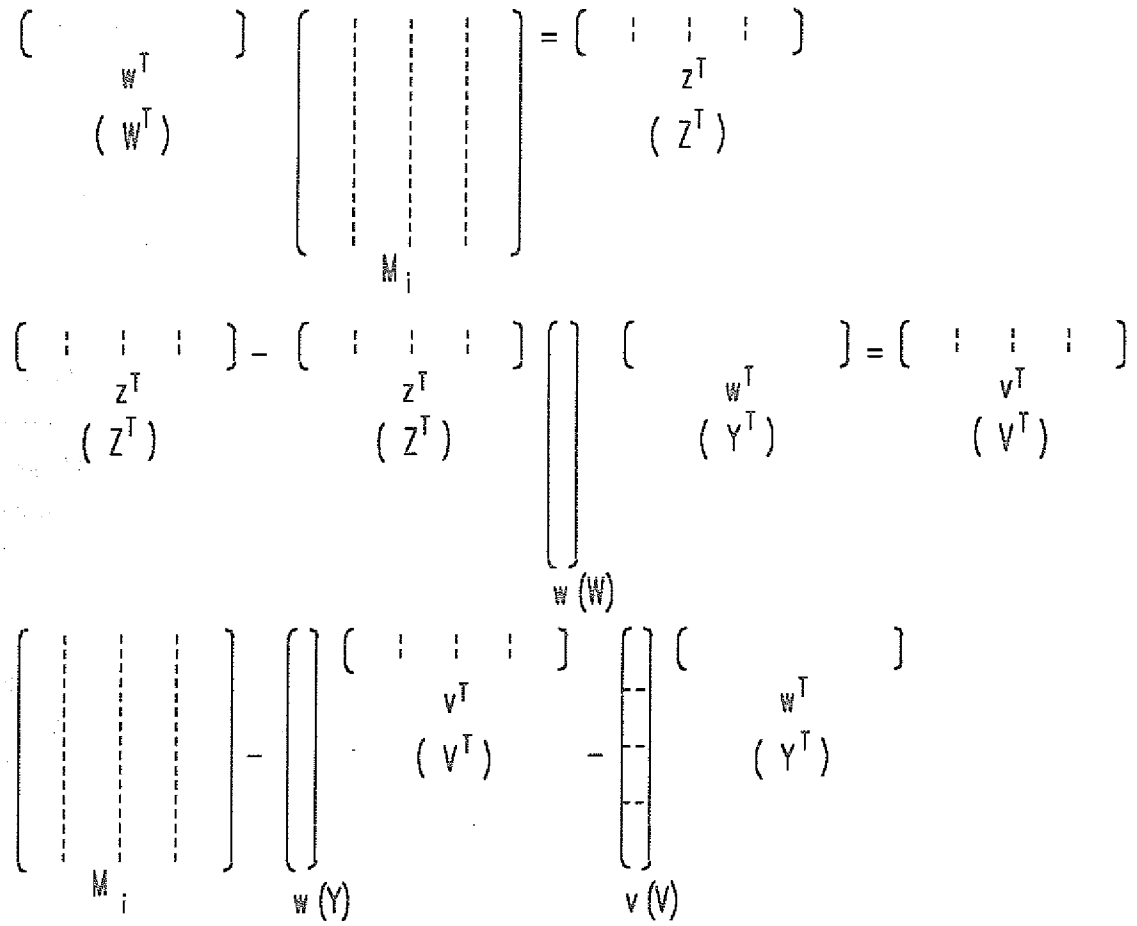


Abb. 12. Die Verteilung der Matrix auf die Prozesse während der Berechnung: Nicht unterteilte Vektoren und Matrizen haben alle Prozesse ganz in ihrem Speicher, von den unterteilten Objekten besitzt jeder Prozeß nur einen Teil.

Sei nun also die Matrix in zusammenhängenden Blöcken auf die Prozesse verteilt, wobei für das Block-Householderverfahren die Anzahl der Spalten, die die Prozesse 1 bis $p-1$ bekommen durch q teilbar sein soll, damit die Grenzen der Blockspalten mit den Prozeßgrenzen zusammenfallen.

Im folgenden wird nur die Parallelisierung des gewöhnlichen Householderverfahrens beschrieben. Das Vorgehen beim Block-Householderverfahren ergibt sich durch Ersetzen von Schritt durch Blockschritt, Spalte durch Blockspalte, Element durch Block, Skalarprodukt durch Produkt einer Zeilenmatrix mit einer Spaltenmatrix; w wird jeweils durch W oder Y ersetzt und v und z durch V und Z.

Im i-ten Schritt ist der Prozeß, der die i-te Spalte der Matrix in seinem Speicher besitzt, der erste, der noch arbeitet. Er berechnet den Householdervektor w und schickt ihn zu allen anderen noch aktiven Prozessen. Mit dem im Kapitel „Kommunikationsstrategien" als zweites beschriebenen Softwarebroadcast ist hierfür die Zeit von $\lceil \log_2 j \rceil$ Kommunikationsaufrufen nötig, wenn j die Anzahl der noch aktiven Prozesse ist. Jeder dieser Prozesse besitzt k Spalten m_κ, $\kappa = (jp-1)\cdot k + 1, \ldots, jp\cdot k$ (wenn dies der Prozeß jp ist)

von M_i und berechnet daraus und aus dem empfangenen Vektor w lokal k Elemente $z_\kappa^T = \sqrt{2}\, w^T m_\kappa$, $\kappa = (jp-1)\cdot k + 1, \ \ldots\ , jp\cdot k$ des Vektors z^T.
Zur Berechnung seines Teils von v^T braucht jedoch jeder Prozeß das Skalarprodukt $z^T w$, und da der Vektor z^T auf alle aktiven Prozesse verteilt ist (s. Abb. 12), ist hierfür Kommunikation erforderlich.
Analog zum Sammeln eines verteilten Vektors könnten die Teilsummen des Skalarprodukts, die jeder Prozeß lokal berechnen kann, gesammelt werden bis schließlich jeder der j Prozesse nach $\lceil \log_2 j \rceil$ Kommunikationsaufrufen das Skalarprodukt in seinem lokalen Speicher hat.
Leider kennt nach dieser Berechnung jeder Prozeß nur einen Teil von v^T bzw. v, da er nur einen Teil von z^T kannte. Er kann somit im nächsten Schritt nur einige Zeilen von vw^T berechnen (s. Abb. 12). Da vw^T nicht symmetrisch ist, kann somit kein Prozeß ganze Spalten von vw^T berechnen, also wäre nochmals Kommunikation nötig, damit jeder Prozeß seinen Spaltenblock von vw^T berechnen kann, den er zur Berechnung seines Spaltenblocks von $\tilde{M}_{i+1}$ braucht.
Um diese erneute Kommunikation zu vermeiden, ist es sinnvoller, gleich jedem Prozeß ganz z^T bekannt zu machen, so daß er ganz v berechnen kann, was nur wenige zusätzliche arithmetische Operationen kostet. (Jeder Prozeß berechnet ein Skalarprodukt der Länge n und ein Produkt Skalar · kompletter Vektor statt eines Teils, der Mehraufwand ist also $O(n)$ arithmetische Operationen.) Damit hat jeder Prozeß alle nötigen Daten, um seinen Spaltenblock von vw^T zu berechnen.
Dieses Sammeln kann, wie im Kapitel „Kommunikationsstrategien" erklärt, durch Weitersenden eines immer größeren Teils von z^T geschehen. Dazu müssen alle j noch aktiven Prozesse parallel je $\lceil \log_2 j \rceil$ Kommunikationsaufrufe durchführen.

Für die gesamte Householdertransformation wird damit in den ersten k Schritten die Zeit für $2\lceil \log_2 p \rceil$, in den nächsten k Schritten die für $2\lceil \log_2 (p-1) \rceil$ Kommunikationsaufrufe benötigt usw.. In den letzten k Schritten ist schließlich keine Kommunikation mehr nötig, da nur noch ein Prozeß arbeitet.
Zusammen ergibt dies einen Kommunikationsaufwand, der der Zeit für

$$k \sum_{j=1}^{p-1} 2 \lceil \log_2 j \rceil \simeq 2k\, p\, (\lceil \log_2 p \rceil - 1)$$

$$\simeq 2\, n\, (\lceil \log_2 p \rceil - 1), \quad (\text{da } k\cdot p = n)$$

Kommunikationsaufrufe entspricht. Schon bei einem relativ kleinen Problem mit $n = 100$ und $p = 4$ ergibt das 350 Kommunikationsaufrufe, mit $p = 5$ sogar schon 560, da $\lceil \log_2 5 \rceil = 3$.

Bei der Block-Householdertransformation werden in jedem Blockschritt die Matrizen W und Y vom ersten noch aktiven Prozeß per Software-Broadcast an alle anderen noch aktiven Prozesse gesendet, was ebenfalls $\lceil \log_2 j \rceil$ Kommunikationsaufrufe erfordert.

Anschließend schickt jeder noch aktive Prozeß an alle anderen noch aktiven Prozesse seinen Teil der Matrix Z, was wiederum mit $\lceil \log_2 j \rceil$ SEND-RECEIVE-Befehlen parallel in allen Prozessen verbunden ist.
Dies bedeutet, daß beim Block-Householderverfahren in einem Blockschritt ebenso viele SEND-RECEIVE-Kommandos benötigt werden wie beim gewöhnlichen Householderverfahren in einem Schritt. Die Anzahl der Kommunikationsaufrufe ist also beim Block-Householderverfahren um den Faktor q kleiner als beim gewöhnlichen, da es nur n/q Blockschritte im Gegensatz zu n Schritten gibt. Im obigen Beispiel ergibt dies für $q = 5$ statt 350 (560) nur 70 (112) Kommunikationsaufrufe.

Zusätzlich muß sich aber an die Block-Householdertransformation noch eine Reduktion der Bandmatrix auf Tridiagonalgestalt anschließen. Wie oben erwähnt geschieht das mit dem sequentiellen Schwarz-Verfahren. Am günstigsten erweist es sich, wenn zuerst die Bandmatrix mit der Sammelroutine allen Prozessen bekannt gemacht wird und dann alle Prozesse die Reduktion gleichzeitig durchführen. Dazu werden wieder $\lceil \log_2 p \rceil$ Kommunikationsaufrufe benötigt, was aber an der Gesamtsumme der Kommunikationsaufrufe wenig ändert.
Der Vorteil dieses Verfahrens liegt darin, daß dann ohne weitere Kommunikation jeder Prozess die komplette Tridiagonalmatrix kennt, was später für das Bisektionsverfahren notwendig ist.
Die Anzahl der Kommunikationsaufrufe bei den verschiedenen Verfahren wird in Tabelle 2 verglichen.
Außer den zusätzlichen arithmetischen Operationen für das sequentielle Schwarz-Verfahren sind beim Block-Householderverfahren noch die Matrizen W und Y zusätzlich zum jeweils aktuellen Householdervektor der QR-Zerlegung einer Blockspalte von einem Prozeß alleine zu berechnen. Im Gegensatz hierzu wird beim gewöhnlichen Householderverfahren nur die Berechnung der Householdervektoren w jeweils von einem Prozeß alleine durchgeführt, also nicht parallelisiert. Die arithmetischen Operationen, die nicht parallel ausgeführt werden, werden in Tabelle 3 verglichen.

7.1.4.2 Parallelisierung bei (block)-zyklischer Aufteilung der Matrix

Anders als bei der Blockaufteilung sind bei der zyklischen bzw. block-zyklischen Aufteilung der Matrix bis zum Schluß alle Prozesse aktiv und somit auch an der Kommunikation beteiligt.
Die Anzahl der Spalten bzw. Blockspalten, die jeder Prozeß zu bearbeiten hat, nimmt nach jeweils p Schritten (Blockschritten) um eine ab. Dies bedeutet, daß die Anzahl der arithmetischen Operationen, die parallel ausgeführt werden können, kontinuierlich abnimmt (s. Tabelle 4).

Die Berechnung der Householdervektoren und der Matrizen W und Y wird, wie bei der Blockaufteilung der Matrix auch, jeweils von einem Prozeß durchgeführt, nur lösen die Prozesse einander bei jedem Schritt ab. Anders als bei der QR-Zerlegung (s. [2]) ergibt sich hieraus jedoch bei der Transformation nur ein sehr geringer zusätzlicher Gewinn durch "Compute-and-Send-Ahead". Der Prozeß, der als nächster den Housholdervektor berechnen muß bzw. die QR-Zerlegung der Blockspalte vornehmen muß, kann dies erst tun, wenn er v bzw. V bereits berechnet hat. Dann muß er noch die erste (die ersten q) Spalte(n) von $wv^T + vw^T$ $(YV^T + VY^T)$ berechnen, ehe er ein neues w (neue W und Y)

berechnen und den anderen Prozessen senden kann. Erst danach berechnet er den Rest von $wv^T + vw^T$ $(YV^T + VY^T)$.
So lange jeder Prozeß noch mehrere (Block-)Spalten von M_i zu aktualisieren hat, können dadurch die Zeiten, während derer die Prozesse auf w (W und Y) warten, verkürzt oder sogar ganz vermieden werden. Anschließend an das Senden von w (W und Y) muß jedoch der sendende Prozeß den Rest seines Anteils von M_i aktualisieren, ehe er mit dem neuen w (W und Y) seinen Anteil des neuen z (Z) berechnen und weitersenden kann. Die übrigen Prozesse beginnen direkt nach dem Empfangen des neuen w (der neuen W und Y) mit der Berechnung des neuen z (Z). Da dieses z (Z) bei allen Prozessen gesammelt werden muß, entsteht jetzt hier ein Synchronisationspunkt, bei dem die übrigen Prozesse auf den Prozeß warten müssen, der mehr arithmetische Operationen ausführen mußte, also auf den, der w (W und Y) berechnet hat. Also entsteht nun die Verzögerung beim Sammeln von z (Z). Man kann davon ausgehen, daß diese Verzögerung nur wenig kleiner ist als die, die sich ohne die vorweggenommene QR-Zerlegung ergeben hätte. Der nicht parallelisierte Rechenaufwand ist also bei zyklischer Aufteilung fast der gleiche wie bei Blockaufteilung der Matrix (s. Tabelle 3).
Da das Sammeln von z bzw. Z ohnehin eine Synchronisation aller Prozesse erfordert, wird hier durch das stufenweise Broadcasting keine zusätzliche Synchronisation eingeführt. Am besten ist hier wahrscheinlich das Senden über einen Binärbaum, da hier der Prozeß, der wegen der Berechnung von w die meisten arithmetischen Operationen auszuführen hat, weniger Zeit mit Senden zubringt als bei den anderen Broadcastversionen. Für SUPRENUM wurde dennoch die als zweite beschriebene Version des Broadcasts implementiert, da für das Broadcasting mit Binärbaum noch keine Routine geschrieben ist.

Die Kommunikationsstruktur ist ähnlich wie bei der Blockaufteilung der Matrix. Der Prozeß, der den Householdervektor bestimmt hat, schickt diesen jetzt per Broadcast an alle anderen Prozesse, da jeder noch Spalten der Matrix M_i in seinem Speicher besitzt. Nachdem jeder Prozeß seinen Anteil des Vektors z berechnet hat, wird z bei allen Prozessen gesammelt. Außer in den letzten p Schritten sind somit in jedem Schritt $2 \cdot \lceil \log_2 p \rceil$ Kommunikationsaufrufe erforderlich.

7.1.4.3 Vergleich der verschiedenen Parallelisierungen

In den folgenden Tabellen geben die Zahlen jeweils die Größenordnung des Kommunikations- und des Rechenaufwandes wieder.
Unter einer arithmetischen Operation ist jeweils eine Multiplikation oder eine Addition zu verstehen. Hinzu kommen noch einzelne Divisionen und Wurzeloperationen, die jedoch an der Größenordnung des arithmetischen Aufwands nichts ändern.

Kommunikation	normale Householder-transformation	Block-Householdertransformation
Kommunikationsaufrufe pro Schritt bei Blockaufteilung	$2 \lceil \log_2 j \rceil$	$2 \lceil \log_2 j \rceil$
Anzahl Schritte bei Block-aufteilung	k Schritte für jedes $j = 1, \ldots, p$	k/q Schritte für jedes $j = 1, \ldots, p$
Kommunikationsaufrufe insgesamt bei Blockaufteilung	$\simeq 2n\,(\lceil \log_2 p \rceil - 1)$	$\simeq 2\frac{n}{q}\,(\lceil \log_2 p \rceil - 1)$
Aufrufe pro Schritt bei (block)-zyklischer Aufteilung	$2 \lceil \log_2 p \rceil$	$2 \lceil \log_2 p \rceil$
Anzahl Schritte bei (block)-zyklischer Aufteilung	n	$\frac{n}{q}$
Aufrufe insgesamt bei (block)-zyklischer Aufteilung	$\simeq 2n \lceil \log_2 p \rceil$	$\simeq 2\frac{n}{q} \lceil \log_2 p \rceil$

Tab. 2. Vergleich des Kommunikationsaufwands: Kommunikationsaufrufe können von verschiedenen Prozessen parallel ausgeführt werden.

nicht parallele Berechnung	normale Householder-transformation	Block Householder-transformation
Arithm. Op. pro Schritt	$3\,(n - i)$	$4\,(n - (l-1)q)\,(q + q^2)$
Anzahl Schritte	$i = 1, \ldots, n$	$l = 1, \ldots, \ell = n/q$
Schwarz-Verfahren	entfällt	$6\,q\,n^2 + 13\,n^2$
Arithm. Op. insgesamt	$\simeq \frac{3}{2}\,n^2$	$\simeq (8q + 15)\,n^2$
Diese Werte gelten sowohl für Blockaufteilung als auch für zyklische bzw. block-zyklische Aufteilung		

Tab. 3. Vergleich des arithmetischen Aufwands der nicht parallelen Berechnung: Wartezeiten für alle außer einem Prozeß.

parallele Berechnung	normale Householder-transformation	Block Householder-transformation
Arithm. Op. pro Schritt bei Blockaufteilung	$6\,k\,(n - i)$	$(6q + 2)\,k\,(n - l \cdot q)$
Anzahl Schritte bei Block-aufteilung	$i = 1, \ldots, n$	$l = 1, \ldots, \ell = \frac{n}{q}$
Arithm. Op. insgesamt bei Blockaufteilung	$\simeq 3\,k\,n^2$	$\simeq 3\,k\,n^2\,(1 + \frac{1}{3q})$
Arithm. Op. pro Schritt bei (block)-zyklischer Aufteilung	$6\,(k - \kappa)\,(n - i)$	$(6q + 2)\,(k - \kappa \cdot q)\,(n - l \cdot q)$
Anzahl Schritte bei (block)-zyklischer Aufteilung	$i = 1, \ldots, n\ ;\ \kappa = \frac{i}{p}$	$l = 1, \ldots, \ell\ ;\ \kappa = \frac{l}{p}$
Arithm. Op. insgesamt bei (block)-zyklischer Aufteilung	$\simeq 2k\,n^2$	$\simeq 2k\,n^2\,(1 + \frac{1}{3q})$

Tab. 4. Vergleich des arithmetischen Aufwands der parallelen Berechnung

Der Vergleich der Tabellen zeigt, daß zum Beispiel der Gewinn bei der Kommunikation beim Block-Housholderverfahren einen erheblichen Mehraufwand bei der nicht parallelisierten Berechnung nach sich zieht. Einem geringeren arithmetischen Aufwand im parallelen Teil bei der zyklischen bzw. block-zyklischen Aufteilung der Matrix steht ein um $2k$ höherer Kommunikationsaufwand bei diesen Aufteilungen gegenüber. Um also Aussagen treffen zu können, welches Verfahren schneller sein wird, muß das Verhältnis der Zeit zur Initialisierung eines Sende- und eines Empfangsvorgangs, $T_{Kom} = t_s + t_\delta$, zu der Zeit für eine Multiplikation oder eine Addition, T_{Ar}, berücksichtigt werden. Da dieses Verhältnis für SUPRENUM noch nicht bekannt ist, können nur theoretische Überlegungen angestellt werden, bei welchem Verhältnis welches Verfahren vorzuziehen ist.
Durch Vergleich des jeweiligen Gesamtaufwandes für die Transformation auf Tridiagonalgestalt, also Zeit für parallelisierte Berechnungen plus Zeit für nicht parallelisierte Berechnungen plus Zeit für Kommunikation, stellt sich heraus, daß bei fast allen Kombinationen von n, p und q das gewöhnliche Householderverfahren mit zyklischer Aufteilung der Matrix am schnellsten sein müßte.

Mit den Bezeichnungen T_{Ar} und T_{Kom} wie oben beschrieben ergibt sich, daß das gewöhnliche Householderverfahren mit zyklischer Aufteilung der Matrix schneller ist als

- das Block-Householderverfahren mit block-zyklischer Aufteilung der Matrix, wenn

$$T_{Kom} \leq \frac{(\frac{2}{3}k + 8q^2 + 13{,}5q)\,n}{2\,(q-1)\,\lceil \log_2 p \rceil}\,T_{Ar}$$

- das Block-Householderverfahren mit kontinuierlicher Aufteilung der Matrix, wenn

$$T_{Kom} \leq \frac{((q+1)k + 8q^2 + 13{,}5q)\,n}{2\,((q-1)\,\lceil \log_2 p \rceil + 1)}\,T_{Ar}$$

- das gewöhnliche Householderverfahren mit kontinuierlicher Aufteilung der Matrix, wenn

$$T_{Kom} \leq \frac{k \cdot n}{2}\,T_{Ar}$$

Für große n sind die beiden letzten Ungleichungen erfüllt, wenn die erste erfüllt ist. Ist jedoch $k\,((q-1)\,\lceil \log_2 p \rceil - \frac{2}{3}) < 8q^2 + 13{,}5q$, so ist die letzte Abschätzung schärfer als die beiden anderen.
In obigem Beispiel mit $n = 100$, $p = 4$, $k = 25$, und $q = 5$ ist das gewöhnliche Householderverfahren mit zyklischer Aufteilung der Matrix vorzuziehen, wenn $T_{Kom} \leq 1\,250 T_{Ar}$, das gewöhnliche Householderverfahren mit kontinuierlicher Aufteilung wäre das schnellste Verfahren, wenn $1\,250 T_{Ar} \leq T_{Kom} \leq 2\,660 T_{Ar}$, das Block-Householderverfahren mit block-zyklischer Aufteilung der Matrix wäre schneller, wenn $2\,660 T_{Ar} \leq T_{Kom} \leq 6\,250 T_{Ar}$, und das Block-Householderverfahren mit kontinuierlicher Aufteilung wäre am schnellsten, wenn $T_{Kom} > 6\,250 T_{Ar}$ wäre. Für größere n in der Größenordnung von 1 000 oder mehr, gilt nahezu unabhängig von p und q, daß das gewöhnliche Householderverfahren mit zyklischer Aufteilung der Matrix am schnellsten ist, wenn nur $T_{Kom} \leq 5\,000 T_{Ar}$. Dies dürfte wohl für alle Parallelrechner mit Botschaftenaustausch für die Kommunikation gegeben sein.

Alle Abschätzungen des Aufwandes berücksichtigen nicht eine explizite Berechnung der Transformationsmatrix Q. Bei der gewöhnlichen Householdertransformation kann diese auch nachträglich berechnet werden, wenn alle Householdervektoren w gespeichert wurden. Bei der Block-Householdertransformation ist nur die Transformationsmatrix Q_B mit $Q_B^T A Q_B = B$, B symmetrische Bandmatrix der Bandbreite q, aus den gespeicherten Matrizen W und Y rekonstruierbar. Um Eigenvektoren der vollbesetzten Matrix zu bestimmen, müssen hier die Eigenvektoren der Bandmatrix durch inverse Iteration bestimmt und dann mit Q_B auf die der vollbesetzten Matrix zurücktransformiert werden (s. [20]).

In dem beschriebenen Programmpaket wird die Rücktransformation der Eigenvektoren wieder schrittweise mit Hilfe der w bzw. der W und Y durchgeführt. Die Matrix Q_B wird nirgendwo explizit berechnet.

7.2 Bestimmung der Eigenwerte und Eigenvektoren einer symmetrischen Tridiagonalmatrix

Gegeben seien die Diagonale D und die Nebendiagonale E einer symmetrischen Tridiagonalmatrix T der Dimension n , deren Eigenwerte und eventuell auch Eigenvektoren bestimmt werden sollen. Zur Bestimmung der Eigenwerte gibt es unterschiedliche Verfahren, von denen zwei in einer parallelen Variante vorgestellt werden sollen.
Die Auswahl des Verfahrens hängt unter anderem davon ab, ob alle oder nur einige Eigenwerte und Eigenvektoren gesucht sind.

7.2.1 Berechnung der Eigenwerte mit Bisektion

Die Bisektionsmethode beruht auf einem "Spectrum-Slicing" Verfahren ähnlich dem Prinzip der Sturmschen Ketten und ist in [1] ausführlich beschrieben.
Sollen alle Eigenwerte berechnet werden, so wird bei dem für SUPRENUM implementierten Verfahren zuerst mit Gerschgorin-Kreisen ein reelles Intervall bestimmt, in dem alle Eigenwerte liegen. Alternativ kann auch ein Intervall vorgegeben sein, so daß alle Eigenwerte, die in diesem Intervall liegen, bestimmt werden sollen. In diesem Fall wird zuerst die Zahl m der Eigenwerte in diesem Intervall bestimmt. Diese ist einfach festzustellen, da die Berechnung der Sturmschen Ketten im Anfangs- und Endpunkt des Intervalls feststellt, wie viele Eigenwerte kleiner als der jeweilige Punkt sind.
Das weitere Vorgehen ist wie folgt:
Das Intervall wird halbiert, am Unterteilungspunkt wird die Sturmsche Kette berechnet, daraus wird festgestellt, in welchem der beiden Teilintervalle Eigenwerte liegen. Mit dem (den) Teilintervall(en), in dem (denen) Eigenwerte liegen, wird wieder wie beschrieben verfahren, bis die Länge eines nicht leeren Teilintervalls kleiner als eine vorgegebene Genauigkeitsschranke ist. Der Mittelpunkt dieses Intervalls wird als Eigenwert akzeptiert.

7.2.2 Verschiedene Parallelisierungsmöglichkeiten der Bisektion

Das Bisektionsverfahren läßt sich leicht auf verschiedene Arten parallelisieren. Bei allen Versionen ist es nötig, daß jeder Prozeß die gesamte Tridiagonalmatrix kennt, da diese zur Berechnung einer Sturmschen Kette nötig ist. Die Verteilung der berechneten Eigenwerte auf die einzelnen Prozesse ist bei den verschiedenen Parallelisierungen unterschiedlich.
Um eine Auswahl zwischen den verschiedenen Verfahren zu ermöglichen und dennoch eine wohldefinierte Schnittstelle zu haben, ist es sinnvoll, für die Ausgabe einer Bisektionsroutine allen Prozessen alle Eigenwerte bekannt zu machen, was bei den Master- Slave Programmen durch ein Broadcasting und sonst durch einen Sammelprozeß geschieht. In beiden Fällen sind dafür $\lceil \log_2 p \rceil$ Kommunikationsaufrufe nötig, wie im Kapitel „Kommunikationsstrategien" beschrieben.

Eine Methode, die zu einer besonders guten Lastverteilung führt, ist die folgende:
Ein Master-Prozeß teilt das ganze Intervall in p Teile (das ergibt $p-1$ Teilungspunkte), jeder der $p-1$ Slave-Prozesse berechnet die Sturmsche Kette in einem Teilungspunkt

und sendet das Ergebnis dem Master. Dieser stellt fest, wie viele Intervalle, die Eigenwerte enthalten, er jetzt kennt. Sind dies mehr als $p-1$, so halbiert er $p-1$ davon und sendet jedem Slave einen Teilungspunkt zur Berechnung der Sturmschen Kette. Sind weniger als $p-1$ Intervalle gefunden, so werden einige in mehr als zwei Teile geteilt, so daß wieder $p-1$ Teilungspunkte entstehen. Der Master stellt fest, ob die Intervalle klein genug sind und bricht für diese Intervalle dann die Berechnung ab. Damit kennt der Master-Prozeß alle Eigenwerte und kann sie per Broadcast allen anderen Prozessen bekannt machen.
Diese Methode braucht ebenso viele SEND-RECEIVE-Kommandos wie Sturmsche Ketten insgesamt zu berechnen sind, also p mal so viele Kommunikationsaufrufe wie parallele Berechnungen von Sturmschen Ketten. Da die Berechnung der Sturmschen Kette nur $O(n)$ arithmetische Operationen benötigt, kann diese Methode nur für sehr große n sinnvoll sein.

Eine andere Methode, die den Kommunikationsaufwand minimiert, ist folgende:
Alle Prozesse teilen das Gesamtintervall in p Teilintervalle auf, und jeder Prozeß berechnet alle Eigenwerte in einem Teilintervall. Dies erfordert gar keine Kommunikation für die Berechnung, nur für das Sammeln der Ergebnisse sind $\lceil \log_2 p \rceil$ Kommunikationsaufrufe erforderlich.
Andererseits ist die Lastverteilung bei dieser Methode davon abhängig, wie die Eigenwerte im Spektrum verteilt sind. Ist die Verteilung gleichmäßig, so ist auch die Last gleichmäßig verteilt, sind die Eigenwerte jedoch überwiegend in einem Teilintervall, so rechnet lange Zeit nur noch der Prozeß, der dieses Intervall bearbeitet.

Als weitere Möglichkeit ist folgender Kompromiß denkbar:
Wie im ersten Beispiel gibt es einen Master und $p-1$ Slaves, und der erste Schritt erfolgt wie oben beschrieben. Statt jedoch einfach die Ergebnisse der Berechnung ihrer Sturmschen Kette an den Master zu senden, stellen der erste und der letzte Prozeß fest, ob das erste bzw. letzte Teilintervall Eigenwerte enthält. Wenn dies so ist, so fahren sie direkt nach dem Senden des Ergebnisses fort, das nicht leere Intervall zu bearbeiten. Stellt ein Prozeß bei der Bisektion fest, daß beide Teilintervalle Eigenwerte enthalten, so sendet er die Informationen über eines davon an den Master. Hat er einen Eigenwert gefunden, so sendet er auch diesen an den Master mit dem Vermerk, daß er auf Arbeit wartet.
Hat ein Prozeß kein Intervall zu bearbeiten, so wartet er darauf, daß der Master ihm aus einer Liste der noch zu bearbeitenden Intervalle die Information eines Intervalls schickt. Ist die Liste leer, und sind alle Prozesse im Wartezustand, so sendet der Master eine Botschaft zur Beendigung der Berechnung an alle. Auch hier kennt am Schluß nur der Master-Prozeß alle Eigenwerte. Er sendet sie per Broadcast an alle anderen Prozesse.
Wenn m die Anzahl der zu berechnenden Eigenwerte ist, so erfordert diese Methode höchstens $3m-1+2p$ Kommunikationsaufrufe, von denen einige auch noch parallel ausgeführt werden können. Dies sind zwar wesentlich weniger als bei der ersten Methode, aber immer noch relativ viele für ein Problem mit so niedrigem arithmetischem Aufwand. Auch bei diesem Verfahren besteht außerdem die Gefahr, daß bei Konzentration der Eigenwerte in einem kleinen Intervall die Lastverteilung schlecht ist.

Ein Kompromiß mit minimalem Kommunikationsaufwand ist folgender:
Wie in [24] beschrieben, wird aus EISPACK die Prozedur TRIDIB verwendet, die eine

gewünschte Anzahl $\hat{m}$ von Eigenwerten ab einem bestimmten Index $\underline{m}$ (bei aufsteigender Anordnung der Eigenwerte) berechnet, also die Eigenwerte $\underline{m}$ bis $\underline{m}+\hat{m}-1$.
Zur Parallelisierung mit p Prozessen wird nun die Gesamtzahl m der gesuchten Eigenwerte durch p geteilt. Sind alle Eigenwerte zu bestimmen, also $m=n$, so berechnet Prozeß(j) dann mit TRIDIB die Eigenwerte mit Index $(j-1)\frac{m}{p}+1$ bis $j\frac{m}{p}$. Sind m Eigenwerte ab Index m_1 zu berechnen, so bestimmt Prozeß(j) die Eigenwerte mit Index $m_1+(j-1)\frac{m}{p}$ bis $m_1+j\frac{m}{p}$, ist dagegen ein Intervall angegeben, in dem alle Eigenwerte zu berechnen sind, so müssen zuerst der untere Index m_1 und die Anzahl der darin liegenden Eigenwerte m bestimmt werden, ehe wie eben beschrieben verfahren wird.
Auch hierbei wird nur Kommunikation erforderlich, um am Ende die auf p Prozesse verteilten Eigenwerte bei allen Prozessen zu sammeln.
Das Vorgehen ist wie folgt: Am Anfang bestimmen alle Prozesse gleichzeitig $m1$ und m aus dem gegebenen Intervall. Dies ist ohne Kommunikation möglich, da alle Prozesse bereits die Tridiagonalmatrix kennen. Die Berechnung der Indizes, ab denen der Prozeß jeweils die Eigenwerte bestimmen soll, läuft dann parallel in allen Prozessen ab.

Die Lastverteilung hängt hier ein wenig von der Größe der Teilintervalle ab, die der einzelne Prozeß zu bearbeiten hat, sie wird jedoch im allgemeinen wesentlich gleichmäßiger sein als bei der Zerlegung des Gesamtintervalls in gleiche Teilintervalle. Andererseits rechnen anfangs mehrere Prozesse so lange dasselbe, bis sie festgestellt haben, in welchem Intervall die von ihnen zu bestimmenden Eigenwerte jeweils liegen.

Dieses Verfahren funktioniert allerdings nicht, wenn exakt gleiche Eigenwerte mit Indizes $j\frac{m}{p}$ und $j\frac{m}{p}+1$ verhindern, daß an dieser Stelle das Intervall geteilt werden kann. In diesem Fall gibt TRIDIB einen Errorcode IERR$\neq 0$, und es werden keine Eigenwerte berechnet. Da aber die Diagonale und die Nebendiagonale der Matrix dabei nicht verändert werden, kann in diesem Fall nachträglich auf ein anderes Verfahren ausgewichen werden.

Zur Zeit ist das Verfahren, bei dem das Gesamtintervall gleichmäßig auf die Prozesse verteilt ist, und das Master-Slave Verfahren, bei dem jeder Prozeß ein Intervall ohne zusätzliche Kommunikation so lange teilt, bis die Genauigkeitsschranke unterschritten ist, auf dem SUPRENUM-Simulator auf einer SUN-Workstation implementiert.

7.2.3 Berechnung der Eigenvektoren mit inverser Iteration

Hat man die Eigenwerte einer Tridiagonalmatrix bestimmt, so lassen sich die zugehörigen Eigenvektoren durch die z.B. in [36] beschriebene inverse Iteration berechnen.
Hierbei wird für jeden vorher bestimmten Eigenwert λ, zu dem ein Eigenvektor gesucht ist, maximal fünf mal das Gleichungssystem $(T-\lambda I)\,y_i = x_i$ gelöst, wobei x_1 ein frei wählbarer Vektor mit $\|x_1\|_1 = 1$ ist und $x_i = \frac{y_i}{\|y_i\|_1}$ für $i > 1$.

Ist die Norm des Residuums $(T-\lambda I)\frac{y_i}{\|y_i\|_1} = \frac{x_i}{\|y_i\|_1}$ klein genug, so wird y_i als Eigenvektor akzeptiert und so normiert, daß $\|y\|_2 = 1$. Ist das Residuum nach fünf Iterationen noch

zu groß, so wird das Programm mit einem Fehlercode IERR $\neq 0$ beendet, wobei IERR angibt, für welchen Eigenwert kein Eigenvektor gefunden wurde.

Falls es exakt gleiche Eigenwerte gibt, werden diese für die Berechnung der Eigenvektoren ganz wenig modifiziert (s. [32]), damit alle Eigenvektoren zu dem mehrfachen Eigenwert bestimmt werden können. Diese Modifikation wird bei der Ausgabe der Eigenwerte rückgängig gemacht.
Eigenvektoren zu Eigenwerten, die dicht beieinander liegen, d.h. für die gilt $|\lambda_i - \lambda_j| < 10^{-3}\|T\|_1$, sind in der Regel nicht ausreichend orthogonal, d.h. ihr Skalarprodukt ist größer als die Genauigkeitsschranke. Sie werden daher untereinander nachorthogonalisiert.

7.2.4 Parallelisierung der inversen Iteration

Die Prozedur TINVIT aus EISPACK bestimmt zu einer beliebigen Menge aufeinander folgender Eigenwerte von T die zugehörigen Eigenvektoren. Für hinreichend gut getrennte Eigenwerte kann somit bei der parallelen Version jeder Prozeß die Eigenvektoren zu $\frac{m}{p}$ aufeinanderfolgenden Eigenwerten mit TINVIT berechnen. Dazu ist keine Kommunikation erforderlich, wenn jeder Prozeß die ganze Matrix und alle Eigenwerte kennt. Auch wenn nur die Eigenwerte, zu denen verschiedene Prozesse die Eigenvektoren berechnen sollen, hinreichend gut getrennt sind, kann jeder Prozeß lokal Eigenvektoren berechnen, und diese zusammen ergeben das gewünschte orthogonale Eigensystem.
Wenn jeder Prozeß alle Eigenwerte kennt, kann jeder Prozeß untersuchen, ob die Eigenwerte hinreichend getrennt sind oder nicht. Bei exakt gleichen Eigenwerten kann jeder Prozeß die notwendigen Korrekturen an allen exakt gleichen Eigenwerten vornehmen, die sicherstellen, daß verschiedene Eigenvektoren zu gleichen Eigenwerten berechnet werden. Hierdurch können dann auch verschiedene Prozesse verschiedene Eigenvektoren zu gleichen Eigenwerten berechnen.
Geht eine Gruppe gleicher oder nicht genügend getrennter Eigenwerte über Prozeßgrenzen hinaus, so müssen alle Prozesse, die zu Eigenwerten aus dieser Gruppe Eigenvektoren berechnet haben, diese Eigenvektoren gemeinsam nachorthogonalisieren. Hierfür ist Kommunikation nötig, und die Anzahl der Kommunikationsaufrufe hängt von der Anzahl der zu orthogonalisierenden Vektoren ab.

Zur Zeit ist noch ein vereinfachtes Verfahren implementiert, das alle Eigenvektoren nachorthogonalisiert, wenn zwei Eigenwerte, zu denen verschiedene Prozesse die Eigenvektoren berechnen, dicht beieinander liegen. Dies ergibt viel unnötigen Rechen- und Kommunikationsaufwand, aber es muß nur eine Orthogonalisierungsprozedur geschrieben werden, die davon ausgeht, daß m Vektoren, die orthogonalisiert werden sollen, gleichmäßig auf p Prozesse verteilt vorliegen.
Dabei wird blockweise zyklisch vorgegangen, so daß die Eigenvektoren nicht in ihrer ursprünglichen Reihenfolge orthogonalisiert werden. Die Blockgröße $q2$ ist frei wählbar. Zuerst orthogonalisiert der erste Prozeß einen Block von $q2$ Eigenvektoren mit einem modifizierten Gram-Schmidt-Verfahren. Diese sendet er per Broadcast den anderen Prozessen. Der nächste Prozeß, der etwas senden muß, ist dann Prozeß(2). Dieser orthogonalisiert seinen ersten Block von $q2$ Eigenvektoren zuerst bezüglich der empfangenen Eigenvektoren und dann untereinander. Anschließend sendet er diesen Block allen

anderen Prozessen und orthogonalisiert dann erst seine übrigen Eigenvektoren zuerst bezüglich der empfangenen Eigenvektoren, dann bezüglich des eigenen ersten Blocks. So wird reihum verfahren, bis alle m Eigenvektoren orthogonalisiert sind.
Jeder der p Prozesse hat dabei $(m/p)/q2$ Blöcke, es sind also $p\cdot(m/p)/q2$ Broadcasts zu je $\lceil \log_2 p \rceil$ Kommunikationsaufrufen durchzuführen, was einen zusätzlichen Aufwand von $p\cdot\lceil \log_2 p \rceil\ (m/p)/q2$ Kommunikationsaufrufen zum arithmetischen Aufwand hinzufügt. Durch die "Compute-and-Send-Ahead"-Technik wird ein Teil dieses Zusatzaufwands parallel zur Arithmetik durchgeführt. Andererseits kann nicht alle Arithmetik parallel ausgeführt werden, es entstehen Wartezeiten.

7.2.5 Berechnung der Eigenwerte und Eigenvektoren mit Cuppens Methode

Das Verfahren ist in [9] und [16] ausführlich beschrieben. Die "Divide-and-Conquer" Methode beruht darauf, daß es möglich ist, die Tridiagonalmatrix zu halbieren, alle Eigenwerte und Eigenvektoren der beiden Hälften zu berechnen und dann alle Eigenwerte und Eigenvektoren der gesamten Matrix durch ein Rang-1-Modifikations-Verfahren zu bestimmen. Für die Modifikation werden auch die Eigenvektoren der Teilmatrizen gebraucht, und es werden immer alle Eigenwerte und Eigenvektoren der Matrix bestimmt.

7.2.6 Parallelisierung der Methode von Cuppen

Zur Parallelisierung wird die Matrix zunächst so oft halbiert, bis $p = 2^{(\log_2 p)}$ Teilmatrizen gleicher Größe entstanden sind. Dabei sollte sinnvollerweise p als Zweierpotenz gewählt werden. Im folgenden wird dies für das Cuppen-Verfahren vorausgesetzt, so daß $\lceil \log_2 p \rceil = \log_2 p$ gilt. Jeder der p Prozesse bekommt eine Teilmatrix, deren Eigenwerte und Eigenvektoren er mit einem QL-Verfahren aus einer Programmbibliothek berechnet. Nun könnte der Modifikationsteil in einem „Einsammelverfahren" ähnlich dem im Kapitel „Kommunikationsstrategien" beschriebenen Sammeln eines verteilten Vektors durchgeführt werden, bei dem jeweils ein Prozeß oder auch beide Prozesse gleichzeitig die Teilergebnisse zweier Prozesse zusammenfassen und den Modifikationsprozeß durchführen. Zum Schluß würde dann ein Prozeß oder alle Prozesse zugleich die beiden Hälften der ursprünglichen Matrix zusammensetzen. Da jedoch, wie Messungen ergaben, allein dieser letzte Modifikationsprozeß beim sequentiellen Verfahren im Durchschnitt etwa die Hälfte der Rechenzeit benötigt, ist Parallelisierung nur dann sinnvoll, wenn auch der Modifikationsprozeß parallelisiert werden kann. Wie dies möglich ist, ist in [24] für einen Hypercube beschrieben, und so kann es im wesentlichen für SUPRENUM übernommen werden. Bei dieser Parallelisierung führt jeder Prozeß die Modifikation von nur $\frac{n}{p}$ Eigenwerten und Eigenvektoren durch.

Kommunikation findet im i-ten Modifikationsschritt $(i = 1,..., \log_2 p)$ zwischen den 2^i Prozessen statt, die ihre Teilmatrizen zusammensetzen. Es gibt dabei $\frac{p}{2^i}$ Teilmengen zu je 2^i Prozessen, die je $2(2^i - 1)$ Kommunikationen durchführen. Die Initialisierungen können jeweils in allen Prozessen parallel ausgeführt werden, so daß im i-ten Modifikationsschritt insgesamt $2(2^i - 1)$ parallele Kommunikationsaufrufe stattfinden. Das gesamte Cuppen-Verfahren benötigt somit $4(p-1) - 2\log_2 p$ parallele Kommunikationsaufrufe.

7.2.7 Vergleich der verschiedenen Verfahren

Sowohl der arithmetische als auch der Kommunikationsaufwand der Verfahren mit Bisektion und inverser Iteration läßt sich nicht vorhersagen, da er stark von der Verteilung der Eigenwerte im Spektrum abhängt.
Auch der arithmetische Aufwand der Methode von Cuppen ist von der Struktur der Matrix abhängig, da die Modifikation entfällt, wenn die Matrix an der Teilungsstelle zerfällt.

EISPACK empfiehlt für die Berechnung von mehr als 25% der Eigenwerte die Verwendung eines QL-Verfahrens, da dieses dann schneller ist als ein Bisektionsverfahren. Dies gilt jedoch, wenn keine Eigenvektoren gewünscht sind, nur für die QL-Verfahren, bei denen die Eigenvektoren nicht mit berechnet werden. Für SUPRENUM existiert nur ein paralleles QL-Verfahren, bei dem sowohl alle Eigenwerte als auch alle Eigenvektoren der Tridiagonalmatrix bestimmt werden, nämlich das nach der Methode von Cuppen. Dieses sollte entsprechend verwendet werden, wenn mehr als 25% der Eigenwerte und Eigenvektoren einer symmetrischen Tridiagonalmatrix berechnet werden sollen, da der Parallelisierungsgrad des Cuppen-Verfahrens ähnlich gut sein wird wie der des Verfahrens mit Bisektion und inverser Iteration. Auch der Kommunikationsaufwand ist nicht wesentlich größer als beim Bisektionsverfahren mit inverser Iteration. Falls wegen dichter Eigenwerte nachorthogonalisiert werden muß, ist der Kommunikationsaufwand bei der inversen Iteration sogar höher als beim Cuppen-Verfahren.
Sollen jedoch nur Eigenwerte oder nur wenige Eigenvektoren berechnet werden, so ist ein Bisektionsverfahren mit anschließender inverser Iteration zu empfehlen.

Ein Vergleich der verschiedenen Parallelisierungen des Bisektionsverfahrens zeigt, daß die Verfahren, bei denen der Kommunikationsaufwand minimiert wird, im allgemeinen den Master-Slave Verfahren vorzuziehen sind.

7.3 Bestimmung der Eigenvektoren einer vollbesetzten Matrix

Die Eigenvektoren der vollbesetzten Matrix werden wie bereits erwähnt, durch Rücktransformation der Eigenvektoren der Tridiagonalmatrix gewonnen. Da beim Schwarz-Verfahren zur Reduktion der Bandmatrix auf Tridiagonalgestalt jedoch die Information über die angewandten Givens-Rotationen nicht gespeichert wird, ist diese Rücktransformation nicht möglich, wenn Block-Householdertransformation und Schwarz-Verfahren zur Reduktion auf Tridiagonalgestalt verwendet wurden. In diesem Fall müssen die Eigenvektoren der Bandmatrix direkt mit inverser Iteration aus den Eigenwerten, die mit Bisektion bestimmt worden sind, berechnet werden. Dann können die Eigenvektoren der Bandmatrix mit den W und Y aus der Block-Householdertransformation zurücktransformiert werden.

Die Methode von Cuppen, die ja nur Eigenwerte und Eigenvektoren einer Tridiagonalmatrix liefert, kann nur angewandt werden, wenn die volle Information über die Transformation auf Tridiagonalgestalt vorhanden ist, also wenn mit dem gewöhnlichen Householderverfahren reduziert wurde.

7.3.1 Berechnung der Eigenvektoren einer Bandmatrix mit inverser Iteration

Das Verfahren läuft analog zur Berechnung der Eigenwerte einer Tridiagonalmatrix mit inverser Iteration ab. Die auftretenden Probleme sind im wesentlichen dieselben. Bei der Bandmatrix kann zusätzlich die Information über das Zerfallen der Tridiagonalmatrix nicht ausgenützt werden.

7.3.2 Rücktransformation der Eigenvektoren einer Band- oder Tridiagonalmatrix auf die der vollbesetzten Matrix

Im Prinzip müssen nur die m Eigenvektoren der Tridiagonal- oder der Bandmatrix, die als die Spalten einer verteilten $n \times m$-Matrix Z gespeichert sind, mit der jeweiligen Transformationsmatrix Q multipliziert werden. Da diese nirgendwo explizit berechnet und gespeichert ist, geschieht die Rücktransformation durch sukzessive Modifikation mit Hilfe der in verschiedenen Prozessen gespeicherten w bzw. den W und Y.

Im Fall der Block-Householdertransformation wird wie folgt verfahren:
Beginnend mit den zuletzt berechneten W und Y bis zu den ersten sendet jeweils der Prozeß, der die gerade aktuellen W und Y in seinem Speicher hat, diese per Broadcast an alle übrigen Prozesse. Jeder Prozeß berechnet dann für seinen Teil Z_j der Matrix Z der Eigenvektoren das Produkt $(I - WY^T)Z_j$.
Verwendet man hierzu die im Kapitel „Kommunikationsstrategien" als zweite beschriebene Version des Softwarebroadcast, so läuft die Rücktransformation in $\frac{n}{q}$ Schritten ab, die jeweils aus einem Softwarebroadcast und Matrixmultiplikationen bestehen. Der arithmetische Aufwand wächst mit jedem Schritt, da immer mehr Zeilen von W und Y von Null verschieden sind. Im i-ten Schritt ($i = 1, \ldots, \frac{n}{q}$) beträgt er $i \cdot q \frac{m}{p} (4q+1)$ arithmetische Operationen. Der Gesamtzeitaufwand beträgt dann

$$T_{ges}^{II} \simeq \frac{n}{q} \lceil \log_2 p \rceil \, T_{Kom} + \frac{n^2}{2q} \frac{m}{p} (4q+1) T_{Ar} \simeq \frac{n}{q} \lceil \log_2 p \rceil \, T_{Kom} + \frac{n^2}{2} \frac{m}{p} (4 + \frac{1}{q}) T_{Ar} \, .$$

Ein "Compute-and-Send-Ahead", bei dem jeweils der Prozeß, der das nächste W und Y verschicken soll, den Broadcastaufruf vor der Matrixmultiplikation ausführt, bringt keine wesentliche Änderung der Situation wegen der Einführung einer Synchronisation der Prozesse durch die verwendete Art des Broadcastings. (Die "Compute-and-Send-Ahead"-Technik ist ohnehin nur bei zyklischer Verteilung der W und Y, also bei zyklischer Aufteilung der vollbesetzten Matrix durchführbar.)

Anders ist die Situation, wenn die an erster Stelle im Kapitel „Kommunikationsstrategien" beschriebene Broadcasttechnik angewandt wird. Hier muß in jedem Schritt wirklich nur der Prozeß, der die benötigten W und Y besitzt, senden. Tut er dies, ehe er mit der Matrixmultiplikation beginnt, und ist er der erste, der die vorigen W und Y empfangen hat, so erreichen die nächsten W und Y die Mailboxes der übrigen Prozesse nur um die Zeit für die Initialisierung eines Empfangens und die Übertragungszeit, also um $t_\delta + 2nqt_\beta$, später als die vorigen Matrizen.
Den Prozeß, der die ersten Matrizen verschickt hat, erreichen die zweiten um $t_\delta + (p-1)t_\gamma + 2n \cdot qt_\beta$ nach dem Zeitpunkt, an dem er das erste Senden vollzogen hat. Inzwischen hat er noch $(p-2)t_\gamma$ mit Kommunikation zugebracht, dann hat er mit der Rechnung begonnen. Die zweiten Matrizen sind also $t_\gamma + t_\delta + nqt_\beta$ nach Beginn des Rechnens bei dem Prozeß, der als erster gesendet hatte. Dies ist gerade die Zeit, die im Kapitel „Kommunikationsstrategien"mit T_{Kom} bezeichnet wurde. Die gleiche Situation ergibt sich jeweils für den Prozeß, der im i-ten Schritt die Matrizen veschickt hat. Auch er bekommt die Matrizen, die im $i+1$-ten Schritt verschickt werden, um die Zeit T_{Kom} später als er mit der Aktualisierung mit Hilfe der i-ten W und Y begonnen hat.

Ist also $T_{Kom} < i \cdot q \frac{m}{p} (4q+1) T_{Ar}$ für alle $i = 1, \ldots, \frac{n}{q}$, so ergeben sich keine Wartezeiten für die Prozesse. Ist dagegen $\hat{i} \cdot q \frac{m}{p} (4q+1) T_{Ar} \leq T_{Kom} \leq (\hat{i}+1) \cdot q \frac{m}{p} (4q+1) T_{Ar}$, für ein $\hat{i} < \frac{n}{q}$ oder gar $T_{Kom} \geq n \frac{m}{p} (4q+1) T_{Ar}$, so ergibt sich jeweils für den Prozeß, der die i-ten $(i \leq \hat{i})$ Matrizen verschickt hat, nach der Aktualisierung mit den i-ten Matrizen eine Wartezeit von $T_{Kom} - i \cdot q \frac{m}{p} (4q+1) T_{Ar}$ auf die $i+1$-ten Matrizen. Der Gesamtzeitaufwand ist gleich dem des Prozesses, der als letzter die ersten Matrizen W und Y bekommt.
Dieser Aufwand setzt sich wie folgt zusammen: Am Anfang entsteht eine Wartezeit, die der Zeit von $p-1$ Sendeinitialisierungen entspricht, bevor bei dem letzten Prozeß die erste Matrix eingetroffen ist. Danach führt dieser Prozeß jeweils $p-1$ Schritte durch, die aus einem Empfangen und der Matrixaktualisierung bestehen. Der p-te Schritt besteht dann aus $p-1$ Sendeinitialisierungen und einer Matrixaktualisierung. Diese p Schritte wiederholen sich $\frac{n}{q} \cdot p$ mal. Sofern $T_{Kom} > i \cdot q \frac{m}{p} (4q+1) T_{Ar}$ für die ersten $i \leq \hat{i}$, verschieben sich die nächsten p Schritte um die Zeit, die der Prozeß, der als erster gesendet hat, auf die zweiten Matrizen wartet. An einigen Stellen ist daher die Zeit für die Matrixaktualisierung durch die Zeit T_{Kom} zu ersetzen, wenn letztere länger ist. Sicher ist die Gesamtzeit nicht unterschätzt, wenn in den ersten $\hat{i}$ Schritten, so lange also, wie

$T_{Kom} > i \cdot q \frac{m}{p}(4q+1)T_{Ar}$, in jedem Schritt für alle Prozesse statt der Zeit für die Arithmetik jeweils T_{Kom} gesetzt wird. Falls $T_{Kom} \geq n \frac{m}{p}(4q+1)T_{Ar}$, so sei $\hat{i} = \frac{n}{p}$.
Insgesamt beträgt die Zeit damit höchstens

$$T_{ges}^{I} \simeq (\frac{p-1}{2} + \frac{n}{q}\frac{(p-1)}{p} + \hat{i})T_{Kom} + \frac{n^2 - (\hat{i} \cdot q)^2}{2}\frac{m}{p}(4 + \frac{1}{q})T_{Ar} ,$$

wobei $\hat{i}$ durch obige Ungleichung gegeben ist. Da zudem $\hat{i} \leq \frac{n}{q}$, gilt für die Gesamtzeit sicher, daß

$$T_{ges}^{I} < (\frac{p-1}{2} + 2\frac{n}{q})T_{Kom} + \frac{n^2}{2}\frac{m}{p}(4 + \frac{1}{q})T_{Ar} .$$

Das heißt, daß für $p > 4$, also $\lceil \log_2 p \rceil > 2$, immer eine Einsparung von wenigstens $(\frac{n}{q}(\lceil \log_2 p \rceil - 2) - \frac{p-1}{2})T_{Kom}$ gegenüber der Version mit Softwarebroadcast in Stufen erreicht werden kann. Ist $\hat{i} \ll \frac{n}{q}$, so beträgt die Einsparung sogar fast $(\frac{n}{q}(\lceil \log_2 p \rceil - 1) - \frac{p-1}{2})T_{Kom}$.

War die vollbesetzte Matrix durch gewöhnliche Householdertransformation auf Tridiagonalgestalt reduziert worden, so gibt es noch mehr Möglichkeiten für die Rücktransformation der Eigenvektoren.
Zum einen kommen die für den Fall der Block-Householdertransformation beschriebenen Versionen mit den beiden Broadcasts in Frage. Falls die Matrix nicht zyklisch verteilt war, die w also nicht zyklisch verteilt sind, ist dabei das Broadcasting in Stufen vorzuziehen, da ein "Compute-and-Send-Ahead" nicht möglich ist. Hier kann man allerdings Kommunikation einsparen, indem der Prozeß, der die nächsten w besitzt, nicht nur eines, sondern gleich einen Block mit $\tilde{q}$ Vektoren w sendet. ($\tilde{q}$ ist dabei beliebig, hier ist kein q von einer Block-Householdertransformation definiert.)
Die benötigte Gesamtzeit beträgt dann

$$T_{ges}^{II'} \simeq \frac{n}{q}\lceil \log_2 p \rceil T_{Kom} + 2n^2 \frac{m}{p} T_{Ar} .$$

Verwendet man das Broadcasting in Stufen bei zyklischer Verteilung der w, so kann immer nur ein w geschickt werden, und es ergibt sich eine Gesamtzeit von

$$T_{ges}^{II} \simeq n\lceil \log_2 p \rceil T_{Kom} + 2n^2 \frac{m}{p} T_{Ar} .$$

Durch die "Compute-and-Send-Ahead"-Technik und Broadcast der ersten Version kann man bei zyklischer Verteilung der w dies auf

$$T_{ges}^{I} \simeq (n\frac{p-1}{p} + \frac{p-1}{2} + \hat{i})T_{Kom} + 2(n^2 - \hat{i}^2)\frac{m}{p} T_{Ar}$$

reduzieren, wobei $0 \leq \hat{i} \leq n$ wieder dadurch gegeben ist, daß

$$4\hat{i}\frac{m}{p} T_{Ar} \leq T_{Kom} \leq 4(\hat{i}+1)\cdot\frac{m}{p} T_{Ar} .$$

Bei zyklischer Verteilung der w gibt es jedoch noch eine weitere Möglichkeit, Kommunikation einzusparen. Statt die Vektoren w reihum durch Broadcast allen Prozessen be-

kannt zu machen, können jeweils p Vektoren w, also von jedem Prozeß einer, durch die im Kapitel „Kommunikationsstrategien" beschriebene Sammelroutine bei allen Prozessen als die Spalten einer Hilfsmatrix gesammelt werden. Anschließend aktualisieren alle Prozesse ihre Matrix Z der Eigenvektoren mit allen p Householdervektoren. Hier, wie auch bei der ersten Methode, die bei kontinuierlicher Verteilung der Vektoren w beschrieben ist, kann zusätzlich, zur Verminderung der Anzahl der Zugriffe auf die Matrix Z, das Produkt der Householdermatrizen aus den p bzw. q Householdervektoren in WY Darstellung gebildet werden, um dann damit einmal eine Multiplikation mit Z durchzuführen. Dafür sind allerdings insgesamt mehr arithmetische Operationen nötig.
Der Gesamtaufwand mit Sammeln der Vektoren w, aber ohne Verwendung der WY-Methode beträgt dann

$$T_{ges}^{III} \simeq \frac{n}{p} \left\lceil \log_2 p \right\rceil T_{Kom} + 2n^2 \frac{m}{p} T_{Ar} \ .$$

Diese als letzte beschriebene Version der Rücktransformation von Eigenvektoren, die dann angewandt werden kann, wenn die Reduktion der vollbesetzten Matrix durch gewöhnliche Householdertransformation bei zyklischer Aufteilung der Matrix erfolgte, ist die einzige, bei der auch der Kommunikationsaufwand mit p, der Anzahl der Prozesse, abnimmt. Da andererseits bei der Reduktion der Matrix auf Tridiagonalgestalt das gewöhnliche Householderverfahren mit zyklischer Aufteilung der Matrix bei großen Matrizen auch für große p voraussichtlich am schnellsten ist, ist auch insgesamt in den meisten Fällen ein p zu finden, so daß das gewöhnliche Householderverfahren mit zyklischer Aufteilung der Matrix für die Berechnung sowohl von Eigenwerten als auch von Eigenvektoren allen anderen Verfahren vorzuziehen ist.

Schlußbemerkung

Da die SUPRENUM Hardware und Betriebssoftware noch ständig verbessert werden, ändern sich auch die Performance-Parameter für den SUPRENUM Rechner noch. Daher ist es wichtig, verschiedene Verfahren zu entwickeln und zu implementieren. Damit ist es möglich, durch Testläufe auf der jeweils aktuellen Hardware festzustellen, welches Verfahren gewählt werden soll.
Nach Änderungen an der Hardware oder der Systemsoftware sollte dann jeweils wieder neu geprüft werden, ob nicht ein anderes Verfahren inzwischen vorzuziehen ist.

Literatur

[1] W. Barth, R.S. Martin, and J.H. Wilkinson, Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by the method of Bisection, in J. H. Wilkinson and C. Reinsch, Handbook for Automatic Computation, Volume II, Linear Algebra, (Springer Verlag 1971) 249-256

[2] C. Bischof, QR-Factorization Algorithms for Coarse-Grained Distributed Systems, Ph.D. thesis, Cornell University, August 1988

[3] C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, and D. Sorensen, LAPACK Working Note #5: Provisional Contents, ANL-88-38, Argonne National Laboratory, September 1988

[4] C. Bischof and C. Van Loan, The WY Representation for Products of Householder Matrices, SIAM Journal on Scientific and Statistical Computing 8, 1 (1987) s2-s13

[5] O. Brewer, J. Dongarra, and D. Sorensen, LAPACK Working Note #6: Tools to Aid in the Analysis of Memory Access Patterns for Fortran Programs, Argonne National Laboratory, Technical Memorandum No. 120, June 1988

[6] O. Büchner, Parallelisierung eines Full-Multigrid-Eigenwertalgorithmus zur Berechnung von Eigenwerten der Maxwellgleichung, in Sprachen, Algorithmen und Architekturen für Parallelrechner, Gemeinsamer Workshop der Fachgruppe 2.1.4 Alternative Konzepte für Sprachen und Rechner und Fachgruppe 3.1.2 PARS Parallel-Algorithmen und -Rechnerstrukturen, Bad Honnef (1988) (ISSN 0177-0454), pp.17-24

[7] J.R. Bunch, C.P. Nielsen, and D.C. Sorensen, Rank-One Modification of the Symmetric Eigenproblem, Numer. Math. 31 (1978), 31-48

[8] E. Chu and A. George, Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor, Parallel Computing 5 (1987), 65-74

[9] J.J.M. Cuppen, A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem, Numer. Math. 36 (1981), 177-195

[10] J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen, Prospectus for the Development of a Linear Algebra Library for High-Performance Computers, Technical Report, ANL-MCS-P88-1, Argonne National Laboratory, August 1988

[11] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart LINPACK Users' Guide, SIAM Philadelphia (1979)

[12] J.J. Dongarra, J.J. Du Croz, I. Duff, and S. Hammarling, A Set of Level 3 Basic Linear Algebra Subprograms, with – Model Implementation and Test Programs, Argonne National Laboratory, Mathematics and Computer Science Division, Preprint No. 1 and 2, August 1988

[13] J.J. Dongarra, J.J. Du Croz, S.J. Hammarling, and R.J. Hanson, An extended Set of Fortran Basic Linear Algebra Subprograms, with – Model Implementation and Test Programs, ACM Trans. Math. Software 14 (1988), 1-32
[14] J.J. Dongarra and E. Grosse, Distribution of Mathematical Software by Electronic Mail, Communications of the ACM 30 (1987), 403-407
[15] J.J. Dongarra, S.J. Hammarling, and D.C. Sorensen, Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations, LAPACK Working Note #2, Argonne National Laboratory, Mathematics and Computer Science Division, September 1987
[16] J.J. Dongarra and D.C. Sorensen, A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem, SIAM J. Sci. Stat. Comput. 8 (1987) s139-s154
[17] A. Emmen et al., Suprenum special, Supercomputer 30, SARA Amsterdam (1989)
[18] G.A. Geist and M.T. Heath, Matrix Factorization on a Hypercube Multiprocessor, in M.T. Heath (ed.), Hypercube Multiprocessors 1986, SIAM Philadelphia (1986) 161-180
[19] G.H. Golub and C.F. VanLoan, Matrix Computations, (John Hopkins University Press, Baltimore, 1983)
[20] R. Grimes, HSSXEV- an Out-of-core Symmetric Eigensolver for Large Dense Problems, BOEING Computer Services (1987), FORTRAN package, received via netlib [14]
[21] I. Gutheil, Implementation einer parallelen Partitionsmethode zur Lösung linearer Gleichungssysteme mit Bandmatrix auf einer simulierten MIMD-Architektur mit lokalem Speicher, PARS-Mitteilungen 4 (1987), pp. 114-121
[22] I. Gutheil, SUPRENUM Software for the Symmetric Eigenvalue Problem, Parallel Computing 7 (1988), 419-424
[23] M.T. Heath and C.H. Romine, Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors, Oak Ridge National Laboratory, ORNL/TM-10384, March 1987
[24] I.C.F. Ipsen and E.R. Jessup, Solving the Symmetric Tridiagonal Eigenvalue Problem on the Hypercube, Research Report YALEU/DCS/RR-548, July 1987
[25] G. Li and T.F. Coleman, A Parallel Triangular Solver for a Hypercube Multiprocessor, in M.T. Heath (de.), Hypercube Multiprocessors 1987, SIAM Philadelphia (1987) 539-551
[26] C. Moler, Matrix Computation on Distributed Memory Multiprocessors, in M.T. Heath (ed.), Hypercube Multiprocessors 1986, SIAM Philadelphia (1986) 181-195
[27] B.N. Parlett, The Symmetric Eigenvalue Problem (Prentice-Hall 1980).
[28] W. Rönsch and H. Strauß, A Linear Algebra Package for a Local Memory Multiprocessor: Problems, Proposals and Solutions, Parallel Computing 7 (1988), 413-418
[29] W. Rönsch and H. Strauß, Design Aspects of a Linear Algebra Package for the SUPRENUM Multiprocessor System, in Proceedings of the second International Conference on Vector and Parallel Computing, June 6-11 1988 in Tromsø, Norway, (Ellis Horwood Ltd. Publ. 1989?)
[30] W. Rönsch and H. Strauß, The Level 3 BLAS Forms of Parallel Factorization Methods, in Proceedings of Parallel Computing '89, Aug. 29 - Sept. 1 1989 in Leiden, The Netherlands, (North-Holland 1989)
[31] H.R. Schwarz, Tridiagonalization of a Symmetric Band Matrix, Numerische Mathematik 12 (1968) 231-241.

[32] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler, Matrix Eigensystem Routines-EISPACK Guide, Lecture Notes in Computer Science 6, 2nd edition (Springer Verlag 1976)

[33] K. Solchenbach and U. Trottenberg, SUPRENUM-System Essentials and Grid Applications, Bonn 1988

[34] H. Strauß and W. Rönsch, Adaptive block techniques for matrix factorization on distributed-memory computers, in Hypercube and Distributed Computers, F. André and J. P. Verjus (eds.), Proceedings of the First European Workshop on Hypercube and Distributed Computers, (North Holland 1989), 355-356

[35] U. Trottenberg, Proceedings of the 2nd International SUPRENUM Colloquium, 30 September - 2 October 1987, Bonn, FRG, Parallel Computing 7 (1988)

[36] J.H. Wilkinson and C. Reinsch, Handbook for Automatic Computation, Volume II, Linear Algebra, (Springer Verlag 1971)