Parallele Numerische Verfahren

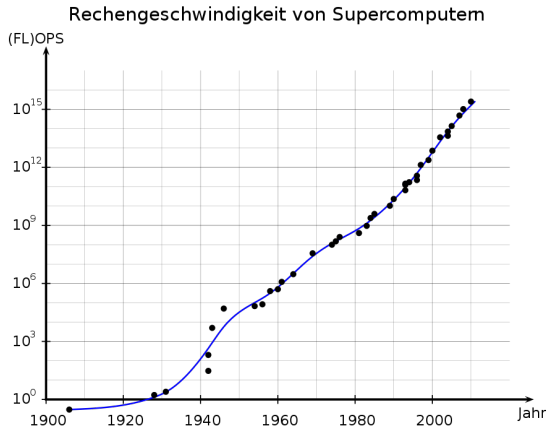
Bernhard Schmitt

Sommer-Semester 2011

Inhaltsverzeichnis

1 Einleitung

Nach dem Mooreschen "Gesetz" verdoppelt sich die Rechenleistung gängiger Computer jeweils in einem Zeitraum von ca. 18 Monaten. Wegen physikalischer Grenzen konnte diese Regel von den Produzenten in den letzten Jahren aber nur noch durch die Verwendung mehrerer Prozessoren eingehalten werden, die *parallel* arbeiten. Bei heutigen Arbeitsplatzrechnern ist dabei die Zahl der Parallelprozessoren (Kerne) noch recht überschaubar (dual core, quad-core) bei Supercomputern wird dagegen schon länger eine sehr große Anzahl von Prozessoren eingesetzt. Der **Mar**burger **R**echen-**C**luster MaRC hat z.Z. 568 Rechnerkerne und eine Spitzenleistung von 2.5 TFLOPS, die aktuellen Spitzenreiter mit einer Rechenleistung im PetaFLOPS-Bereich dagegen ca. 200 Tausend. Die Graphik (Quelle: Wikipedia) zeigt den historischen Zuwachs der Leistungen. Antrieb für den Leistungszuwachs ist der Wunsch, immer feiner auflösende Modelle realer Probleme (Klimamodelle, Wettervorhersage, etc) zu rechnen, d.h. Modelle mit immer mehr Variablen, um die Qualität der Ergebnisse zu verbessern. Denn bei vielen Problemklassen wird die Genauigkeit der Simulation vor Allem durch die Zahl der Freiheitsgrade im Modell bestimmt.

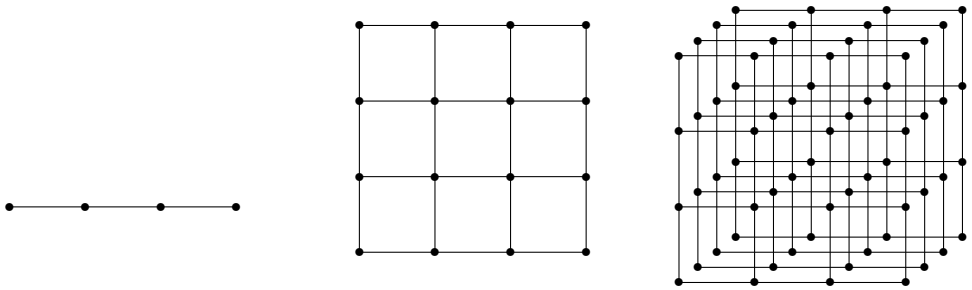


Bei biologischen oder ökonomischen Modellen kann die Zahl der Freiheitsgrade durch die Zahl der Spezies bzw. Marktteilnehmer bestimmt sein, von der die Genauigkeit irgendwie abhängt. Bei einer wichtigen Problemklasse ist dagegn der Zusammenhang zwischen Genauigkeit und Modellgröße recht genau bekannt. Partielle Differentialgleichungen, die etwa bei Klimamodellen, Maschinen-Simulation (Autos), Wetter-Modellen etc. auftreten, beschreiben Lösungsfunktionen, die von bis zu drei Ortsvariablen und der Zeit abhängen können, also auf einem Gebiet $\Omega \subseteq \mathbb{R}^d$ definiert sind. Zur Approximation kann man das Gebiet durch ein Gitter unterteilen und man berechnet Lösungsnäherungen auf den Gitterpunkten. Nimmt man in jeder Ortsrichtung ca. $m \in \mathbb{N}$ Gitterpunkte, dann ist die Genauigkeit typischerweise von der Form $\sim m^{-2}$. Bei einer feineren Auflösung (gößeres m) wird der Fehler tatsächlich schnell kleiner. Eine spezielles Beispiel hierfür wird in §5.1 genauer beschrieben. Bei Problemen mit Ortsdimension $d = 1$ liefert dieses Verfahren recht einfach gute Näherungen. Bei höheren Dimensionen wächst die Zahl der Gitterpunkte aber wie $n \sim m^d$. Die folgende Tabelle zeigt daher direkt den Zusammenhang zwischen Modellgröße n und dem Fehler $\sim m^{-2} \sim n^{-2/d}$ bzw. die für einen Fehler $10^{-\ell}$ erforderliche Problemgröße n:

Gebiet im	R^1	$\mathbb{R}^2$	$\mathbb{R}^3$
Problemgröße $n \sim$	m	m^2	m^3
Fehler $\sim$	n^{-2}	n^{-1}	$n^{-2/3}$
$Fe \leq 10^{-\ell} \Rightarrow n \sim$	$10^{\ell/2}$	10^{ℓ}	$10^{1.5\ell}$

Die folgende Graphik veranschaulicht den Zusammenhang $n = m^d$ für $m = 4$.

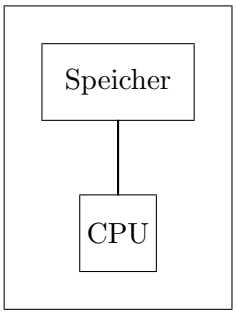
4 Gitterpunkte 1D 4^2 Gitterpunkte 2D 4^3 Gitterpunkte 3D



Bei Parallel-Rechnern ist Vorsicht bei den Leistungsangaben angebracht. Die Spitzenleistung (*peak performance*) ist eine "Garantie" des Herstellers, dass man diese Leistung nie erreicht, denn sie addiert einfach aller Einzelleistungen. Ein Grundproblem bei gleichzeitiger Bearbeitung von Aufgaben ist aber immer der wachsende Abstimmungsaufwand, der der Beschleunigung entgegensteht. Daher vergleicht man Rechner anhand der tatsächlichen Maximalleistung, welche durch *Benchmarks* an wichtigen Einsatzbereichen (z.B. der Linearen Algebra) ermittelt wird.

1.1 Rechner-Aufbau

Von-Neumann-Rechner Ein klassischer Rechner besteht aus einer zentralen Recheneinheit (CPU = central processing unit) und einem Speicher, wenn man Ein- und Ausgabe vernachlässigt. In dem Speicher liegen die zu bearbeitenden Daten, aber auch das verarbeitende Programm selbst. In der CPU gibt es einen Taktgeber, der die einzelnen Verarbeitungsschritte strukturiert. Die CPU kann jeweils nur **eine** Programmanweisung ausführen, welche durchaus mehrere Taktzyklen in Anspruch nehmen kann, aber mindestens einen. Dabei operiert die CPU



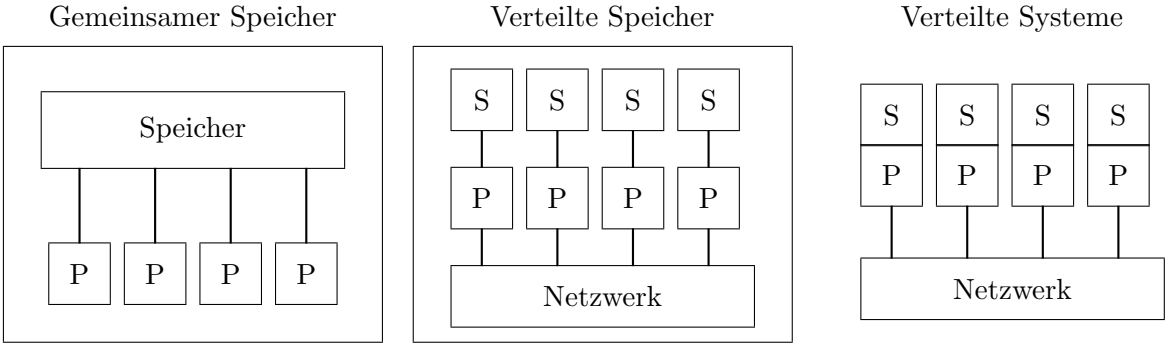
mit Daten aus wenigen lokalen Speicherplätzen (Registern) und dem (Haupt-) Speicher. Der Zugriff auf den Speicher ist auch getaktet, allerdings mit längeren Zugriffszeiten. Da das Signal die CPU innerhalb eines Takts in ihrer ganzen Ausdehnung erreichen muß, stellt die Lichtgeschwindigkeit bei gegebener Chipgröße eine physikalische Schranke für die Taktung dar. Bei gängigen PC-Prozessoren liegt diese jetzt bei ca. 2 GHz, also einer Taktzeit von einer halben Nanosekunde, was die Leistung eines solchen Rechners auf wenige Giga-FLOPS (*FLoating point Operation Per Second*) begrenzt.

Wenn man die beiden Resourcen Speicher und CPU als gleichwertig ansieht, stellt die CPU hier eine Engstelle (Flaschenhals) dar, weil sie dauernd aktiv ist, die einzelnen Speicherbereiche aber eher selten benutzt werden. Die Lasten sind also ungleich verteilt. Weitere Leistungssteigerungen erforderten daher Korrekturen bei Rechnerarchitektur und Algorithmenaufbau.

Parallele Architekturen

Man kann Prozesse dadurch beschleunigen, dass man kritische Aufgaben auf Mehrere verteilt. Da im Von-Neumann-Rechner die CPU den Flaschenhals bildet, erwartet man daher durch Ver-

wendung mehrerer Prozessoren ein Beschleunigung. Dabei stellt sich aber gleich die Frage der

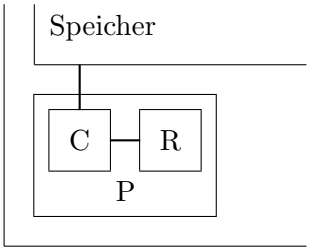


Speicherzuordnung. Moderne PCs besitzen einen *gemeinsamen Speicher*, auf den die Prozessoren, die als Kerne i.d.R. auf einem Chip untergebracht sind (z.B. "Quadcore"), in Konkurrenz zugreifen. Hier sind alle Daten für alle Prozessoren verfügbar, Koordinationsprobleme beim Datenzugriff löst eine geeignete, schnelle und teure Hardware. Bei der Entwicklung von Algorithmen kann man sich auf die Parallelisierung der Operationen konzentrieren. Diesen Maschinentyp bezeichnet man auch als *Multiprozessor-Systeme*.

Bei einer großen Anzahl von Prozessoren kann man jedem Prozessor einen eigenen Speicher zuordnen (verteilte Speicher), wie beim Marburger Cluster (MaRC besteht genau genommen aus 140 Quadcores). Diese Architektur stellt aber hohe Anforderungen an das verbindende Netzwerk zwischen den Prozessoren. Bei Parallelalgorithmen müssen jetzt nicht nur die Arbeiten, sondern auch die Daten auf die verschiedenen Prozessoren verteilt werden und der im Vergleich langsame Datenaustausch (Kommunikation) zwischen den Prozessoren für Zwischenergebnisse ist zu berücksichtigen. Man bezeichnet diese Architektur auch als *Multicomputer-Systeme*. In der Regel hat man hier auch ein festes Betriebssystem und zueinande sehr ähnliche Prozessoren, also ein homogenes System.

Eine allgemeinere Variante dieser Architektur wurde durch das Internet interessant, wo man Rechner von verschiedenen Orten als *verteilte Systeme* für gemeinsame Aufgaben einsetzt (Stichwort *Grid Computing*). Hier ist die Kommunikation noch langsamer und der Einsatz lohnt sich nur dann, wenn viele unabhängige Teilaufgaben zu bearbeiten sind und ein recht geringer Anteil der verwendeten Daten ausgetauscht werden muss (z.B. Primzahlsuche, SETI@home-Projekt). Zusätzliche Komplikationen ergeben sich daraus, dass die Systeme meist heterogen sind, weil die Rechner verschiedene Betriebssysteme und unterschiedliche Leistungen haben. Dann ist es nicht immer leicht abzuschätzen, wie lange die Bearbeitungszeit für eine Teilaufgabe (*Task*) sein wird. Dies erschwert die Entwicklung von Algorithmen.

Bemerkung: In der Vorlesung wird nur selten auf spezielle Hardwareeigenschaften eingegangen. Ein Aspekt hat allerdings grundlegende Bedeutung und soll daher hier angesprochen werden.



Bei heutigen Rechnern ist der Zugriff auf den Speicher wesentlich langsamer als die Taktung der CPU. Daher müssen die oben gezeigten Skizzen, etwa bei gemeinsamem Speicher, folgendermaßen modifiziert werden. Jeder Prozessor P besitzt neben der Recheneinheit R noch einen kleinen *Cache*-Speicher C, auf den er um Größenordnungen schneller zugreifen kann als auf den eigentlichen (Haupt-) Speicher. Alle aus dem Hauptspeicher geholten Daten werden im Cache zwischengespeichert ("*cache*" vom französischen *cacher*) und können daher bei späteren Operationen schnell wiederverwendet werden, solange sie nicht durch neue Daten verdrängt wurden. Mittlerweile ist sogar eine Hierarchie von Caches üblich. Gerade bei Algorithmen der Linearen Algebra (z.B. Matrix-Vektor-Multiplikation) sieht man daher erhebliche Effizienzunterschiede je nachdem, ob die Gesamtmenge der bearbeiteten Daten in den Cache paßt oder nicht. Im ungünstigen Fall müssen Daten unter zusätzlichem Zeitaufwand mehrfach nachgeladen werden (cache-fault). Dies sollte man durch Aufteilung der Matrizen bzw. Vektoren in kleinere Tranchen vermeiden ($\rightarrow$ §3).

Zur Erläuterung der typischen Unterschiede zwischen sequentiellen und parallelen Algorithmen wird folgendes Beispiel behandelt.

Beispiel 1.1.1 (Rekursion parallelel)

Eine generelle Schwierigkeit bei der Entwicklung von Algorithmen ist, dass viele der üblichen Standardverfahren schrittweise vorgehen und dabei viele Abhängigkeiten von Zwischenergebnissen exisitieren. Eine Parallelisierung kann dadurch verhindert werden, dass eine Rechenoperation das Ergebnis einer direkt davor auszuführenden verwendet. Ein Paradebeispiel hierfür ist die lineare Rekursion ($a_i, b_i \in \mathbb{R}$) mit $x_1 := a_1$ und

$$x_i := b_i x_{i-1} + a_i, \quad i = 2, \ldots, n. \tag{1.1.1}$$

Für $b_i \equiv 1$ wird einfach eine Summe berechnet. Anscheinend ist hier eine Berechnung von x_n erst dann möglich, wenn x_{n-1} bekannt ist. Dies würde einen Zeitbedarf von n Zeitschritten bedeuten. Tatsächlich ist aber eine Berechnung in $\log_2 n$ Zeitschritten möglich, wenn man n Prozessoren verwendet. Der Hintergrund dabei ist eine explizite Darstellung der Lösung von (1.1.1). Anhand der ersten Ergebnisse $x_2 = a_2 + b_2 a_1$, $x_3 = a_3 + b_3 a_2 + b_3 b_2 a_1$, $x_4 = a_4 + b_4 a_3 + b_4 b_3 a_2 + b_4 b_3 b_2 a_1$ erkennt man die allgemeine Darstellung

$$x_k = \sum_{j=1}^{k} \prod_{i=j+1}^{k} b_i a_j. \tag{1.1.2}$$

Auf dieser Formel beruht das Verfahren der *rekursiven Verdopplung*, das jetzt für $n = 8 = 2^3$ vorgeführt wird. Denn das Ergebnis x_8 läßt sich in folgender Weise klammern

$$x_8 = (a_8 + b_8 a_7) + (b_8 b_7)(a_6 + b_6 a_5) + (b_8 b_7)(b_6 b_5)\big((a_4 + b_4 a_3) + (b_4 b_3)(a_2 + b_2 a_1)\big). \tag{1.1.3}$$

Es kann mit $p = 7$ Prozessoren schon in $3 = \log_2 8$ Schritten berechnet werden unter der vereinfachenden Annahme, dass pro Schritt eine Multiplikation oder eine `axpy`-Operation der Form

$y := ax + y$ durchgeführt werden kann. Im Computer werden Variablen einfach überschrieben, hier werden Zwischenergebnisse mit Strichen unterschieden:

Start	$a_1,\quad b_2, a_2,$	$b_3, a_3,\quad b_4, a_4,$	$b_5, a_5,\quad b_6, a_6,$	$b_7, a_7,\quad b_8, a_8$
1.Schritt	$a'_2 := a_2 + b_2 a_1,$	$a'_4 := a_4 + b_4 a_3,$ $b'_4 := b_4 b_3$	$a'_6 := a_6 + b_6 a_5,$ $b'_6 := b_6 b_5$	$a'_8 := a_8 + b_8 a_7$ $b'_8 := b_8 b_7$
2.Schritt		$a''_4 := a'_4 + b'_4 a'_2,$		$a''_8 := a'_8 + b'_8 a'_6$ $b''_8 := b'_8 b'_6$
3.Schritt				$a'''_8 := a''_8 + b''_8 a''_4$

(1.1.4)

Der Ablauf entspricht also der folgenden Klammerung in (1.1.3):

$$x_8 = \underbrace{\underbrace{\underbrace{(a_8 + b_8 a_7)}_{a_8} + \underbrace{b_8 b_7}_{b'_8}\underbrace{(a_6 + b_6 a_5)}_{a'_6}}_{a''_8} + \underbrace{\underbrace{b_8 b_7}_{b'_8}\underbrace{b_6 b_5}_{b'_6}}_{b''_8}\underbrace{\big((a_4 + b_4 a_3) + \underbrace{b_4 b_3}_{b'_4}\underbrace{(a_2 + b_2 a_1)}_{a'_2}\big)}_{a''_4}}_{a'''_8}.$$

Bei diesem einfachen Beispiel sind schon einige typische Eigenschaften zu erkennen, auf die man bei Parallelalgorithmen oft stößt:

- die sequentielle Rekursion (1.1.1) benötigt $n-1$ `axpy`-Operationen, die rekursive Verdopplung dagegen $(n-1) + (\frac{n}{2} - 1) + (\frac{n}{4} - 1) + .. = 2n - ..$, also ungefähr doppelt so viele. Der parallele Algorithmus hat also eine höhere sequentielle *Laufzeit* als der sequentielle.
- selbst wenn noch mehr Prozessoren vorhanden sind, ist eine weitere Laufzeitverkürzung unter $\mathcal{O}(\log_2 n)$ kaum zu erreichen, der Algorithmus hat also einen begrenzten *Parallelisierungsgrad*.
- die 7 benötigten Prozessoren werden nur im ersten Schritt alle verwendet, im 2.Schritt sind es nur noch drei, das Endergebnis errechnet nur noch ein einzelner Prozessor. Dies ist eine ungünstige *Lastverteilung*.

Alle diese Punkte treten i.d.R. bei großen, komplex aufgebauten Problemen auf und verhindern meist, dass man bei Verwendung von p Prozessoren nur das $1/p$-fache der ursprünglichen Laufzeit benötigt. Die in diesem Zusammenhang üblichen Begriffsbildungen werden im nächsten Abschnitt §2 besprochen.

Bemerkung: Bei der Darstellung wurde nicht die Kommutativität der reellen Zahlen ausgenutzt. Daher ist der Algorithmus auch mit Vektoren $a_j, x_j \in \mathbb{R}^m$ und Matrizen $b_j \in \mathbb{R}^{m\times m}$ einsetzbar. Hier ist der Aufwand für (1.1.1) ca. $\mathcal{O}(2nm^2)$. Im parallelen Verfahren (1.1.4) gibt es die gleiche Anzahl von `axpy`-Operationen, aber auch noch ca. n Matrix-Multiplikationen mit einem seq. Aufwand von je $2m^3$, zusammen also etwa das $(m+1)$-Fache. Allerdings sind jetzt doch mehr als $n-1$ Prozessoren einsetzbar und die parallele Laufzeit kann doch kleiner sein. Parallel-Algorithmen für die Matrix-Multiplikationen werden in §3 behandelt.

Die Diskussion bestimmter aktueller Rechnerstrukturen steht nicht im Vordergrund der Vorlesung, da deren Bedeutung von der Verfügbarkeit am Ort abhängt und durch technische Entwicklungen bald wieder überholt sein könnte. Dagegen wird das Hauptgewicht der Vorlesung darauf liegen, für wichtige Problemklassen und Algorithmen die Parallelisierbarkeit auf einigen typischen (abstrakten) Rechnerstrukturen zu untersuchen bzw. besser parallelisierbare Alternativen zu diskutieren.

2 Grundbegriffe

Bei Entwurf und Analyse von Parallel-Algorithmen beobachtet man schnell einige grundlegende Zusammenhänge und neue Schwierigkeiten, die zu speziellen Begriffsbildungen geführt haben. Diese werden jetzt angesprochen.

2.1 Rechner-Architekturen

Bei Ablauf eines Programms werden verschiedene Operationen mit unterschiedlichen Daten vorgenommen, bei Parallelverarbeitung sind unterschiedliche Versionen möglich. Je nach Anzahl der zu jedem Zeitpunkt im Rechner eingesetzten Größen ist es üblich, folgende Klassifikation der Architekturen vorzunehmen ("Single/multiple instruction, single/multiple data"):

	Einzel-Daten	Datensätze
Gemeinsame Anweisung	SISD von-Neumann	SIMD Vektorrechner, GPU
Beliebige Anweisungen	MISD ??	MIMD verteilte Systeme

Verarbeitungsmodi (pro Zeiteinheit)

Dabei erfordert die Art der Unterbringung der Datensätze in der rechten Spalte eigentlich eine weitere Unterteilung in Rechner mit gemeinsamem oder verteiltem Speicher. Wie erwähnt ist der Entwurf von Algorithmen bei gemeinsamem Speicher einfacher. Auf Herstellerseite liegt die Schwierigkeit in der Entwicklung schneller Schaltwerke für den Speicherzugriff bei vielen Prozessoren (teure Technologie). Rechner mit verteiltem Speicher sind meist kostengünstiger herzustellen (Extremfall PC-Netz, Grid), die wesentlichen Schwierigkeiten verlagern sich dann auf die Programmentwicklung.

2.1.1 SIMD-Rechner

Bei Basis-Algorithmen der Linearen Algebra treten oft gleichartige Operationen mit großen Datenmengen auf. Dies rechtfertigt das SIMD-Prinzip, wo jeweils die gleiche Anweisung mit vielen Daten parallel ausgeführt wird, wie es z.B. bei der Addition oder der `saxpy`-Operation ("skalar a mal x plus y", $a \cdot x + y$, $a \in \mathbb{R}$) mit Vektoren der Fall ist.

Frühe Superrechner waren *Vektorrechner*, deren Geschwindigkeitssteigerung auf dem Fließbandprinzip beruht. Dazu muß man berücksichtigen, dass eine (arithmetische) Operation aus mehreren Teilschritten aufgebaut ist und daher oft mehrere Taktzyklen benötigt.
Für die Gleitpunkt-Multiplikation sind diese rechts skizziert. Die Idee ist nun, wie beim Produktionsfließband ("Pipeline"), bei vielen Datenelementen unterschiedliche Teilschritte an unterschiedlichen Orten durchzuführen und die Operanden jeweils weiterzureichen. Nach einer gewissen Anlaufzeit (dem Auffüllen des Fließbandes) bekommt man dann in jedem Arbeitstakt ein Ergebnis. Bei großen Vektorlängen fällt die anfängliche Wartezeit nicht ins Gewicht und die Rechnerleistung beträgt beinahe ein FLOP pro Taktzyklus.

REAL-Multiplikation

Trennung Mantissen/Exponenten
Mult. Mantissen
Add. Exponenten
Normalisierung
Vorzeichenbestimmung

Die Bedeutung von Vektorrechnern ist heute gering, das SIMD-Paradigma hat aber eine ganz neue Aktualität durch die parallele Rechenleistung moderener Graphikprozessoren (GPUs) gewonnen. Diese sind hoch spezialisiert auf die Implementierung von Graphik-Algorithmen bei 3D-Szenen (Rendering, Anti-Aliasing, Objekt-Sichtbarkeit, etc.), welche sehr stark parallelisierbar sind. Im Vergleich zur CPU besitzen die Graphikprozessoren einen sehr geringen Funktionsumfang und wenig Cache-Speicher, durch Parallelverarbeitung ($\leq$ 1000 Recheneinheiten) aber eine vielfach höhere Rechenleistung (aktuell bis 2 TFLOPS). Daher werden Graphikprozessoren mittlerweile auch massiv für numerische Berechnungen eingesetzt (GPGPU=*General Purpose Computation on Graphics Processing Unit*) und sogar beim Bau von Supercomputern verwendet.

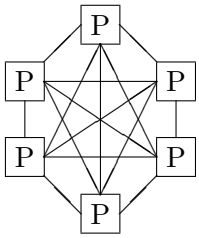
2.1.2 MIMD-Rechner mit gemeinsamem Speicher

Aus Sicht eines Programmierers sind Rechner mit gemeinsamem Speicher einfacher und im Einsatz auch flexibler. Da Programme und Daten nur einmal vorhanden sind und alle darauf zugreifen können, muß man sich nur um Parallelisierung der Operationen kümmern. Konkurrierende Schreibzugriffe regelt eine Schaltung in der Hardware. Außerdem kann das System anstehende Teilaufgaben (*Tasks*) sofort an unbeschäftigte Prozessoren vergeben. Das erlaubt auch die Verwendung einfacher, maschinenunabhängiger Programmiermodelle wie OpenMP. OpenMP stellt eine einfache Erweiterung von Standard-Programmiersprachen wie Fortran oder C++ dar, wo Nutzerkommentare als Anweisungen zur Parallelisierung interpretiert werden. So kann z.B. durch eine $OMP DO - $OMP END DO Klammer um eine Programmschleife bestimmt werden, dass die einzelnen Schleifendurchläufe auf die vorhandenen Prozessoren verteilt werden. Das Programm läßt sich aber genauso auch auf Einfach-Prozessoren ausführen.

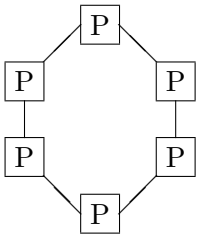
2.1.3 MIMD-Rechner mit verteiltem Speicher

Bei modernen, verteilten Rechnern mit einigen Prozessoren verbindet man diese gerne durch ein schnelles Netzwerk, das unabhängig von den Prozessoren arbeitet (DMA-Controller), und in

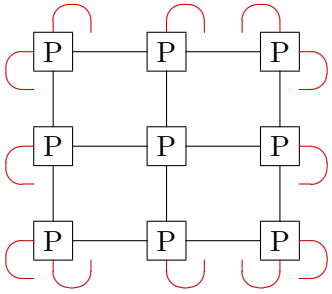
dem die Vermittlung duch einen zentralen *Switch* vorgenommen wird. Das Netz hat dann eine Sternstruktur und seine Leistung und Verfügbarkeit hängt von dem einen Switch ab. Dies ist so beim MaRC. Bei älteren Parallelrechnern waren Prozessoren direkter mit miteinander verbunden mit einer gewählten Netz-Topologie. Wichtige Beispiele werden jetzt behandelt. Bei großen modernen Rechnern mit mehreren Switches kann man dies auf deren Verbindungstopologie beziehn.



vollständig vernetzt

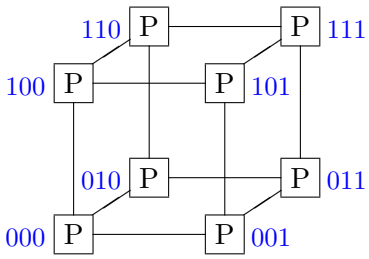


Ring



Gitter/Torus

Die vollständige paarweise Verbindung benötigt $p(p-1)/2$ Leitungen und ist bei größeren Prozessorzahlen p unpraktikabel. In den einfachsten Modellen, Ring und 2D-Gitter, dagegen hat jeder Prozessor jeweils nur zwei bzw. vier direkte Nachbarn und Verbindungen. Beim Torus sind die äußersten Prozessoren zusätzlich mit denen auf der anderen Seite verbunden (durch Bögen angedeutet). Der größte Abstand zweier Prozessoren ist allerdings $p/2$ beim Ring und ca. $2\sqrt{p}$ beim Gitter bzw. $\sqrt{p}$ beim Torus. Einen attraktiven Kompromiß zwischen beiden Extremen mit sehr flexibler Struktur stellt der Hyperwürfel (Hypercube) mit $p = 2^d$ Ecken dar. Jeder Prozessor hat $d = \log_2 p$ Verbindungen, die größte Entfernung hat den gleichen Wert.



Hyperwürfel, $p = 2^3$

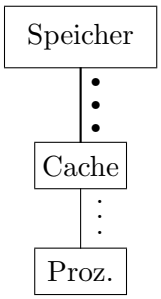
Farbig sind mögliche Adressen der Knoten angedeutet. Jeder Knoten hat d Nachbarn, deren Adresse sich um genau ein Bit von seiner eigenen unterscheidet. Insbesondere können in den Hyperwürfel auch Ring, Gitter und Torus eingebettet werden. Zur Simulation realistischer physikalischer Modelle ist bei aktuellen Hochleistungsrechnern das Netz oft ein 3D-Torus. Dies gilt etwa für den zur Zeit zweitschnellsten Rechner *Cray XT5 Jaguar* mit 224000 Rechnerkernen und einer Leistung von 1.7 PFLOPS.

Die Programmierung von Algorithmen bei Multicomputersystemen ist noch viel schwieriger als bei Multiprozessorsystemen, da Aufgaben und Daten verteilt werden müssen und neue Fehlerquellen auftreten können, welche im nächsten Abschnitt §2.2 angesprochen werden. Zur

Unterstützung der Programmentwicklung gibt es verschiedene Software. Sehr verbreitet ist das *Message Passing Interface* MPI, das Programme weitgehend von der verwendeten Hardware unabhängig macht.

2.1.4 Cache-Hierarchien

Bei modernen Rechnern gibt es einen weiteren Aspekt, der entscheidenden Einfluß auf die Laufzeiten von Algorithmen hat. Das ist der technisch bedingte langsame Zugriff auf den großen Hauptspeicher. Dieses Problem wird von Prozessorherstellern durch Einbau eines schnellen Cache auf dem Prozessorchip selber abgemildert. Wenn der Prozessor eine einzelne Zahl aus dem Speicher lesen oder dorthin schreiben will, dann geht die Anforderung zunächst an den Cache. Dessen Controler prüft v.a. beim Lesen durch schnellen Adreßabgleich, ob die Zahl dort vorhanden ist



(Cache-Treffer, *cache hit*) oder nicht (Cache-Fehler *cache fault*). Bei einem Treffer erhält der Prozessor diese Zahl und kann sie verarbeiten. Bei einem Fehlzugriff lädt der Cache-Controler die Zahl aus dem Hauptspeicher nach (und verdrängt dort andere Daten). Der Prozessor muß so lange mit der Arbeit warten oder andere Arbeiten vorziehen, sofern diese unabhängig von dem Ergebnis sind. Ein Aspekt hat dabei eine entscheidende Bedeutung für die Effizienz von Algorithmen. Beim Laden aus dem Hauptspeicher werden nicht einzelne Worte bzw. Zahlen transportiert, sondern gößere Speicherblöcke (*cache lines*). Dies ist im Diagramm durch die unterschiedliche Größe der Punkte angedeutet. Ein Programm läuft daher schneller, wenn es nahe beieinander gespeicherte Zahlen verarbeitet, die innerhalb der geladenen Cacheblöcke liegen, als wenn es weit verstreute Daten verknüpft, deren Verwendung zu vielen Cache-Fehlern führt. Diesen Aspekt muß man gerade bei Algorithmen der linearen Algebra berücksichtigen, da die Reihenfolge der Bearbeitung bei Matrix-Vektor- oder Matrix-Matrix-Operationen sehr unterschiedlich wählbar ist und Summen beliebig umgeordnet werden können. Eine Standard-Maßnahme zur Berücksichtigung von Cache-Größen ist hier die Blockbildung, wobei man beteiligte Matrizen in kleinere Blöcke unterteilt, die als Ganzes in den Cache passen.

Aus Kostengründen werden oft sogar zwei oder drei Ebenen von Caches wachsender Größe eingesetzt. Vereinfachend kann man bei Multicomputern auch den privaten Speicher des Prozessors als einen Cache betrachten, der dem nur virtuell vorhandenen Gesamtspeicher des Rechnersystems vorgeschaltet ist. Daher ähneln sich die Maßnahmen (Blockbildung) bei der Berücksichtigung von Cache und Rechnerstruktur, vgl.§3.2.

2.2 Bausteine für Parallel-Algorithmen

Bei der Diskussion von Parallel-Algorithmen kommt man früh auf Fragen, die zunächst einmal unabhängig von der konkreten Rechnerstruktur sind. Einige der dazu gehörigen Begriffe werden jetzt besprochen. Das Hauptinteresse bei parallelen Algorithmen gilt naturgemäß der Laufzeit

bzw. Komplexität in Abhängigkeit von einer Problemgröße n (Anzahl der Eingabedaten bzw. Variablen). Bei Parallelrechnern ist als zusätzlicher Parameter die Prozessoranzahl p zu berücksichtigen. Daher wird mit

$$T_p(n)$$

die Laufzeit eines Algorithmus mit n Variablen *auf* p *Prozessoren* bezeichnet. Zur Analyse betrachtet man zunächst nur die Operationszyklen unter Vernachlässigung der Kommunikation und interessiert sich meist für asymptotische Aussagen mit $n \to \infty$. Bei allgemeiner Prozessoranzahl p hängt das Ergebnis i.a. von der Relation n/p ab. Bei der linearen Rekursion (1.1.1) und dem Spezialfall der Summation ($b_i \equiv 1$) etwa können für $p < n$ jeweils Teile der Länge n/p auf den Einzelprozessoren normal berechnet, und dann durch rekursive Verdopplung zusammengefügt werden. Daher gilt für die Laufzeit dieses Algorithmus (und vieler anderer) eine Schranke der Form

$$T_p(n) = \frac{n}{p} + \log_2 p. \tag{2.2.1}$$

Das Hauptinteresse bei Parallelverarbeitung gilt natürlich dem Effizienzzuwachs im Vergleich zur sequentiellen Laufzeit $T_1(n)$. Grundlegende Begriffe wie dieser werden jetzt der Reihe nach behandelt.

Speed-up ist das Laufzeitverhältnis im Vergleich von sequentieller und paralleler Bearbeitung

$$S_p(n) := \frac{T_1(n)}{T_p(n)}. \tag{2.2.2}$$

Im Beispiel (2.2.1) hat dieses die Form

$$S_p(n) = \frac{n}{\frac{n}{p} + \log_2 p} = \frac{p}{1 + \frac{p \log_2 p}{n}}$$

und man sieht, dass das Verhältnis für $n \gg p$ tatsächlich dem Wert p nahe kommt. Dieser Grenzübergang für $n \to \infty\, (n \gg p)$ wird als *Schwerlastwert* bezeichnet. Ist die Anzahl der Prozessoren dagegen nahe bei $n \cong p$, sinkt der Wert ab auf $p/\log_2 p$, da in den letzten Schritten von (1.1.4) nur wenige Prozessoren arbeiten. Dies ist das Problem der →*Lastverteilung.*

Effizienz Skaliert man den Speed-up mit seinem Maximalwert bekommt man die Effizienz

$$E_p(n) := \frac{S_p(n)}{p} := \frac{T_1(n)}{p\, T_p(n)}.$$

Für beliebiges p liegt dieser Wert zwischen null und eins.

Lastverteilung (*load balance*) Die maximale Leistung wird natürlich nur erreicht, wenn alle Prozessoren gleichzeitig arbeiten. Dies sollte schon Ziel beim Verfahrensentwurf sein, kann aber u.U. auch durch dynamische Verteilung von Aufgaben auf gerade freie Prozessoren erreicht werden. Bei Multicomputer-Systemen ist das aber nicht ohne Weiteres möglich, wenn der freie Prozessor die benötigten Daten nicht besitzt. Bei Multiprozessor-Systemen

erfordert die Umverteilung einen geringen Zusatzaufwand, der gegen die Einsparung aufzurechnen ist. Dies hängt ab von der

Granularität Dieser Begriff beschreibt die Ebene, auf der die Parallelisierung eines bestimmten Algorithmus ansetzt, durch die Größe der parallel verarbeiteten Teilaufgaben (*Tasks*). Bei Multiprozessoren kann man sehr feinkörnig parallelisieren bis auf die Ebene einzelner arithmetischer Operationen. Bei verteilten Systemen ist wegen der anfallenden Kommunikationskosten eine dynamische Aufgabenverteilung nur in größeren Blöcken sinnvoll. Bei der rekursiven Verdopplung (1.1.4) war die Parallelisierungsebene eine `axpy`-Operation und daher feinkörnig. Bei Matrix-Vektor-Rekursionen $x_i := B_i x_{i-1} + a_i$ mit $x_i, a_i \in \mathbb{R}^m$ fallen aber grobkörnigere Teilaufgaben an, nämlich Matrix-Vektor-Multiplikationen $Bx + a$ mit $\mathcal{O}(m^2)$ Operationen, sowie Matrix-Matrix-Multiplikationen $B \cdot B$ mit $\mathcal{O}(m^3)$ Operationen, welche aber bei $p > n$ Prozessoren weiter parallelisiert werden können ($\rightarrow$ §3). Auch bei der Ausführung von Algorithmen im Rechner ist zu beachten, dass die Erzeugung und Beendung unabhängiger Teilaufgaben geringe Kosten verursacht.

Skalierbarkeit eines Algorithmus. Eine gute Skalierbarkeit bedeutet, dass das Verfahren bei sehr unterschiedlichen Prozessorzahlen gut funktioniert. Sie hängt zusammen mit der Granularität. Eine sehr feinkörnige Aufgabe (z.B. Vektor-Addition) mit großen Datenmengen skaliert natürlich gut.

Abhängigkeiten können zwischen verschiedenen Daten oder Aufgaben (Tasks) eines Algorithmus bestehen. Dies ist der Fall, wenn ein bestimmtes Ergebnis ohne ein anderes nicht berechnet werden kann, oder eine Aufgabe unbedingt nach einer anderen ausgeführt werden muß (Streichen nach dem Tapezieren). Die frühere Aufgabe blockiert die Ausführung der späteren. Im Beispiel der Rekursion 1.1.1 wurden die blockierenden Datenabhängigkeiten zwischen den x_i durch Übergang zu einem anderen Algorithmus aufgelöst.

Synchronisation Daten- oder Aufgaben-Abhängigkeiten verhindern, dass ein Algorithmus in beliebig viele unabhängige Teilaufgaben zerlegt wird. Soll ein Prozessor ein fremdes Ergebnis verwenden, dann muß er natürlich warten, bis dieses tatsächlich zur Verfügung steht, seine Tätigkeit muß also mit der des anderen Prozessors synchronisiert werden. Dabei ist wie zuvor zwischen Ablauf- und Zugriffskontrolle zu unterscheiden. Je nach Art der Speicheraufteilung können dazu unterschiedliche Methoden verwendet werden.

Der Stellenwert dieser Begriffsbildungen wird anhand einer einfachen Überlegung klar, die zeigt, dass man bei Effizienzbetrachtungen immer den gesamten Prozeß beachten muß und punktuelle Beschleunigungen von Teilalgorithmen einen nur geringen Effekt haben. Das Ergebnis ist das **Gesetz von Amdahl**. Man betrachtet dazu den realistischen Fall, dass in einem Algorithmus mit sequentieller Laufzeit $T_1 = 1$ (oBdA) nur ein Anteil $x,\ 0 \leq x \leq 1$, der Operationen parallel mit p-facher Rechenleistung durchgeführt werden kann (Beispiel "Flug Frankfurt-Berlin" mit Anfahrt, Flug, Abfahrt; Flug beschleunigen?). Die Laufzeit des Algorithmus ist also $T_p(x) =$

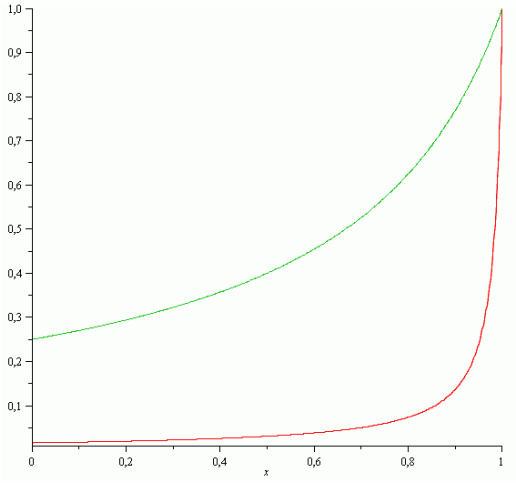
$\frac{x}{p} + \frac{1-x}{1} = 1 - x(1 - \frac{1}{p})$. Dies ergibt einen Wert von Speed-up bwz. Effizienz mit

$$S_p(x) = \frac{T_1}{T_p(x)} = \frac{1}{1 - x(1 - 1/p)}, \quad E_p(x) = \frac{1}{x + (1 - x)p}. \tag{2.2.3}$$

Für alle Prozessoranzahlen gilt daher die Schranke $S_p(x) \leq \frac{1}{1-x}$ und für große p sogar $E_p(x) \to 0$ $(p \to \infty)$. Für $p \gg 1$ hat der gezeigte Graph von E_p rechts eine steile Tangente, $E_p'(1) = p - 1$. Daher erreicht man die Spitzenbeschleunigung $S_p = p$ annähernd nur bei sehr hohen Parallelisierungsgraden. Tatsächlich bekommt man den halben Spitzenwert $S(x_h) = \frac{1}{2}S(1) = \frac{p}{2}$, $E_p(x_h) = \frac{1}{2}$ erst beim Anteil

$$x_h = \frac{p-2}{p-1} = 1 - \frac{1}{p-1}.$$

Bei $p = 101$ verliert man selbst bei einem Parallelisierungsgrad von 99% schon 50% der Spitzenleistung!



Effizienz für $p = 4$ (grün), $p = 64$ (rot).

2.2.1 Kommunikation

Wenn Aufgaben in unabhängige Teilaufgaben zerlegt werden können, erfordert die Lösung in der Regel eine Kooperation der beteiligten Prozesse, sie müssen miteinander kommunizieren. Bei **Multiprozessorsystemen** läuft die Kommunikation über *gemeinsame Variable* bzw. Datenbereiche. In parallelen Programmiersprachen kann man Variable daher entweder als private oder gemeinsame (*shared*) deklarieren. Bei gemeinsamen Variablen muß man jetzt aber beim Schreibzugriff für eine *Synchronisation* der Zugriffe sorgen, damit man nicht bei wechselndem Zeitbedarf der einzelnen Teilaufgaben (die von Zufälligkeiten abhängen können) unterschiedliche Ergebnisse bekommt. Ein solches nichtdeterministisches Verhalten nennt man zeitkritisch (*race conditions*). Es gibt verschiedene Maßnahmen zur Ablauf-Synchronisation.

Kritische Bereiche können durch eine spezielle Sperrvariable oder **Semaphor** geschützt werden. Voraussetzung für die Synchronisation ist, dass jeder Prozessor, der einen kritischen Bereich benutzen will, zunächst ein Unterprogramm `lock(s)` mit dem Semaphor s aufruft, das ihm nur dann den Zugang zum Bereich gestattet, wenn kein anderer Prozessor darauf zugreift. Erst dann kann er den kritischen Bereich nutzen und muß ihn danach wieder freigeben mit Hilfe eines Aufrufs `unlock(s)`. Mit einer booleschen Variable s als Semaphor kann die Maßnahme folgendermaßen umgesetzt werden:

```
while (lock(S)){ } //*warte
{ kritischer Abschnitt, Datenänderung }
unlock(S);
```

Voraussetzung für die Wirkungsweise ist aber erstens, dass sich jeder Prozess an den Ablauf hält und dass außerdem der Aufruf von `lock` eine unteilbare Basisoperation darstellt, die nur von einem Prozessor gleichzeitig ausgeführt werden kann. Im Java-Code

boolean lock (boolean S)
{ boolean $merke = S$; $S = true$; return $merke$; }

muß das Betriebssystem sicherstellen, dass die 3 Anweisungen nur als Einheit ausgeführt werden und für $S = false$ nicht gleichzeitig 2 Prozessoren eine fehlende Blockade feststellen und danach $S = true$ setzen und fortfahren.

Eine andere Methode der Ablaufsynchronisation ist eine **Barriere** im Programm, an der jeder Prozeß solange blockiert wird, bis der letzte dort angekommen ist. Damit kann man z.B. die vollständige Bearbeitung von Daten vor ihrer Weiterverwendung sicherstellen. Die Barriere ist vor allem bei statischer Aufgabenverteilung (d.h. einer bei Kompilierung bekannten) mit gleich großen Teilaufgaben nützlich. Ein typisches Beispiel ist eine parallel ausführbare Laufanweisung (z.B. bei Vektoraddition)). Hier ist das Schleifenende eine Barriere, die sicherstellt, dass die Vektorsumme vollständig berechnet wurde. Diese Konstruktion kann bei ungünstiger Lastverteilung aber die Leistung beeinträchtigen, da alle Prozesse auf den langsamsten warten müssen.

Bei **Multicomputern** mit verteiltem Speicher ist die Kommunikation über das Netzwerk schon ein grundlegender Bestandteil des Systems, Zugriffe auf fremde Daten können nur über einen Nachrichtenaustausch über das Verbindungsnetzwerk erfolgen. Auf Programmebene werden daher zumindest zwei elementare Zusatzoperationen benötigt,

sende(p,x); empfange(p,x);

Dabei schickt `sende(p,x)` das Datenpaket x an den Prozessor mit der Nummer p und bei `empfange(p,x)` wartet der ausführende Prozeß auf das Eintreffen des Datenpakets x vom Prozessor p. Die Abwicklung wird hier nicht genauer spezifiziert, da sie stark vom konkreten Rechnermodell abhängt. Auch die Kosten der Übermittlung lassen sich schlecht allgemein beschreiben. Als einfache Vorstellung kann man aber eine affin lineare Abhängigkeit $a + bG + cE$ von der Größe G des Datenpakets und der Prozessorentfernung E im Netzwerk annehmen. Der feste Anteil a beschreibt den Verwaltungsaufwand für die Kommunikation. Alle diese Zeitkonstanten liegen i.d.R. weit über denen innerhalb von Prozessoren. Bei wichtigen Basisalgorithmen kann eine einfachere Kommunikation mit direkten Nachbarn im Netz ausreichen. Die Nachrichtenübermittlung stellt natürlich auch automatisch eine Ablauf-*Synchronisation* her, da eine Nachricht nicht empfangen werden kann, bevor sie gesendet wurde. Die Warteschleife im oben beschriebenen `lock`-Kommando wird durch das Warten bei `empfange` ersetzt.

Der Algorithmen-Entwurf für Multicomputer ist offensichtlich schwieriger als bei Multiprozessoren da auch die Aufteilung der Daten auf Prozessoren, also die Datenstrukturen anzupassen sind. Denn diese haben einen wesentlichen Einfluß auf den Kommunikationsaufwand.

3 Basis-Algorithmen der Linearen Algebra

Ein wesentliches Hindernis für die parallele Ausführung von Teilen eines Algorithmus sind *Abhängigkeiten* zwischen den verwendeten Daten. Numerische Algorithmen werden oft in iterativer und sequentieller Formulierung formuliert. Das einfachste Beispiel ist die Berechnung der Summe $s := \sum_{i=1}^{n} a_i$ durch die Rekursion

double $s = a[1]$;
for $(i = 2\,..\,n)$ { $s = s + a[i]$; }

Diese Darstellung suggeriert fälschlicherweise, dass nur die sequentielle Berechnung in Frage kommt. Im Bsp 1.1.1 wurde aber gezeigt, dass sogar für die allgemeinere Form der linearen Rekursion eine parallele Berechnung möglich ist, welche sich bei der Summe durch die Klammerung

$$(\ldots((a_1 + a_2) + (a_3 + a_4)) + ((a_5 + a_6) + \ldots)$$

veranschaulichen läßt. Die Datenabhängigkeit der Zwischenergebnisse in s war also nicht schon im Problem vorhanden, sondern wurde erst im speziellen Algorithmus über die Hilfsvariable s eingeführt.

Um bei anderen Problemen Ansatzpunkte für die Parallelisierung zu finden, ist es daher wesentlich, sich von konkreten Algorithmen zu lösen und übergeordnete Strukturen zu erkennen. Bei einigen aufwändigen, aber einfachen Basis-Operationen wie Vektor-Produkt, Matrix-Vektor- und Matrix-Matrix-Multiplikation werden im Folgenden naheliegende Ansätze zur Parallelisierung diskutiert. Zunächst wird aber eines der wichtigsten Standardverfahren betrachtet, welches nicht ganz offensichtlich parallelisierbar ist, nämlich den Gauß-Algorithmus für lineare Gleichungssysteme. Denn hier kann man sehen, dass die in den Grundvorlesungen behandelte Standardversion nur eine von vielen möglichen ist, welche unterschiedliche Eigenschaften aufweisen.

3.1 Verschiedene Formulierungen des Gauß-Algorithmus

Zunächst geht es in diesem Abschnitt darum, wichtige Varianten der Gauß-Elimination zu diskutieren, um alle Alternativen zu kennen. Die Umsetzung auf verschiedenen Parallel-Architekturen wird später in §4 behandelt.

3.1.1 Auflösung von Dreiecksystemen

Das Endergebnis der Gauß-Elimination ist die Zerlegung der gegebenen Matrix $A = LR$ in das Produkt einer unteren Dreieckmatrix L mit Einsen in der Hauptdiagonale und einer oberen Dreieckmatrix R. Die Tatsache, dass dabei elementare Zeilenoperationen eingesetzt werden, tritt aber jetzt in den Hintergrund. Die Auflösung des Systems $Ax = b$ erfolgt über die beiden

Dreiecksysteme $Ly = b$ und $Rx = y$. Zunächst wird das erste System $Ly = b$ betrachtet

$$\begin{array}{ccccccc} y_1 & & & & = & b_1 \\ l_{21}y_1 & +y_2 & & & = & b_2 \\ \vdots & \vdots & \ddots & & & \vdots \\ l_{n1}y_1 & +l_{n2}y_2 & \cdots & +y_n & = & b_n \end{array} .$$

Seine Lösung kann in sequentieller Weise durch die Rekursion

$$y_i := b_i - \sum_{j=1}^{i-1} l_{ij}y_j, \quad i = 1, \ldots, n, \tag{3.1.1}$$

erfolgen. In Form eines Programmteils heißt das

```
for (i = 1..n) {
   y[i] = b[i];
   for (j = 1..i − 1) { y[i] = y[i] − l[i][j] ∗ y[j]; }
}
```

Die Summation in der j-Schleife wird dabei i.a. in einer Hilfsvariablen durchgeführt. Ähnlich wie bei der einfachen linearen Rekursion erscheint eine Parallelisierung auf den ersten Blick unwahrscheinlich. Ein halbherziger Ansatz wäre die Berechnung der Summe in (3.1.1) durch die rekursive Verdopplung. Dagegen führt eine einfache Vertauschung der beiden Laufanweisungen (und das Überschreiben von b durch y) auf folgenden übersichtlichen Algorithmus

```
for (j = 1 .. n − 1) {
   for (i = j + 1 .. n) { b[i] = b[i] − l[i][j] ∗ b[j]; }
}
```
(3.1.2)

In dieser Version sind die Operationen in der i-Schleife unabhängig voneinander und können daher parallel ausgeführt werden. Diese zweite Zeile in (3.1.2) hat die Struktur einer `saxpy`-Operation $y := ax + y$ (*skalar-a x plus y*) mit Skalar $a = b_j$ und Vektoren $y = (b_{j+1}, \ldots, b_n)^T$, $x = (l_{j+1,j}, \ldots, l_{nj})^T$, die allerdings für wachsendes j immer kürzer werden. Bei Verwendung von $\mathcal{O}(n)$ Prozessoren ist daher die Lastverteilung nicht optimal.

3.1.2 Erzeugung der LR-Zerlegung

Die Elimination zur Erzeugung der oberen Dreieckform im Gauß-Algorithmus kann man kompakt (mit Überschreiben der Matrixelemente a_{ij} von A) in der folgenden Form beschreiben,

$$\left\{ a_{ij} := a_{ij} - \frac{a_{ik}a_{kj}}{a_{kk}}, \quad i, j = k+1, \ldots, n, \right\} \; k = 1, \ldots, n-1. \tag{3.1.3}$$

Dabei wurde natürlich voraussgesetzt, dass die auftretenden Pivotelemente $a_{kk} = a_{kk}^{(k)}$ genügend groß sind. Das Element $l_{ik} = a_{ik}/a_{kk}$ des Dreieckfaktors L wird üblicherweise unter der Hauptdiagonalen an der Stelle von a_{ik} abgespeichert. Dann lautet die Anweisung in (3.1.3) einfach

$$a_{ij} := a_{ij} - a_{ik}a_{kj}$$

und wird durch 3 Schleifen mit den Indizes i, j, k gesteuert. Es ist eine wichtige Erkenntnis, dass man bei der Abfolge der Anweisungen wesentlich mehr Möglichkeiten hat, als man zunächst vermutet. Es hat sich dabei eingebürgert, die Varianten des Gauß-Arlgorithmus durch die Reihenfolge der Schleifen zu charakterisieren. So wird die Variante (3.1.3) (ohne Hilfsvariable) realisiert durch folgende Anweisungen.

KIJ-Gauß:

```
for (k = 1 .. n − 1) {
  for (i = k + 1 .. n) { a[i][k] = a[i][k]/a[k][k]; }
  for (i = k + 1 .. n) {
    for (j = k + 1 .. n)
    { a[i][j] = a[i][j] − a[i][k] ∗ a[k][j]; }
}}
```

Die Divisionen in der i-Schleife wurde hier aus der i, j-Doppelschleife herausgezogen. So ist wie in (3.1.3) erkennbar, dass alle Anweisungen dieser Doppelschleife parallel durchführbar sind. Dies erkennt man noch besser, wenn man einen Schritt der äußeren k-Schleife in Matrixschreibweise betrachtet. Mit $s := (0, .., 0, a_{k+1,k}, .., a_{nk})^T$ und $z := (0, .., 0, a_{k,k+1}, .., a_{kn})$ lautet) dieser Schritt in (3.1.3 so:

$$A := A - \big(\frac{1}{a_{kk}}\ s\big)\ z^T, \tag{3.1.4}$$

die Änderung der aktuellen Teilmatrix entspricht also einer Rang-1-Korrektur (äußeres Vektorprodukt). Die Elemente können bei diesem Schritt in beliebiger Reihenfolge bestimmt werden.

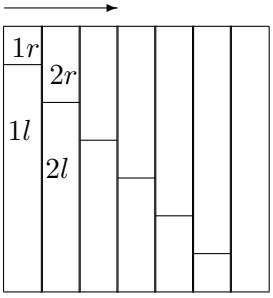
Daher kann man analog zur Auflösung von Dreiecksystemen andere Varianten nach Vertauschung der Schleifenreihenfolge betrachten. Beim Gauß-Algorithmus (3.1.4) können offensichtlich die i- und j-Schleifen vertauscht werden ($\rightarrow$ KJI-Variante). Größere Umformungen erfordern jedoch weitergehende Überlegungen. Endergebnis des Gauß-Algorithmus ist die LR-Zerlegung $A = LR$. Dadurch berechnet man tatsächlich die Lösung des Gleichungssystems

$$a_{ij} = \sum_{k=1}^{\min\{i,j\}} l_{ik} r_{kj}, \tag{3.1.5}$$

in das die Dreieckstruktur von L und R eingeht. Die Reihenfolge, in der man nach den Unbekannten $l_{ik}, i > k$, und $r_{kj}, k \le j$, läßt sich aber in großem Umfang variieren. Üblicherweise überschreibt man die Elemente a_{ij} dabei durch die genannten Werte. Für $j \ge i$, z. B., kann man aus der Bedingung $a_{ij} = r_{ij} + \sum_{k=1}^{i-1} l_{ik} r_{kj}$ nur dann r_{ij} bestimmen, wenn alle Werte $l_{i1}, \ldots, l_{i,i-1}$ und $r_{1j}, \ldots, r_{i-1,j}$ schon bekannt sind. Eine Variante ist die *spaltenweise* Berechnung:

JIK-Gauß:

```
for (j = 1 .. n) {
  for (i = 1 .. j) { a_ij = a_ij − Σ_{k=1}^{i−1} a_ik a_kj; }
  for (i = j + 1 .. n) { a_ij = (a_ij − Σ_{k=1}^{j−1} a_ik a_kj)/a_jj; }
}
```



Dabei werden in der 2. Zeile zunächst die Elemente r_{ij} berechnet, in der 3.Zeile dann die l_{ij}. Der Ablauf ist im Diagramm angedeutet. Verlegt man die Division durch a_{jj} in eine getrennte Schleife, kann man sogar beide i-Schleifen zu einer einzigen zusammenfassen, die dann von $1 \ldots n$ durchläuft. Weitere Vorteile der spaltenweisen Bearbeitung hängen ab vom verwendeten Compiler. Bei FORTRAN werden zweidimensionale Felder spaltenweise gespeichert. Daher liegen die Elemente einer Spalte in aufeinander folgenden Speicheradressen und können daher schneller angesprochen werden. Ein moderner Aspekt ist hier, dass sie dann auch zusammen im *Cache* gehalten und mglw weniger oft geladen werden. Bei normalen Rechnern hat das Verfahren in dieser Form den weiteren Vorteil, dass man die Summen leicht mit doppelter Genauigkeit berechnen kann ($\rightarrow$ Verfahren von Crout). Allerding läßt sich gerade diese Summenbildung schlecht parallelisieren.

Analog zu (3.1.2) kann man die Summen aber auch spaltenweise bilden. Dies führt sogar auf ein verblüffend einfaches Verfahren. Dazu betrachtet man im j-ten Schritt der JIK-Variante die j-te Spalte der beiden vereinigten Matrizen der LR-Zerlegung und betrachtet zur Verdeutlichung deren gesamte j-te Spalte vor der Division, d.h.

$$y := (r_{1j}, .., r_{jj}, l_{j+1,j} r_{jj}, .., l_{nj} r_{jj})^T.$$

Dann sieht man, dass man ohne Division in diesem Schritt die Lösung des Dreiecksystems

$$\begin{array}{llllllll}
y_1 & & & & & & = & a_{1j} \\
a_{21}y_1 & +y_2 & & & & & = & a_{2j} \\
\vdots & \vdots & \ddots & & & & & \vdots \\
a_{j1}y_1 & +a_{j2}y_2 & \cdots + a_{j,j-1}y_{j-1} & +y_j & & & = & a_{jj} \\
\vdots & \vdots & \vdots & & \ddots & & & \vdots \\
a_{n1}y_1 & +a_{n2}y_2 & \cdots + a_{n,j-1}y_{j-1} & +0 & \cdots & +y_n & = & a_{nj}
\end{array}$$

berechnet. Die Koeffizienten a_{ik} dieses Systems auf der linken Seite sind dabei schon die Werte aus den ersten $j-1$ Spalten der LR-Zerlegung, während rechts tatsächlich die j-te Spalte der Ausgangsmatrix A steht. Die Auflösung dieses Dreiecksystems in der Form (3.1.2) ergibt die letzte Variante, den

JKI-Gauß:

```
for (j = 1 .. n) {
   for (k = 1 .. j − 1) {
      for (i = k + 1 .. n) { a[i][j] = a[i][j] − a[i][k] ∗ a[k][j]; }
   }
   for (i = j + 1 .. n) { a[i][j] = a[i][j]/a[j][j]; }
}
```

Es wurden hier die drei Varianten, KIJ, JIK und JKI, des Gauß-Algorithmus besprochen, die alle das gleiche Ergebnis, die LR-Zerlegung liefern. Allerdings wurde auch schon angemerkt, dass die Varianten je nach verwendetem System (Compiler und Hardware) unterschiedlich effizient sein können. Dies gilt insbesonder im Hinblick auf Parallel-Verarbeitung und wir werden

uns später tatsächlich mit der Durchführung auf verschiedenen Rechnertypen beschäftigen. Entscheidend sind dabei unterschiedliche Charakteristiken bei Anzahl und Verteilung der parallel durchführbaren Operationen, aber auch die Verteilung der Daten bei Multicomputern.

3.2 Matrix-Vektor-Multiplikation

Einfachere Verfahren als der Gauß-Algorithmus sind die Standardoperationen wie Matrix-Vektor-Multiplikation. Da diese in Algorithmen oft als Teilschritt auftritt und das Ergebnis dann weiter verarbeitet wird, betrachtet man meist die `gaxpy`-Operation $z := Ax + y$ (*Groß-A x plus y*), die jetzt behandelt wird. Auch die inneren Schleifen bei den vektororientierten Varianten (3.1.2) und dem JKI-Gauß ähneln der entsprechenden Variante der `gaxpy`-Operation. Zunächst kann man sich abstrakt überlegen, wie die minimalen Laufzeiten bei Nutzung (beliebig) vieler Prozessoren aussehen. Da bei einer $m \times n$-Matrix A die m Ergebniskomponenten unabhängig voneinander sind und die zeilenweisen Summen mit n Prozessoren in $\mathcal{O}(\log n)$ Schritten gebildet werden können hat man folgende Aussagen

Prozessoren	1	m	$\mathcal{O}(mn)$
Laufzeit	$\mathcal{O}(mn)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

(3.2.1)

Bei großen Matrizen und Multiprozessor-Systemen mit $p \ll n$ ist die Parallelisierung durch eine geeignete Unterteilung der Matrix recht einfach durchzuführen. Bei Multicomputern ist für große Matrizen dagegen auch die Aufteilung auf die Prozessoren entscheidend.

3.2.1 Gaxpy bei gemeinsamem Speicher

Für Prozessoranzahlen $p \ll n$ werden bei einer $n \times n$-Matrix A nur zwei einfache Partitionierungen betrachtet, eine zeilen- und eine spaltenorientierte. Andere Varianten, wie Blockzerlegungen werden zurückgestellt. Zur Vereinfachung sei $n = pr,\ r \in \mathbb{N}$. Dann läßt sich die Anweisung $z := Ax + y$ zeilenweise zerlegen in der Form

$$\begin{pmatrix} z_1 \\ \vdots \\ z_p \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_p \end{pmatrix} + \begin{pmatrix} A_1 \\ \vdots \\ A_p \end{pmatrix} x, \tag{3.2.2}$$

mit Teilvektoren $z_j, y_j \in \mathbb{R}^r$ und $A_j \in \mathbb{R}^{r \times n}$. Falls p kein Teiler von n ist, wird für einige Teilvektoren und -Matrizen die Dimension um eins vergrößert. Für eine kompakte Formulierung von Blockoperationen wird die Bezeichnung $x[i\,..\,j]$ für den Teilvektor mit den Indizes von i bis $j > i$ einschließlich eingeführt. Dann kann man die Operation (3.2.2) durch die Anweisung

for $(j = 1\,..\,n)$ {
$z[1 + (j-1)r\,..\,jr] = z[1 + (j-1)r\,..\,jr] + A[1 + (j-1)r\,..\,jr][1\,..\,n] * x[1\,..\,n];$ }

umsetzen. Diese stellt ein vollständig parallelisierbares Verfahren dar, denn es gibt keine Schreib-

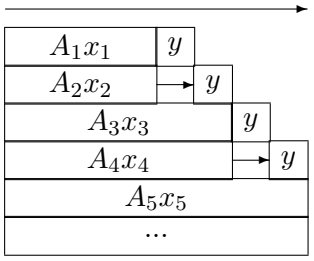
konflikte bei z, nur x wird von allen Prozessoren gleichzeitig gelesen (Zeitbedarf für Mehrfachkopien?).

Wenn eine spaltenweise Zerlegung der Matrix A gewünscht ist (evtl. aus übergeordneten Gründen), hat die `gaxpy`-Operation die Darstellung

$$z := y + A_1x_1 + \ldots + A_px_p.$$

Bei einer Zerlegung von A (spaltenweise) und x in gleichgroße Teile können die Teilprodukte wieder parallel berechnet werden, zur Summenbildung könnte man Kaskadensummation einsetzen. Da aber die Summation der p Summanden in einen größeren Rahmen stattfindet, kann man die Kollision bei der Summation auch durch eine ungleichförmige Zerlegung von A vermeiden. Dazu sei jetzt A_j eine $n \times n_j$-Matrix. Dann kann man die Wartezeiten bei der Addition $z = (..((y+A_1x_1)+A_2x_2)+..)+A_px_p$ dadurch überbrücken, dass n_j immer größer gewählt wird.

Bei der Diskussion wird berücksichtigt, dass eine Vektoraddition $y := y + s$ genau n Operationen, die Teilmultiplikation A_jx_j aber $2nn_j$ Operationen benötigt. Die erforderliche Wahl der n_j geht aus dem *Laufzeitdiagramm* rechts hervor ("Wettrennen"), die Prozessoren sind untereinander angeordnet. Wenn man zuläßt, dass jeder zweite Prozessor einen Schritt wartet (kurze Pfeile), sollte die Teildimension n_j alle zwei Schritte um eins erhöht werden.



Die Koordination bei den Additionen kann durch einen Semaphor (vgl. §2.2) erzwungen werden, der die Variable y bzw. z gegen Mehrfachschreibzugriffe schützt und dadurch die im Diagramm durch Pfeile angedeuteten Wartezeiten erzwingt. Beim Algorithmus wird für eine gerade Prozessorzahl p also $n_{2k-1} = n_{2k} = n_1 + k - 1$ gesetzt. Dies führt über die Forderung

$$n = \sum_{j=1}^{p} n_j = 2\sum_{k=1}^{p/2}(n_1 - 1 + k) = p(n_1 - 1) + \frac{p}{2}(\frac{p}{2} + 1)$$

auf die Wahl

$$n_1 = \frac{n}{p} + \frac{1}{2} - \frac{p}{4} < \frac{n}{p}, \quad n_p = \frac{n}{p} + \frac{p}{4} - \frac{1}{2}.$$

Man startet natürlich mit einem unterdurchschnittlich großen Block. Die Gesamtlaufzeit des Algorithmus ist $T_p(n) = 2n(n_p+1) = n(2\frac{n}{p} + \frac{p}{2} + 1)$. Im Vergleich mit der sequentiellen Laufzeit $T_1(n) = 2n^2$ bekommt man die Beschleunigung

$$S_p(n) = \frac{T_1(n)}{T_p(n)} = \frac{2n^2}{n\big(2\frac{n}{p} + \frac{p}{2} + 1\big)} = \frac{p}{1 + (p^2 + 2p)/4n}.$$

Daher ist die Effizienz $E_p = S_p/p \cong 1/(1 + p^2/(4n))$ also doch etwas schlechter als die bei der Kaskadensummation mit $1/(1 + (p \log_2 p)/n)$, vgl. (2.2.1). Bei kleineren Prozessoranzahlen $p^2 < n$ ist das gerade besprochene Verfahren aber sicher einfacher zu realisieren.

3.2.2 Gaxpy bei verteiltem Speicher

Bei Multicomputern sind nicht nur die Tätigkeiten, sondern auch die Daten auf mehrere Prozessoren zu verteilen. Wenn man zunächst das einfache Problem betrachtet, einen Vektor x der Länge n auf $p = n/r$ Prozessoren zu verteilen, sieht man wie bei der Ausgabe eines Kartenspiels zwei offensichtliche Möglichkeiten. Die erste ist die Verteilung in p Blöcken der Länge r, die andere die zyklische. Im folgenden werden Teilvektoren jeweils durch Indizes identifiziert. Mit der Bereichsnotation von oben lauten die beiden Varianten so:

$$\text{Block:} \qquad x^T = \Big(x[1\,..\,r], x[r+1\,..\,2r], \ldots, x[(p-1)r+1\,..\,pr]\Big), \tag{3.2.3}$$

$$\text{zyklisch:} \qquad x^T = \Big(x[1], x[p+1], \ldots \Big| \; x[2], x[p+2], \ldots \Big| \ldots \Big| \; x[p], x[2p], \ldots \Big). \tag{3.2.4}$$

Diese Aufteilung kann man so interpretieren, dass der Gesamtvektor $x[1\,..\,pr]$ spaltenweise als eine $r \times p$-Matrix angeordnet wird, wo jede Spalte einem der Prozessoren zugeordnet wird, bzw. einer Anordnung als $p \times r$-Matrix, wo zu jeder Zeile ein Prozessor gehört. Während die Block-Aufteilung übersichtlicher ist, hat auch die zyklische Speicherung in bestimmten Fällen Vorteile.

Bei Matrizen können beide Zuordnungsmethoden unabhängig voneinander für Zeilen und Spalten eingesetzt werden. Diese führt also auf vier unterschiedliche Speichermethoden. Falls keine speziellen Gründe dagegen sprechen, wird vor allem die blockweise Verteilung verwendet. Bei dieser kann man die `gaxpy`-Operation schon auf einem sehr einfachen Rechnernetz, dem Ring, effizient durchführen. Da sich Ringe in komplexere Rechnernetze einbetten lassen, ist dieser Algorithmus auch von allgemeinem Interesse. Im Verfahren erfolgt Kommunikation nur zwischen direkten Nachbarn und es werden keine Mehrfachkopien von Daten benötigt. Der Algorithmus beruht auf einer $p \times p$-Blockzerlegung der Matrix in Blöcke $A_{ij} \in \mathbb{R}^{r\times r}$,

$$\begin{pmatrix} z_1 \\ \vdots \\ z_p \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_p \end{pmatrix} + \begin{pmatrix} A_{11} & \ldots & A_{pp} \\ \vdots & & \vdots \\ A_{p1} & \ldots & A_{pp} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_p \end{pmatrix}. \tag{3.2.5}$$

Die Arbeitsweise beschreibt man am Einfachsten für jedes Indexpaar i, j durch Angabe desjenigen Bearbeitungszyklus, in dem die Teiloperation $y_i := y_i + A_{ij}x_j$ durchgeführt wird. Im Bild rechts wird für $p = 4$ die Diagonal-Reihenfolge gezeigt. Dabei werden im Prozessor i die Zeilen-Abschnitte $y_i = y[1+(i-1)r\,..\,ir]$ und $A_{i,1..p} = A[1+(i-1)r\,..\,ir][1\,..\,n]$ fest untergebracht und am Ende entsteht dort das Ergebnis $z_i = z[1+(i-1)r\,..\,ir]$.

1	4	3	2
2	1	4	3
3	2	1	4
4	3	2	1

Die n^2 Daten von A sind also fest gespeichert, dafür müssen aber die Teile des Vektors x zyklisch im Ring nach rechts verschoben werden (wie beim Ringelreihen): Nach Ausführung der Anweisung $y_i := y_i + A_{ii}x_i$ im ersten Zyklus sendet jeder Prozessor P_i seinen aktuellen Block x_i an den rechten Nachbarn $P_{1+i \bmod p}$. Der zweite Zyklus liefert dann $y_i := y_i + A_{i,i-1}x_{i-1}$,

$i = 2 \ldots p$, $y_1 := y_1 + A_{1p}x_p$, etc. Für den Aufenthaltsort der Teilvektoren bekommt man also folgendes Bild

Schritt	P_1	P_2	P_3	P_4
1	x_1	x_2	x_3	x_4
2	x_4	x_1	x_2	x_3
3	x_3	x_4	x_1	x_2
4	x_2	x_3	x_4	x_1

Der Algorithmus wird durch ein Programm beschrieben, das **jeder** Prozessor beim Ablauf auszuführen hat. Dazu muß der Prozessor zunächst in der Lage sein, seine eigene Nummer festzustellen (Unterprogramm *wer_bin_ich*). Außerdem kommuniziert er (nur) mit seinen direkten Nachbarn *rechts* und *links*. In jedem Prozessor gibt es lokale Variable (mit Namenszusatz l), die die ihm zugeteilten Abschnitte der (nicht vorhandenen) globalen enthalten. Bei Initialisierung sei im Prozessor i also der oben erwähnten Block $yl[1\,..\,r] = y[1 + (i-1)r\,..\,jr]$ und $Al = A[1 + (i-1)r\,..\,ir][1..n]$ sowie der Abschnitt $xl[1\,..\,r] = x[1 + (i-1)r\,..\,ir]$. Jeder Prozessor führt folgendes Programm aus

Ring-Gaxpy:

$i = wer\_bin\_ich$;
for $(t = 1\,..\,p)$ {
  $tau = i + 1 - t$;
  if $(tau <= 0)$ { $tau = tau + p$; } //* $xl = x[1 + (\tau - 1)r\,..\,\tau r]$
  $yl[1\,..\,r] = yl[1\,..\,r] + Al[1\,..\,r][1 + (tau - 1) * r\,..\,tau * r] * xl[1\,..\,r]$;
  $sende(xl, rechts)$; $empfange(xl, links)$;
}

Das Ergebnis z von (3.2.5) entsteht ebenfalls verteilt, in P_i steht am Ende $z[1 + (i-1)r\,..\,ir] = yl[1\,..\,r]$. Dieser Algorithmus hat wieder eine optimale Lastverteilung, da kein Prozessor leerläuft. Es können sogar alle zur gleichen Zeit die gleichen Anweisungen ausführen. Daher könnte dieses Verfahren auch auf einem *systolischen* Rechner ablaufen, bei dem mit einem Anweisungsstrang alle Prozessoren synchron gesteuert werden. Außerdem wird *Kommunikation* jeweils nur mit einem direkten Nachbar und über disjunkte Kanäle abgewickelt, d.h. auch alle Netzregionen werden identisch belastet. Überdies kann das Senden parallel zur Rechnung ablaufen, da nur Eingabedaten weitergereicht werden. Zur Effizienz-Betrachtung wird aber angenommen, dass Arithmetik und Senden nacheinander ablaufen, und dass die Übermittlung einer Nachricht aus k Zahlen an direkte Nachbarn die Zeit $s_N + kt_N$ in Anspruch nimmt. Dabei entspricht $1/t_N$ der Übertragungsgeschwindigkeit des Netzwerks, während s_N die Anlaufzeit bei der Übermittlung (Verbindungsaufbau, Senden der Netzadresse,..) angibt. Bei den auftretenden Zeitkonstanten muß man realistischerweise im Vergleich mit dem Zeitbedarf t_A für eine arithmetische Operation die Relation $s_N > t_N > t_A$ annehmen. Damit ergibt sich beim Algorithmus Ring-Gaxpy bei

$n = pr$ die Laufzeit und Effizienz zu

$$T_p(n) = p(2r^2 t_A + s_N + r t_N) \;\Rightarrow$$
$$E_p(n) = \frac{2n^2 t_A}{p^2(2r^2 t_A + s_N + r t_N)} = \Big(1 + \frac{t_N}{2t_A}\frac{1}{r} + \frac{s_N}{2t_A}\frac{1}{r^2}\Big)^{-1}.$$

Man sieht, dass bei fester Prozessorzahl p die Effizienz allein von der Blockgröße r, also von der *Granularität* des Problems abhängt. Analog zur Diskussion aus §2.2 kann man überlegen, bei welcher Blockgröße r die Effizienz, etwa auf 90% abgesunken ist. Dies führt auf

$$1 + \frac{t_N}{2t_A}\frac{1}{r} + \frac{s_N}{2t_A}\frac{1}{r^2} \stackrel{!}{=} \frac{10}{9} \;\Rightarrow\; r_{0.9} = \frac{9}{4}\Big(\frac{t_N}{t_A} + \sqrt{\frac{t_N^2}{t_A^2} + \frac{8s_N}{9t_A}}\Big) \geq \frac{9t_N}{2t_A}.$$

Falls im Netz kein extrem aufwändiges Protokoll benutzt wird ($s_N \gg t_N$), erkennt man als Faustregel für das Ring-Gaxpy also eine minimale sinvolle Blockgröße von $5t_N/t_A$.

Ein völlig anderes Bild ergibt sich, wenn die Matrix eine besondere Struktur besitzt wie untere Dreieckgestalt. In diesem Fall bliebe die Laufzeit die gleiche, aber bei nur halb so viel sequentiellem Aufwand, die Effizienz sinkt dann also auf ca. 50%. Denn im ersten Prozessor ist nur eine Blockoperation durchzuführen, während nur der letzte den vollen Aufwand hat. Doch auch hier kann man wieder eine gleichmäßige Lastverteilung erreichen indem man für die Vektoren die zyklische Aufteilung verwendet. Die Idee dabei ist es, jedem Prozessor ungefähr gleich viele kurze und lange Matrixzeilen zuzuteilen. Dies wird klar, wenn man sich ansieht wie die Dreieckmatrix nach einer Umordnung entsprechend (3.2.4) ausssieht. Die Zweckmäßigkeit der Umordnung geht auch nicht verloren, wenn A nur eine Block-Dreickstruktur besitzt. Im folgenden Diagramm ist daher durch die Sterne eine Matrix mit normaler Dreieckstruktur markiert, Punkte geben die zusätzlichen Elemente einer Matrix mit 3×3-Block-Dreieckstruktur an ($p = 3$ aus Platzgünden).

	1	2	3	4	5	6	7	8	9
1	*	.	.						
2	*	*	.						
3	*	*	*						
4	*	*	*	*	.	.			
5	*	*	*	*	*	.			
6	*	*	*	*	*	*			
7	*	*	*	*	*	*	*	.	.
8	*	*	*	*	*	*	*	*	.
9	*	*	*	*	*	*	*	*	*

	1	4	7	2	5	8	3	6	9
1	*			.			.		
4	*	*		*	.		*	.	
7	*	*	*	*	*	.	*	*	.
2	*			*			.		
5	*	*		*	*		*	.	
8	*	*	*	*	*	*	*	*	.
3	*			*			*		
6	*	*		*	*		*	*	
9	*	*	*	*	*	*	*	*	*

(Block-)Dreieckmatrix in links normaler und rechts in 3-zyklischer Numerierung

Das obige Ring-Gaxpy-Verfahren besitzt in dieser Zuordnung wieder eine gute Lastverteilung, bei Block-Dreieckstruktur ist sie sogar optimal, da dann nämlich alle permutierten Blöcke die gleiche Struktur haben. Jeder Prozessor hat nun pro Zyklus nur noch $r(r+1)$ Operationen auszufüheren. Da der Komunikationsbedarf aber gleich bleibt, ist die Effizienz etwas geringer mit $E_p(n) = \big(1 + \frac{t_N}{t_A r} + \frac{s_N}{t_A r^2}\big)^{-1}$.

3.3 Multiplikation von Matrizen

Analog zum letzten Abschnitt wird nicht die reine Matrix-Matrix-Multiplikation betrachtet sondern die kombinierte Operation

$$D := A \cdot B + C,$$

der Einfachheit halber für den quadratischen Fall $A, B, C, D \in \mathbb{R}^{n\times n}$ mit einem sequentiellen Aufwand von $2n^3$ Operationen. Auch hier kann man sich zunächst die theoretischen Schranken für große Prozessorzahlen überlegen und bekommt folgende Übersicht

Prozessoren	1	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$
Laufzeit	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

(3.3.1)

Für sehr große Matrizen werden zwei Ansätze diskutiert, der erste ist eine einfache Übertragung der Gaxpy-Algorithmen aus dem letzten Abschnitt. Ein wesentlicher Nachteil dieses Zugangs ist die bezogen auf den Gesamtaufwand jetzt recht geringe Parallelität der Verfahren, da eine Spaltenaufteilung vorgenommen wird auf $p \leq n$ Prozessoren, bei denen jeder $\geq 2n^2$ Operationen durchzuführen hat. Daher wird danach auch eine kompliziertere Implementierung auf einem zwei-dimensionalen Netz behandelt, dem Torus, bei dem bis zu $p = n^2$ Prozessoren eingesetzt werden können mit je $2n$ Operationen.

3.3.1 Block-Gaxpy

Bei einer Aufteilung der Spalten von A, C, D auf $p = n/r \leq n$ Stellen wie bei $A = (A_1, \ldots, A_p)$, und einer Blockzerlegung von B entsprechend (3.2.5), kann man dem Prozessor P_j die Berechnung des Teilergebnisses

$$D_j := C_j + \sum_{k=1}^{p} A_k B_{kj} \tag{3.3.2}$$

übertragen. Auf einem Rechner mit *gemeinsamem Speicher* (Multiprozessor) kann man dabei mögliche Zugriffskollisionen beim Lesen der A_k dadurch verhindern, dass eine verschobene Anordnung der Summanden gewählt wird (wie im Diagramm unter (3.2.5)),

$$D_j = C_j + A_j B_{jj} + \ldots + A_p B_{pj} + A_1 B_{1j} + \ldots + A_{j-1} B_{j-1,j}.$$

Dieses Verfahren wird nicht ausformuliert, da der Ablauf genau dem nächsten Algorithmus entspricht.

Bei *verteilten Systemen* (Multicomputern) reicht wieder ein Ring zur Durchführung aus. Analog zum Algorithmus Ring-Gaxpy werden dieses Mal die Spaltenblöcke A_k in einem Ringelreihen zu den verschiedenen Prozessoren geschickt. In jedem Prozessor seien wieder lokale Felder vereinbart, wobei P_i die Startwerte gemäß $blk = 1 + (i-1)r \,..\, ir$, $Al = A[1 \,..\, n, blk]$, $Bl = B[1 \,..\, n, blk]$, $Cl = C[1..n, blk]$, enthält. Nun muß wieder jeder Prozessor das Programm

Ring-Matrix-Multiplikation:

$i = wer\_bin\_ich$;
for $(t = 1 \,..\, p)$ {
  $tau := j + 1 - t$;
  if $(tau <= 0)$ { $tau = tau + p$; } //* $Al = A[1 \,..\, n][1 + (\tau - 1)r \,..\, \tau r]$
  $Cl[1 \,..\, n][1 \,..\, r] = Cl[1 \,..\, n][1 \,..\, r] + Al[1 \,..\, n][1 \,..\, r] * Bl[1 + (tau - 1) * r \,..\, tau * r][1 \,..\, r]$;
  $sende(Al, rechts)$; $empfange(Al, links)$;
}

ausführen, um am Ende in Cl das Ergebnis $D[1 \,..\, n][1 + (j-1)r \,..\, jr]$ zu erhalten. Da das Programm außer dem Schleifenende keine Entscheidungen enthält, führt jeder Prozessor *systolisch* genau die gleichen Aufgaben aus wie bei einem SIMD-Rechner. Mit den Annahmen zum Rechen- und Kommunikationsbedarf bekommt man für Laufzeit und Effizienz die Werte

$$T_p(n) = p(2nr^2 t_A + s_N + nrt_N), \quad E_p(n) = \left(1 + \frac{t_N}{2t_A r} + \frac{s_N}{2t_A nr^2}\right)^{-1}. \tag{3.3.3}$$

Im wesentlichen stimmen diese mit denen vom Ring-Gaxpy überein, nur der Einfluß von s_N ist noch geringer geworden, da in jedem Schritt größere Nachrichtenblöcke übertragen werden. Diese Größe kann aber auch nachteilig sein, denn für großes n kann der in jedem Prozessor benötigte Pufferspeicher von $3nr$ Realzahlen oder die Größe einer Nachricht, nr, durchaus den verfügbaren Rahmen sprengen. In diesem Fall muß zwingend die Granularität des Algorithmus verringert werden. Dies ist zwar auch durch feinere Aufteilung im Rahmen dieses Verfahrens möglich, attraktiver ist aber folgende Variante.

3.3.2 Matrix-Multiplikation auf einem Torus

Der Algorithmus aus dem letzten Abschnitt kann durch einfache Übertragung des vektororientierten, also eindimensionalen *Ring-Gaxpy* zustande. Der Parallelitätsgrad kann aber noch wesentlich gesteigert werden, indem man bei Matrizen eine wirkliche, zweidimensionale Verallgemeinerung anstrebt. Das zweidimensionale Analogon zum Ring ist der Torus. Hier wird die Multiplikation tatsächlich durch einen doppelten zyklischen Umlauf der Faktoren realisiert. Die Herleitung wird auf einem einfachen 3×3-Torus betrachtet, die Prozessoren besitzen daher zwei Indizes, $P_{ij},\ i, j = 1, 2, 3$. Der Multicomputer hat also die Gestalt

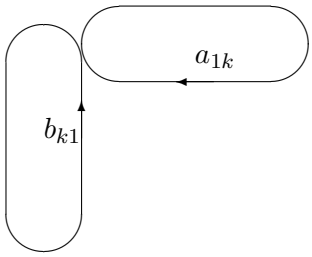
P_{11}	P_{12}	P_{13}
P_{21}	P_{22}	P_{23}
P_{31}	P_{32}	P_{33}

Jeder Prozessor hat vier Verbindungen Nord, Ost, Süd, West zu seinen direkten Nachbarn. Bei am Rand gelegenen Prozessoren laufen die Nachrichten zyklisch weiter, der westliche Nachbar von P_{21} etwa ist P_{23}. Zur Durchführung der 3×3-Multiplikation (mit Blöcken) bekommt Prozessor P_{ij} die Aufgabe, das Ergebniselement d_{ij} zu berechnen. Dies ist, z.B., bei P_{11} die Summe

$$d_{11} = c_{11} + a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}.$$

Man überträgt das Ring-Verfahren jetzt dadurch, dass man die beteiligten Elemente zyklisch verschiebt, und zwar die a_{1k} in horizontaler und die b_{k1} in vertikaler Richtung. Dazu bringt man diese Elemente (allgemein Matrixblöcke) zu Beginn folgendermaßen im Netz unter:

a_{11}	b_{11}	a_{12}	.	a_{13}	.
.	b_{21}	.	.	.	.
.	b_{31}	.	.	.	.



Das rechte Diagramm deutet an, wie die Elemente jeweils in P_{11} aufeinander treffen und dort multipliziert werden. In drei Schritten führt P_{11} jeweils die Operation $c := c + a_{1k}b_{k1}$ aus und empfängt dann $a_{1,k+1}$ aus Osten und $b_{k+1,1}$ aus Süden. Danach liegt das Ergebnis d_{11} in P_{11} vor. Durch geeignete Einpassung der übrigen Matrixelemente in die noch freien Plätze der Tabelle werden analog auch alle anderen Ergebnisse in diesen drei Schritten berechnet. Die Zuordnung ergibt sich dabei zwangsläufig, denn in P_{21} paßt im ersten Schritt zur Aufgabe $d_{21} = c_{21}+a_{2x}b_{21}+..$ nur das Element a_{22}, nach P_{12} paßt nur b_{22}, etc.. Die Matrixelemente werden also schon mit einer zyklischen Verschiebung abgespeichert. Nach jedem Schritt werden dann alle a nach Westen und alle b nach Norden gesendet. Im 3×3-Fall hat man daher den folgenden Ablauf. Zur Verdeutlichung werden die Elemente der dritten Spalte von A und Zeile von B hervorgehoben. Man sieht so direkt, wie die richtigen Elemente in allen Prozessoren einmal aufeinander treffen.

t=1

a_{11}	b_{11}	a_{12}	b_{22}	a_{13}	b_{33}
a_{22}	b_{21}	a_{23}	b_{32}	a_{21}	b_{13}
a_{33}	b_{31}	a_{31}	b_{12}	a_{32}	b_{23}

t=2

a_{12}	b_{21}	a_{13}	b_{32}	a_{11}	b_{13}
a_{23}	b_{31}	a_{21}	b_{12}	a_{22}	b_{23}
a_{31}	b_{11}	a_{32}	b_{22}	a_{33}	b_{33}

t=3

a_{13}	b_{31}	a_{11}	b_{12}	a_{12}	b_{23}
a_{21}	b_{11}	a_{22}	b_{22}	a_{23}	b_{33}
a_{32}	b_{21}	a_{33}	b_{32}	a_{31}	b_{13}

Bei Beginn ist die Matrix A also horizontal nach Westen "verdreht", die Matrix B vertikal nach Norden. A rotiert nach Westen und B nach Norden. Daher steht zu jedem Zeitpunkt in P_{ij} ein Element a_{ix} und b_{yj}. Formal wird der Algorithmus wieder durch folgenden Programmteil beschrieben, der auf jedem Prozessor abläuft. Zur besseren Übersichtlichkeit wird nur der skalare Fall mit $p = n^2$ angegeben, das Verfahren läßt sich analog natürlich mit $r \times r$-Matrixblöcken durchführen. Zur Abkürzung sei außerdem

$$< i >_p := 1 + (i-1) \mathrm{mod} p,$$

die zyklische Indexabbildung $\mathbb{N} \to \{1, .., p\}$ definiert.

Initialisierung, in P_{ij}: $cl = c[i][j]$; $al = a[i][< i + j - 1 >_p]$, $bl = b[< i + j - 1 >_p][j]$.

Torus-MatMul: for $(t = 1 \,..\, n)$ {
$cl = cl + al * bl$;
$sende(al, west)$; $sende(bl, nord)$;
$empfange(al, ost)$; $empfange(bl, sued)$;
}

Bei Durchführung der Operation $cl + al * bl$ zum Zeitpunkt t befinden sich in P_{ij} die Elemente $a[i, < i + j + t - 2 >_p]$ und $b[< i + j + t - 2 >_p, j]$. Nach einem vollen p-Zyklus $t = 1 \,..\, n$ enthält cl daher das Ergebnis d_{ij}. Auch hier ist die Arbeitsweise systolisch, jeder Prozessor tut genau das Gleiche mit maximaler Auslastung.

Da hier wieder alle Prozessoren gleichmäßig ausgelastet sind, auch bei $p \times p$-Blockmatrizen, kann man zur Bestimmung der Effizienz wie früher vorgehen. Mit Blöcken der Größe $r \times r$ und $p = (n/r)^2$ Prozessoren sind in jedem Schritt des Algorithmus pro Prozessor $2r^3$ Operationen durchzuführen und 2 Nachrichten der Größe r^2 zu senden und zu empfangen. Die Zahl der Zyklen (Länge der t-Schleife) ist natürlich $\sqrt{p}$. Daher sind Laufzeit und Effizienz gegeben durch

$$T_p(n) = 2\sqrt{p}(r^3 t_A + s_N + r^2 t_N), \quad E_p(n) = \left(1 + \frac{t_N}{t_A r} + \frac{s_N}{t_A r^3}\right)^{-1}. \tag{3.3.4}$$

Dies ist eine mit (3.3.3) vergleichbare Größe. Es ist aber zu beachten, dass im Torus-Algorithmus bis zu n^2 Prozessoren einsetzbar sind, während bei der Gaxpy-Variante nur n Verwendung finden konnten. Realistisch wird die Gegenüberstellung nur bei Einsatz gleicher Resourcen. Gegeben sei daher ein Rechner mit $p = n = m^2$ Prozessoren, der sowohl als Ring als auch als Torus konfigurierbar ist. Für den Ring-Algorithmus bekommt dann jeder Prozessor jeweils $n \times 1$-Abschnitte der Matrizen, die Formel (3.3.3) ist also mit $r = 1$ anzuwenden. Beim Torus erhält jeder Prozessor $m \times m$-Blöcke mit der gleichen Größe wie beim Ring-Verfahren, in (3.3.4) ist somit $r = m = \sqrt{n}$ zu verwenden. Dies ergibt für die Effizienzen $E := S_p(n)/p$ für Ring (E_R) und Torus (E_T) den Vergleich

$$E_R = \left(1 + \frac{t_N}{2t_A} + \frac{s_N}{2nt_A}\right)^{-1} < E_T = \left(1 + \frac{t_N}{mt_A} + \frac{s_N}{mnt_A}\right)^{-1}. \tag{3.3.5}$$

Denn beim Torus-Verfahren werden die kritischen Größen im Nenner durch m dividiert und nicht nur durch 2. Der wichtigste Grund für den Unterschied sind die unterschiedlichen Relationen von Arithmetik und Kommunikation. Jeder Prozessor empfängt beim Ring-Verfahren pro Schritt einen Block von $n = m^2$ Daten und führt damit $n = m^2$ Operationen aus. Beim Torus-Verfahren werden ebenfalls je $n = m^2$ Daten ausgetauscht, damit dann aber mehr Operationen ausgeführt, nämlich m^3. Das Verhältnins von Arithmetik und Nachrichten ist also beim Torus viel günstiger. Dies ist ein sehr wichtiger Gesichtspunkt, der in vielen Algorithmen zu beobachten ist und bei Standardisierungsbemühungen im Software-Bereich berücksichtigt wurde.

3.4 Standard-Schnittstelle BLAS

Für Rechnernutzer ist es natürlich sehr schwer und lästig, die speziellen Eigenschaften eines verwendeten Rechners bei der Programmierung zu berücksichtigen, da viele unterschiedlichen Rech-

nertypen verfügbar sind und sich durch die fortschreitende Entwicklung auch dauernd ändern. Daher hat man sich schon früh im Bereich der Linearen Algebra bei der Entwicklung der Programmbibliothek LINPACK dazu entschieden, die Schnittstelle für den Benutzer in den Bereich *komplexerer Basisoperationen* zu verlegen. Dazu wurden **B**asic **L**inear **A**lgebra **S**ubprograms, die sogenannten BLAS-Routinen in verschiedenen Stufen definiert. Die niedrigste Stufe BLAS-1 enthält Operationen wie Vektor-Addition, -Linearkombination (saxpy), -Innenprodukt, etc. Bei der Entwicklung neuer Hardware legen Rechnerhersteller Wert darauf, optimale Implementierungen dieser BLAS-Operationen bereitzustellen (evtl. im Maschinencode). Benutzerprogramme können damit die innersten Laufanweisungen der meisten Matrix-Algorithmen durch BLAS-Aufrufe ersetzen. Bei verteiltem Speicher darf man dabei aber die Frage der Datenaufteilung nich außer Acht lassen. Die drei Stufen der BLAS-Konvention unterscheiden sich anhand der Komplexität der Operationen, bei Matrix- bzw. Vektor-Dimension n sind dies in

Stufe 1.	$\mathcal{O}(n)$ Operationen mit	$\mathcal{O}(n)$ Daten,	d. h. Vektor-Vektor-Operationen,
Stufe 2.	$\mathcal{O}(n^2)$ Operationen mit	$\mathcal{O}(n^2)$ Daten,	z. B. Matrix-Vektor-Operationen,
Stufe 3.	$\mathcal{O}(n^3)$ Operationen mit	$\mathcal{O}(n^2)$ Daten,	d. h. Matrix-Matrix-Operationen.

Gründe für diese dreistufige Hierarchie wurden gerade besprochen. Da der Zeitbedarf für Datentransfer gegenüber der Arithmetik nicht vernachlässigt werden kann, ist klar, dass die Effizienz wesentlich von dem Verhältnis von Transfer und Arithmetik anhängt. Man sieht in der Einteilung, dass in den beiden ersten Stufen der Umfang der beiden ungefähr gleich ist und erst bei Stufe 3 ein günstiges Verhältnis von Datenmenge und Operationszahl der Größenordnung $= \mathcal{O}(1/n)$ erreicht wird.

Aus Effizienzgründen ist daher bei Algorithmen i. a. eine Durchführung in Form von Block-Matrix-Matrix-Operationen (z.B. §3.3.2) derjenigen mit Block-Matrix-Vektor-Operationen (z.B. §3.3.1) vorzuziehen.

Mittlerweile gibt es auch Modifikationen von BLAS für spezielle Umgebungen. Entscheidend ist bei allen aber die Sichtweise, dass man beim Algorithmenentwurf möglichst nicht die einzelne arithmetische Operation als Basisoperation ansieht, sondern versucht, die Algorithmen in grobkörnigeren Einheiten zu formulieren, um dort eine Hardware-nahe Parallel-Implementierung zu erreichen. Eine wichtige Funktionalität dieser BLAS-Implementierungen ist z.B. auch eine automatische Blockbildung angepaßt an die vorhandene Cache-Größe. Dies bedeutet, dass etwa bei der Matrixmultiplikation (z.B. auf dem Torus) die Multiplikation der Teilblöcke $A_{ik} \cdot B_{kj} \in \mathbb{R}^{r \times r}$ noch einmal in kleinere Blöcke unterteilt wird so, dass beide Teilblöcke einer Operation und das Produkt vollständig in den Cache passen. Denn bei Stufe-3-Algorithmen werden alle Daten mehrfach verknüpft (s.o.).

4 Gauß-Elimination auf verteilten Systemen

Die Algorithmen zur Matrix-Vektor- und Matrix-Matrix-Multiplikation aus dem letzten Kapitel wiesen fast optimale Leistungsdaten auf. Der Hauptgrund dafür ist die weitgehende Unabhängigkeit der einzelnen Verfahrensteile voneinander sowie die Freiheit in der zeitlichen Anordnung einiger Operationen, z.B. in (3.3.2). Obwohl in §3.1 verschiedene Varianten des Gauß-Algorithmus vorgestellt wurden, liegen hier wesentlich strengere Abhängigkeiten vor, die die Leistung beeinträchtigen werden. Dennoch kann man einige der bei den Multiplikationsverfahren behandelten Maßnahmen auch hier erfolgreich anwenden. Zur Straffung werden nur Multicomputer mit verteiltem Speicher behandelt. Hier bieten sich zur Parallelisierung vor allem die KIJ- und JKI-Varianten an. Zunächst wird wieder der Gauß-Algorithmus ohne Pivotisierung diskutiert.

4.1 Ring-LR-Zerlegung

Das Ring-Gaxpy-Verfahren aus §3.2.2 dient hier als Vorlage für die `gaxpy`-ähnliche Variante, den JKI-Gauß-Algorithmus. Zunächst sei $p = n$ und Prozessor P_j bekommt die Aufgabe zugeteilt, die j-te Spalte der LR-Zerlegung, also den Vektor $(r_{1j}, \ldots, r_{jj}, l_{j+1,j}, \ldots, l_{nj})^T$ zu berechnen. In der JKI-Formulierung hat er dazu Operationen der Form

$$Al[k+1\,..\,n] := Al[k+1\,..\,n] - A[k+1\,..\,n][k] * Al[k],\ k = 1, \ldots, j-1, \tag{4.1.1}$$

durchzuführen mit dem lokalen Vektor $Al = A[1\,..\,n][j]$ und den Ergebnissen $A[1\,..\,n][k]$ der Prozessoren mit kleinerem Index $k < j$. Dies bedeutet aber, dass umgekehrt kein Prozessor mit diesen `saxpy`-Operationen beginnen kann, bevor Prozessor P_1 seine Berechnung von $A[1\,..\,n][1]$ (d.h. die Division durch $A[1][1]$) beendet hat. Geht man von einer Ringstruktur aus und nimmt an, dass P_1 sein Ergebnis nicht gleichzeitig allen anderen in Form einer Radiosendung mitteilen kann, muß jede Matrixspalte im Ring der Reihe nach weitergereicht werden. Zur Beschleunigung des Ablaufs sollte jeder Prozessor dies tun, bevor er seine `saxpy`-Operation beginnt. Nach Empfang aller Spalten mit Indizes $k < j$ kann P_j nun die abschließende Division durchführen und dann sein Ergebnis, zumindest aber $A[j+1\,..\,n][j]$, nach rechts schicken. Das von jedem Prozessor auszuführende Programm lautet mit einem lokalen Speicher $S[1..n]$, der die fertigen Spalten der anderen Prozessoren der Reihe nach aufnimmt, und bei Initialisierung von $Al = A[1\,..\,n][j]$ in P_j also folgendermaßen:

Ring-LR:
$j = wer\_bin\_ich$;
for $(t = 1\,..\,j-1)$ {
  $empfange(S[t+1\,..\,n], links)$;
  $sende(S[t+1\,..\,n], rechts)$; //* S=A[t+1 .. n,t]
  $Al[t+1\,..\,n] = Al[t+1\,..\,n] - S[t+1\,..\,n] * Al[t]$;
}
$Al[j+1\,..\,n] = Al[j+1\,..\,n]/Al[j]$;
$sende(Al[j+1\,..\,n], rechts)$;

Zur Erläuterung des Ablaufs und der auftretenden Wartezeiten wird das Laufzeitdiagramm für $n = p = 8$ betrachtet. Zur Vereinfachung sei der Zeitbedarf für alle Vorgänge gleich. Pro Zeitschritt ist die jeweilige Tätigkeit des einzelnen Prozessors markiert (S: Senden, _E: Empfang mit Warten, $*$: Mult., /: Div., >: Sprung):

```
P0: >/S.................................
P1: >__S*>/S............................
P2: >___S*>_S*>/S.......................
P3: >____S*>_S*>_S*>/S..................
P4: >_____S*>_S*>_S*>_S*>/S.............
P5: >______S*>_S*>_S*>_S*>_S*>/S........
P6: >_______S*>_S*>_S*>_S*>_S*>_S*>/S...
P7: >________E*>_E*>_E*>_E*>_E*>_E*>_E*>/..
```

Offensichtlich hat man eine sehr ungleichmäßige Prozessorauslastung. Denn P_j muß zunächst auf das Eintreffen von $A[1\,..\,n][1]$ warten, und hat nach Berechnung seiner Spalte $A[1\,..\,n][j]$ nichts mehr zu tun. Im Fall $n = pr, 1 < r \in \mathbb{N}$, kann man analog zum Vorgehen bei der Gaxpy-Operation mit einer Dreieckmatrix durch *zyklische Aufteilung* eine bessere Lastverteilung erreichen. Die Einzelaufgaben (Berechnung von $A[\cdot, j]$) wird so auf die Prozessoren verteilt, dass etwa Prozessor P_1 außer der ersten Spalte auch die Spalten $p + 1$, $2p + 1$ usw. berechnet. Das Verfahren kann dabei fast unverändert beibehalten werden, wenn die Ergebnisse im Ring mehrfach umlaufen. Zu beachten ist dabei allerdings, dass beim oben gezeigten Laufzeitdiagramm die Ergebnisse im Ring schneller umlaufen, als Prozessoren für deren Bearbeitung frei werden. Daher muß entweder die Nachrichtenverwaltung in der Lage sein, genügend viele Nachrichten zu speichern (vor allem im letzten Prozessor), oder der Umlauf muß künstlich verzögert werden. Dies hat nicht unbedingt eine Auswirkung auf die Gesamtlaufzeit, da diese allein durch den letzten Prozessor P_p bestimmt wird (vgl. Schaubild). Dieser muß zunächst auf das Eintreffen von $A[1\,..\,n, 1]$ warten und dann jeweils $(p-1) + (2p-1) + \ldots + (rp-1)$ Operationen (4.1.1) sowie r Spalten-Divisionen (mit unterschiedlichen Längen) ausführen. Im einzelnen hat man so in P_p i.w. folgende Zeiten

- Anfang, Warten: $p(s_N + nt_N)$
- Weiterreichen und Saxpy im k-ten Umlauf, $1 \le k \le r$: $kps_N + (t_N + 2t_A)(kpn - \frac{1}{2}k^2p^2)$,
- Abschluß, Division und Senden im k-ten Umlauf: $s_N + (t_N + t_A)(n - kp)$.

Die Summation der beiden letzten Größen über die r Zyklen ergibt eine Gesamtlaufzeit von

$$T_p(n) \cong \frac{1}{p}\Big(\frac{1}{2}n^2 s_N + \underbrace{\frac{1}{3}n^3 t_N} + \underbrace{\frac{2}{3}n^3 t_A}\Big) \tag{4.1.2}$$

Die Effizienz wird, wie früher, i.w. durch das Verhältnis von Kommunikations- (incl. Wartezeiten) und Arithmetik-Aufwand bestimmt. Für dieses Verhältnis ergibt sich hier mit

$$\frac{\frac{1}{3}n^3 t_N}{\frac{2}{3}n^3 t_A} = \frac{t_N}{2t_A}$$

einen von der Problemgröße i.w. unabhängigen Wert. Für $t_N > t_A$ wird die Gesamtlaufzeit somit durch den Nachrichtenaustausch dominiert, das Ring-Verfahren hat also nur BLAS-2-Niveau.

Dafür besitzt das Verfahren aber einen wichtigen Vorteil bei der *Pivotisierung*. Da auf jedem Prozessor jeweils eine ganze Matrixspalte untergebracht ist, sind Zeilenvertauschungen einfach durchzuführen, denn diese können lokal in jedem Prozessor ablaufen. Die entsprechende Information muß nur vom auslösenden Prozessor mit seinem Ergebnisvektor mitgeschickt werden.

4.2 Gitter-LR-Zerlegung

Wenn das Rechnernetz zweidimensional angelegt ist, mit Prozessoren P_{ij} (z.B. $i,j = 1,\ldots,n$) und einer Verbindung als Gitter, kann man analog zum Vorgehen in §3.3.2 dem Prozessor P_{ij} die Aufgabe zuweisen, das Element $l_{ij}, j < i$, bzw. $r_{ij}, j \geq i$, aus der LR-Zerlegung der Matrix A zu berechnen. Für dieses zweidimensionale Vorgehen bietet sich jetzt der normale KIJ-Gauß-Algorithmus (3.1.4) an mit Rang-1-Vektorprodukten. Dieser Vorgang wird aus Sicht eines einzelnen Prozessors betrachtet und zur Verdeutlichung werden an die Matrixelemente Versionsnummern angehängt, wobei $A^{(1)} = A$ ist. Zur Durchführung des Basisschritts

$$a_{ij}^{(k+1)} := a_{ij}^{(k)} - a_{ik}^{(k+1)} a_{kj}^{(k)},\ k < \min\{i,j\}, \tag{4.2.1}$$

benötigt P_{ij} das Element $a_{ik}^{(k+1)}, k < i$, aus Westen und gleichzeitig $a_{kj}^{(k)}, j < k$, aus Norden. Davor hatten die Prozessoren $P_{ik}, k < i$, für die Division $a_{ik}^{(k+1)} := a_{ik}^{(k)} / a_{kk}^{(k)}$ das Hauptdiagonalelement aus Norden benutzt. Umgekehrt sendet daher P_{ij} sein fertiges Element nach Süden, wenn es zu R gehört, also für $j \geq i$ bzw. nach Osten für $j < i$, bei einem L-Element. Die Elemente werden von dazwischenliegenden Prozessoren bis zum Spalten- bzw. Zeilenende weitergereicht.

Der erste Schritt beginnt also mit den Prozessoren P_{1j} der ersten Zeile, die ihre Elemente nach Süden senden. Im nächsten Schritt kann P_{21} seine Division durchführen und nun $a_{21}^{(2)}$ nach Osten senden, etc. Wird jeweils ein Sende-/Empfang-Zyklus mit der arithmetischen Operation als ein Zeitschritt behandelt, ergibt sich für die Zeiten $t_{ij}^{(2)}$, in denen $a_{ij}^{(2)}$ in P_{ij} vorliegt, die Zeitmatrix

$$T^{(2)} = \begin{pmatrix} 0 & 0 & 0 & 0 & . \\ 1 & 2 & 3 & 4 & . \\ 2 & 3 & 4 & 5 & . \\ 3 & 4 & 5 & 6 & . \\ . & . & . & . & . \end{pmatrix}$$

Der Abschluß dieser Berechnung läuft also als Welle nach Südosten. Der zweite Eliminationsschritt ($k = 2$) muß nun allerdings nicht bis zur Ankunft dieser Welle bei P_{nn} warten, sondern P_{22} kann nach Fertigstellung von $a_{22}^{(2)}$ dieses nach Süden senden und damit die zweite Welle anstoßen. Auch für die späteren Schritte erhält man ähnliche Abschlußzeiten,

$$T^{(3)} = \begin{pmatrix} - & - & - & - & . \\ - & 2 & 3 & 4 & . \\ - & 4 & 5 & 6 & . \\ - & 5 & 6 & 7 & . \\ . & . & . & . & . \end{pmatrix}, T^{(4)} = \begin{pmatrix} - & - & - & - & . \\ - & - & - & - & . \\ - & - & 5 & 6 & . \\ - & - & 7 & 8 & . \\ . & . & . & . & . \end{pmatrix}, \ldots \Rightarrow \bar{T} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & . \\ 1 & 2 & 3 & 4 & 5 & . \\ 2 & 4 & 5 & 6 & 7 & . \\ 3 & 5 & 7 & 8 & 9 & . \\ 4 & 6 & 8 & 10 & 11 & . \\ . & . & . & . & . & . \end{pmatrix}.$$

Wenn man aus jeder der Matrizen den letzten Tätigkeitszeitpunkt eines Prozessors nimmt, bekommt man am Ende die Matrix $\bar{T}$, welche den Zeitpunkt der Fertigstellung von $a_{ij}^{(n)}$ angibt. Auch hier sieht man eine mit halber Geschwindigkeit nach Südosten laufende (geknickte) Welle. Der Algorithmus mit lokalen Variablen $al = a_{ij}$, $aw = a_{ik}$, $an = a_{kj}$ lautet so:

Gitter-LR:
$(i, j) = wer\_bin\_ich$;
for $(t = 1 .. min(i, j) - 1)$ {
  $empfange(an, nord)$; $sende(an, sued)$;
  $empfange(aw, west)$; $sende(aw, ost)$;
  $al = al - aw * an$;
}
if $(j < i)$ {
  $empfange(an, nord)$; $sende(an, sued)$;
  $al = al/an$;
  $sende(al, ost)$
} else { $sende(al, sued)$; }

In diesem Programm, das auf jedem einzelnen Prozessoren läuft, wird insbesondere die Division $al = al/an$; erst ausgeführt, wenn die t-Schleife darüber beendet ist, anschließend beendet der Prozessor seine Tätigkeit. Zur Abkürzung wurde bei der Formulierung auf die Unterdrückung von überflüssigen Sendungen (nach Osten für $j = n$, Süden für $i = n$) verzichtet. Den genauen Ablauf für ein 4×4-Gitter erkennt man im ausführlichen Protokoll (Bezeichnungen wie zuvor, die Sendemarkierung kann aber die E überschreiben, da im gleichem Zeittakt abgewickelt):

```
P00: ><S.......................
P01: ><S.......................
P02: ><S.......................
P03: ><S.......................
P10: ><_S/S....................
P11: >__S__S*><S...............
P12: >__S___S*><S..............
P13: >__S____E*><S.............
P20: ><__S/S...................
P21: >___S__S*><S/S............
P22: >___S___S*>_S_S*><S.......
P23: >___S____E*>_S_E*><S......
P30: ><___E/S..................
P31: >____E__S*><E/S...........
P32: >____E___S*>_E_S*><E/S....
P33: >____E____E*>_E_E*>_E_E*><..
```

Anhand des Protokolls bzw. bei Durchsicht des Programms macht man folgende Beobachtung: Prozessor P_{ij} fügt jeweils nur ein einziges Mal ein neues Element in den durchlaufenden Nachrichtenstrom ein, und zwar bei Abschluß seiner Berechnungen, am Ende der Abfolge $a_{1j}^{(1)}, .., a_{i-1,j}^{(i-1)}$

für $j \geq i$ bzw. $a_{i1}^{(2)}, .., a_{i,j-1}^{(j)}$ für $j < i$. Daher wird die Anordnung dieser Elemente nicht verändert. Somit sind die k-ten Elemente, die in P_{ij} aus Norden bzw. Westen eintreffen gerade die passenden Faktoren a_{kj} und a_{ik} in Formel (4.2.1). Diese Beobachtung zeigt außerdem für den Algorithmus mit $\frac{2}{3}n^3$ sequentiellen Operationen eine lineare Gesamtlaufzeit von ca. $2n$ Zyklen (jeweils Empfang+ Senden+ Arithmetik). Die Wartezeiten tragen dazu ungefähr die Hälfte bei und die Prozessoren der ersten Zeile sind kaum aktiv, die Lastverteilung ist also nicht optimal. Der Speed-up ist mit $S_p \doteq \frac{2}{3}n^3/(2n) = \frac{1}{3}n^2$ ganz beachtlich. Da er allerdings mit $p = n^2$ Prozessoren erreicht wurde, liegt die Effizienz nur bei $E_{n^2}(n) \doteq \frac{1}{3}$. Dies ist ein allerdings ein prinzipielles Problem bei direkten Methoden zur Lösung von Linearen Gleichungssystemen und Anlaß für die Diskussion von Iterationsverfahren in §5.

Wie bei der Matrix-Multiplikation ist bei diesem Verfahren v.a. die Blockversion mit Blockaufteilungen von $A = (A_{ij})_{i,j=1}^{\sqrt{p}}$, für $p = (n/r)^2$ interessant. Statt der einfachen Operation (4.2.1) hat der Prozessor P_{ij} jetzt die Operationen

$$\text{löse} \quad A_{ik}^{(k+1)} A_{kk}^{(k)} = A_{ik}^{(k)},\ i > j = k, \quad A_{ij}^{(k+1)} := A_{ij}^{(k)} - A_{ik}^{(k+1)} A_{kj}^{(k)},\ i, j > k, \tag{4.2.2}$$

mit Blöcken der Größe $r \times r$ durchzuführen. Dazu kann Prozessor P_{kk} vorab eine LR-Zerlegung von $A_{kk}^{(k)}$ oder deren Inverse berechnen oder jeder Prozessor P_{ik}, $i > k$ löst sein Problem unabhängig. In dieser Version ist eine Laufzeitdiskussion schwierig, da abhängig vom Verhältnis von Kommunikation und Arithmetik unterschiedliche Synchronisationsbedingungen herrschen. Sicher liegt die Laufzeit aber in der Größenordnung $\sqrt{p}\, r^3 = n^3/p$ und die Beschleunigung erreicht nur einen Bruchteil $(\frac{1}{3}..\frac{1}{2})$ von p, dies aber ziemlich unabhängig von der Problemgröße. Da jeder Prozessor (z.B. $P_{p-1,j}, P_{i,p-1}$) größenordnungsmäßig $\sqrt{p}r^2$ Daten empfängt und sendet und damit $\sqrt{p}\, r^3$ Operationen ausführt, hat das Verfahren aber jetzt tatsächlich BLAS-3-Niveau.

Fazit: Das Verfahren ist gut skalierbar für Prozessoranzahlen zwischen 1 und n^2, bei genügend vielen Prozessoren, also $p = n^2$, hat es eine lineare Laufzeit in n.

Im Vergleich zum Ring-Algorithmus kann man beim Gitterverfahren eine Pivotisierung nicht so leicht vornehmen, da Zeilenvertauschungen jetzt mit erheblichem Nachrichtenverkehr verbunden sein können. Allerdings ist diese Frage bei realistischen Größenordnungen nicht so dringend, da der Algorithmus dann in der Blockversion (4.2.2) durchgeführt wird. Voraussetzung für Durchführbarkeit ist dann natürlich nur noch die Regularität (beschränkte Invertierbarkeit) der gesamten Diagonalblöcke $A_{kk}^{(k)}$. Innerhalb dieser Blöcke kann natürlich leicht (und sollte auch) pivotisiert werden.

4.3 Tridiagonalsysteme

Eine besonders einfache Form dünnbesetzter Matrizen ist die Bandgestalt, was bedeutet, dass nur bis zu einem bestimmten Abstand von der Hauptdiagonale nichttriviale Elemente auftreten. Man sagt, die Matrix A hat Bandbreite $2m+1$, wenn $a_{ij} = 0$ ist für $|i-j| > m$. Diese Gestalt ist beim Gauß-Algorithmus vorteilhaft, da sie bei fehlender Pivotisierung erhalten bleibt (vgl. Numerik 1).

Daher erfordert die Auflösung von Gleichungssystemen nur einen linearen Aufwand $\mathcal{O}(m^2 n)$ in der Dimension n. Obwohl der normale Gauß-Algorithmus hier nicht parallelisierbar ist, hat eine andere Form der Elimination eine kurze prallele Laufzeit. Diese sog. *zyklische Reduktion* orientiert sich am Spezialfall der Tridiagonal-Matrizen mit $m = 1$.

$$\begin{pmatrix} a_1 & b_1 & & & \\ c_2 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & c_n & a_n \end{pmatrix} \qquad \begin{pmatrix} A_1 & B_1 & & & \\ C_2 & A_2 & B_2 & & \\ & \ddots & \ddots & \ddots & \\ & & C_{k-1} & A_{k-1} & B_{k-1} \\ & & & C_k & A_k \end{pmatrix}$$

Tridiagonalmatrix Block-Tridiagonalmatrix

Da jede $2m+1$-Bandmatrix aber als Block-Tridiagonalmatrix mit $m \times m$-Blöcken darstellbar ist, ist der Algorithmus allgemeiner einsetzbar. Der Algorithmus wird nur für Tridiagonalsysteme behandelt, der Übergang zur Blockversion ist analog zu (4.2.2) möglich. Am Besten funktioniert das Verfahren für Dimension $n = 2^k - 1$ und wird daher am Beispiel $n = 7$ vorgeführt. Beim Gleichungssystem

$$\begin{array}{rrrrrrrl} a_1x_1 & +b_1x_2 & & & & & & = r_1 \\ c_2x_1 & +a_2x_2 & +b_2x_3 & & & & & = r_2 \\ & c_3x_2 & +a_3x_3 & +b_3x_4 & & & & = r_3 \\ & & c_4x_3 & +a_4x_4 & +b_4x_5 & & & = r_4 \\ & & & c_5x_4 & +a_5x_5 & +b_5x_6 & & = r_5 \\ & & & & c_6x_5 & +a_6x_6 & +b_6x_7 & = r_6 \\ & & & & & c_7x_6 & +a_7x_7 & = r_7 \end{array} \tag{4.3.1}$$

ist die Lösung x gesucht. Im ersten Schritt werden hier die Gleichungen mit ungeraden Nummern dazu verwendet, durch Elimination **nach unten und oben** die ungeraden Variablen x_1, x_3, x_5, x_7 in den beiden benachbarten Gleichungen mit geraden Nummern zu eliminieren. Man löst also diese Gleichungen nach den x_{2j-1} auf,

$$x_{2j-1} = \frac{1}{a_{2j-1}}\big(r_{2j-1} - c_{2j-1}x_{2j-2} - b_{2j-1}x_{2j}\big),\ 1 \le j \le (n+1)/2, \tag{4.3.2}$$

(mit $c_0 = b_n := 0$) und setzt sie in die anderen Gleichungen ein. Daher heißt das Verfahren im Englischen auch *odd-even-reduction*.

Dazu subtrahiert man also das c_2/a_1-fache der ersten Gleichung und das b_2/a_3-fache der dritten von der zweiten Gleichung und eliminiert dort dadurch die Variablen x_1 und x_3, bekommt allerdings eine Abhängigkeit von x_4 dazu. Allgemein subtrahiert man für jedes j mit $1 \le j \le n/2$ das c_{2j}/a_{2j-1}-fache von Gleichung $2j-1$ und das b_{2j}/a_{2j+1}-fache von Gleichung $2j+1$ von der Gleichung Nummer $2j$. Dort verschwinden daher die Unbekannten x_{2j-1} und x_{2j+1}, allerdings treten dafür x_{2j-2} und x_{2j+2} auf. Den Effekt erkennt man, wenn man das Ergebnis für die

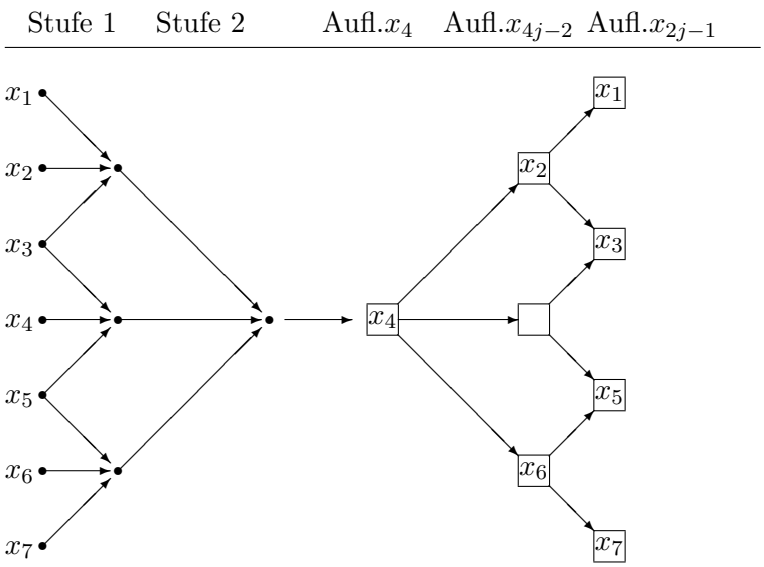
geraden Gleichungen im Beispiel (4.3.1) betrachtet:

$$\begin{array}{rrrrl}
\big(a_2 - \frac{c_2 b_1}{a_1} - \frac{c_3 b_2}{a_3}\big) x_2 & & -\frac{b_2 b_3}{a_3} x_4 & & = r_2 - \frac{c_2}{a_1} r_1 - \frac{b_2}{a_3} r_3 \\
-\frac{c_4 c_3}{a_3} x_2 & + \big(a_4 - \frac{c_4 b_3}{a_3} - \frac{c_5 b_4}{a_5}\big) x_4 & & - \frac{b_4 b_5}{a_5} x_6 & = r_4 - \frac{c_4}{a_3} r_3 - \frac{b_4}{a_5} r_5 \\
& & -\frac{c_6 c_5}{a_5} x_4 & + \big(a_6 - \frac{c_6 b_5}{a_5} - \frac{c_7 b_6}{a_7}\big) x_6 & = r_6 - \frac{c_6}{a_5} r_5 - \frac{b_6}{a_7} r_7
\end{array}$$

Dies ist ein eigenes tridiagonales Gleichungssystem (mit neuen Koeffizienten a'_2, b'_2, etc.) für die 3 Variablen x_2, x_4, x_6, welches durch eine erneute Reduktion auf eine einzige Gleichung

$$a''_4 x_4 = r''_4$$

zusammenschrumpft. Wenn daraus x_4 bestimmt wurde, kann man aus dem reduzierten System x_2 und x_6 berechnen und am Ende aus dem Originalsystem (4.3.2) die restlichen x_1, x_3, x_5, x_7. Der entscheidende Vorteil gegenüber dem Gauß-Algorithmus ist aber, dass die einzelnen Eliminationsschritte einer Stufe unabhängig voneinander sind und daher **parallel** ausgeführt werden können. Das Gleiche gilt für die Rückeinsetzung. Der Ablauf wird im folgenden Diagramm skizziert. Umformungen (Pfeile), die untereinander stehen, können parallel bearbeitet werden:



Für allgemeines $n = 2^p - 1$ läßt sich ergänzen:

- Wenn man Matrixelemente außerhalb des Indexbereichs $1 \,..\, n$ wieder null setzt, benötigt man keine Ausnahme für die erste und letzte Zeile. Dann lauten die Eliminationsschritte in der Stufe $k \geq 0$ für $j = 1, \ldots, (n+1)/2^{k+1}$:

$$\begin{aligned}
a_{j2^{k+1}} &:= a_{j2^{k+1}} - \frac{b_{(2j-1)2^k}}{a_{(2j-1)2^k}} c_{j2^{k+1}} - \frac{c_{(2j+1)2^k}}{a_{(2j+1)2^k}} b_{j2^{k+1}}, \\
b_{j2^{k+1}} &:= - \frac{b_{(2j+1)2^k}}{a_{(2j+1)2^k}} b_{j2^{k+1}} \\
c_{j2^{k+1}} &:= - \frac{c_{(2j-1)2^k}}{a_{(2j-1)2^k}} c_{j2^{k+1}}, \\
r_{j2^{k+1}} &:= r_{j2^{k+1}} - \frac{r_{(2j-1)2^k}}{a_{(2j-1)2^k}} c_{j2^{k+1}} - \frac{r_{(2j+1)2^k}}{a_{(2j+1)2^k}} b_{j2^{k+1}}.
\end{aligned}$$

- die Laufzeit ist offensichtlich $\mathcal{O}(\log_2 n)$ etwa mit $p = \frac{n}{2}$ Prozessoren. Ähnlich wie bei der rekursiven Verdopplung arbeiten aber nur im ersten Schritt alle Prozessoren.

- Bei verteilter Implementierung auf Multicomputern, wo jeweils (Gruppen von) Koeffizienten und Variablen auf verschiedenen Prozessoren liegen, ist zu beachten, dass in den verschiedenen Eliminationsstufen immer größere Indexdifferenzen 2^k auftreten. Für die erforderliche Kommunikation wäre daher eine Hypercube-Struktur günstig.

- Wie auch sonst bei der Elimination ist das Verfahren nur bei nichttrivialen Pivotelementen durchführbar. Hinreichende Kriterien dafür sind wie bei der üblichen Gauß-Elimination wieder die Definitheit oder Diagonaldominanz der Matrix. Die exakten Kriterien für allgemeine Matrizen (nichttriviale Hauptminoren) lauten aber anders als im Standardverfahren.

5 Parallele Iterationsverfahren

Bisher befaßten sich die besprochenen Verfahren vor allem mit Operationen bei vollbesetzten Matrizen. Aber gerade bei großen Problemen mit massivem Rechenbedarf sind die Matrizen oft dünnbesetzt. In einem gewissen Umfang kann man direkte Lösungsverfahren zwar an bestimmte dünne Besetzungen anpassen wie im Fall der Bandsysteme, vgl. §4.3. Da aber direkte Lösungsverfahren meist eine ungleichmäßige Lastverteilung besitzen, nutzt man eine spärliche Besetzung von linearen Gleichungssystemen am Einfachsten durch Verwendung von Iterationsverfahren aus. Hier gibt es verschiedene Ansätze bei der Parallelisierung. Zunächst achtet man natürlich auf die Parallelität im Iterationsverfahren selbst. zusätzlich kann man aber auch äquivalente Umformulierungen des LGS betrachten so, dass Verfahren mit begrenzter Parallelität wesentlich schneller konvergieren (*Präkonditionierung*). Das macht natürlich nur Sinn, wenn diese Umformung gut parallelisierbar ist. Mit Iterationsverfahren bekommt man aber auch ganz neue Optionen für die Verfahren. Während bei den bisherigen Algorithmen eine ganz präzise Ablaufplanung und Synchronisation bei der parallelen Durchführung notwendig war, können Iterationen sogar (fast) ohne globale Koordination, d.h. asynchron oder chaotisch, funktionieren. Dies ist interessant, wenn man im Netz verteilte Rechner unterschiedlicher Leistung zur Erledigung von Teilaufgaben heranziehen will (*Grid Computing*, *Cloud Computing*), oder wenn der Rechenbedarf der verschiedenen *Tasks* unterschiedlich groß ist. Einige der Maßnahmen setzen aber speziellere Matrixeigenschaften voraus. Daher wird zunächst ein einfaches wichtiges Beispiel eingeführt, das viele solcher Eigenschaften besitzt.

5.1 Ein Referenzbeispiel: Poisson-Gleichung

Die einfachste Version eines elliptischen Randwertproblems im $\mathbb{R}^d$ ist die Poisson-Gleichung, welche auf einem Gebiet $\Omega \subseteq \mathbb{R}^d$ definiert ist. Dabei wird eine Funktion $u : \Omega \to \mathbb{R}$ gesucht, welche am Gebietsrand $\partial\Omega = \bar{\Omega} - \Omega$ feste Randwerte (z.B. null) annimmt und deren mittlere Krümmung in Ω mit vorgegebenen Werten $g : \Omega \to \mathbb{R}$ übereinstimmt. Dieses Randwertproblem lautet also

$$-\Delta u(x_1, \ldots, x_d) = g(x_1, \ldots, x_d),\ (x_1, \ldots, x_d) \in \Omega, \tag{5.1.1}$$

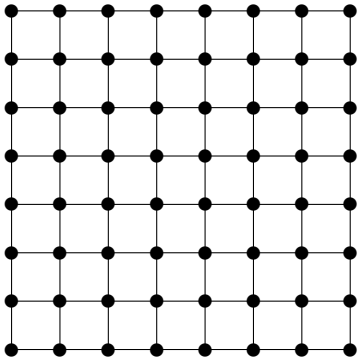
$$u(x_1, \ldots, x_d) = 0, \qquad (x_1, \ldots, x_d) \in \partial\Omega. \tag{5.1.2}$$

Dabei ist $\Delta = \frac{\partial^2}{\partial x_1^2} + \ldots + \frac{\partial^2}{\partial x_d^2}$ der Laplace-Operator. Das Problem beschreibt z.B. die Temperaturverteilung $u(x)$ in einem homogenen Körper, der am Rand gekühlt wird mit Wärmequellen g im Inneren oder kleine Auslenkungen $u(x)$ einer dünnen Membran (Seifenhaut), die am Rand eingespannt ist unter der Flächenlast g. Bei allgemeinere Varianten solcher *elliptischer* Randwertprobleme treten Koeffizientenfunktionen vor den Ableitungen und Ableitungen niederer Ordnung auf. Auch nichtlineare Abhängigkeiten von u sind möglich.

Für eine numerische Approximation von glatten Lösungen $u(x)$ sucht man nur Näherungen an einer endlichen Anzahl von Punkten in Ω. Am Einfachsten verwendet man dabei ein

regelmäßiges Gitter von Punkten, dazu wird beispielhaft als Grundgebiet ein Quadrat oder Würfel $\Omega = [0,1]^d$ betrachtet. In jeder Ortsrichtung unterteilt man das Intervall $[0,1]$ in $m+1$ Teilintervalle mit Trennstellen $x_i^{(j)} = jh$, $j = 0, \ldots, m+1$ der Schrittweite $h = 1/(m+1)$. Bei Dimension $d = 2$ bekommt man so tatsächlich ein Gitter $(x_1^{(i)}, x_2^{(j)})$, $0 \le i, j \le m+1$. In der Differentialgleichung (5.1.1) approximiert man nun die zweiten Ableitungen durch zweite dividierte Differenzen (vg. Numerik 1). Dazu gilt für eine viermal stetig diffbare Funktion u im Punkt $x = (x_1, x_2)$ mit dem Einheitsvektor e_i:

8^2 Gitterpunkte 2D

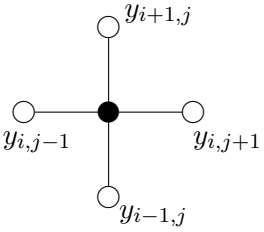


$$\frac{\partial^2 u}{\partial x_i^2}(x) = \frac{u(x - he_i) - 2u(x) + u(x + he_i)}{h^2} + \mathcal{O}(h^2). \qquad (5.1.3)$$

Die Fehleraussage läßt sich auch direkt durch Taylorentwicklung nachweisen. Betrachtet man diese Approximation in einem der inneren Gitterpunkte $x = (x_1^{(j_1)}, x_2^{(j_2)})$ mit $1 \le j_1, j_2 \le m$, dann liegen die anderen Funktionswerte aus der Formel (5.1.3) ebenfalls auf dem Gitter. Daher ersetzt man zur Approximation in der Gleichung (5.1.1) den Laplace-Operator durch die Differenzenquotienten in den Koordinatenrichtungen und vernachlässigt den Fehlerterm. Für die unbekannten Näherungen $y_{ij} \cong u(x_1^{(j)}, x_2^{(i)})$ erhält man im 2D-Fall so das Lineare Gleichungssystem

$$\frac{1}{h^2}\big(- y_{i-1,j} - y_{i,j-1} + 4y_{ij} - y_{i,j+1} - y_{i+1,j}\big) = g_{ij} = g(x_1^{(j)}, x_2^{(i)}), \quad 1 \le i, j \le m. \qquad (5.1.4)$$

Die Differentialgleichung wird also nur in den inneren Gitterpunkten verwendet. Die Graphik rechts zeigt, wie in einer dieser Gleichungen der schwarz gefüllte Wert y_{ij} mit den direkten Nachbarpunkten eines *Fünf-Punkte-Sterns* verknüpft wird. Die in (5.1.4) auftretenenden Nachbarwerte y_{kl} mit $k, l = 0$ oder $k, l = m+1$ liegen auf dem Rand und haben aufgrund der Randbedingung (5.1.2) den vorgegebenen Wert null.

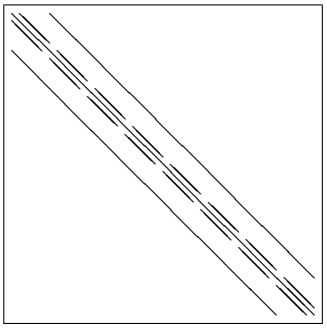


Somit stellt (5.1.4) tatsächlich ein quadratisches Gleichungssystem für die $n = m^2$ unbekannten Gitterwerte y_{ij}, $1 \le i, j \le m$, dar. Da aber in jeder Gleichung nur maximal 5 nichttriviale Koeffizienten auftreten, ist das System offensichtlich sehr *dünn besetzt*. Zur Lösung des Gleichungssystems müssen diese doppelt indizierten Unbekannten y_{ij} natürlich noch in eine lineare Reihenfolge/Nummerierung $y[1\,..\,n]$ gebracht werden. Doch dies ist eine Maßnahme, die erheblichen Einfluß auf die Struktur der zugehörigen Matrix und den Rechenaufwand von Lösungsverfahren hat und daher genauer betrachtet werden muß.

- Zeilenweise Nummerierung: Bei einer Nummerierung in der Reihenfolge $y_{11}, y_{21}, \ldots, y_{m1}$, $y_{12}, \ldots, y_{m2}, \ldots, y_{mm}$ haben vertikal benachbarte Gitterpunkte aufeinander folgende Indizes, bei horizontal benachbarten ist die Indexdifferenz dagegen genau m. Daher ist die Matrix eine $2m+1$-Bandmatrix und Block-Tridiagonalmatrix mit $m \times m$ Blöcken der

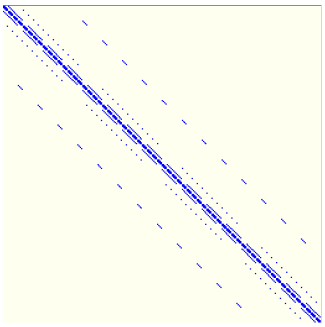
Größe $m \times m$ folgender Form

$$\frac{1}{h^2}\begin{pmatrix} T & -I & & & \\ -I & T & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & T & -I \\ & & & -I & T \end{pmatrix} = \quad T = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{pmatrix}.$$



Die Hauptdiagonalblöcke T sind selber wieder tridiagonal, mit -1 in der Nebendiagonale und der Hauptdiagonale 4. Bei allgemeineren elliptischen RWPen 2. Ordnung ändern sich zwar die Werte der nichttrivialen Matrixeinträge, die Struktur bleibt aber i.w. erhalten. Für parallele Blockverteilungen hat man hier also Hauptdiagonalblöcke der Größe m, die Anzahl der Nachbarn eines Blocks (Nachbarzeilen) ist $\mathcal{O}(m)$ ≅Blockgöße.

- Kachelung: Eine Alternative ist die Zerlegung des Gebiets in $k \times k$ Kacheln. Die Zahl der Unbekannten in einer Kachel (Block) ist dann $(m/k)^2$. Die Randlänge einer jeden Kachel ist nur $\mathcal{O}(m/k)$ und dies ist auch die Anzahl direkter Nachbarn einer Kachel. Damit ist $(Anz.Nachbarn) \cong \sqrt{Blockgröße}$ und bei verteilten Algorithmen ist der Kommunikationsaufwand geringer als vorher. Das Bild zeigt die Matrix-Struktur bei einer 4×4-Kachelung für $m = 16$ (256 Gitterpunkte).



Die Matrix hat außerdem einige wichtige Eigenschaften, die unabhängig von der Nummerierung sind. Sie ist nämlich symmetrisch und positiv definit, schwach diagonaldominant, und hat eine spezielle Vorzeichenverteilung, nämlich eine relativ große positive Hauptdiagonale und nichtpositive Nebendiagonalen. Die Konsequenzen daraus werden später besprochen. Man kann für diesen speziellen Fall sogar die Eigenwerte der Matrix explizit ausrechnen. Für den kleinsten Eigenwert gibt es eine von m unabhängige positive untere Schranke, während der größte Eigenwert offensichtlich die Größenordnung $O(h^{-2}) = O(m^2)$ besitzt. Dies ist daher auch die Größenordnung der Konditionszahl $\kappa \sim m^2 = n$. Für dreidimensionale Probleme mit $n = m^3$ gilt dagegen $\kappa \sim m^2 = n^{2/3}$. Diese Größe ist für die Konvergenz vieler Iterationsverfahren wichtig.

5.2 Parallelität bei Standardverfahren

Eine Klasse von Iterationsverfahren für reguläre lineare Gleichungsysteme $Ax = b$ bekommt man aus einer Zerlegung $A = M - N$ mit einer regulären Matrix M, indem man das System umschreibt in die Form

$$Mx = Nx + b.$$

Wenn Systeme der Form $Mx = y$ einfach auflösbar sind, kann man auf der rechten Seite eine Näherung $x^{(k)}$ der Lösung einsetzen und die linke Seite nach einer neuen auflösen, $Mx^{(k+1)} = Nx^{(k)} + b$. Dies kann man wiederholen für $k = 0, 1, \ldots$ und bekommt so eine Iterationsfolge

$$x^{(k+1)} := \underbrace{M^{-1}N}_{=:B} x^{(k)} + M^{-1}b = Bx^{(k)} + r, \quad k = 0, 1, 2, \ldots$$

Voraussetzung für die Konvergenz dieser Folge ist, dass für den Spektralradius gilt

$$\varrho(B) = \varrho(M^{-1}N) < 1.$$

Die einfachsten dieser Verfahren sind das Gesamtschritt- (GSV) und Einzelschritt-Verfahren (ESV, Gauss-Seidel-Iteration), welche nichttriviale Hauptdiagonalelemente voraussetzen. Beim *GSV* (Jacobi-Iteration) wählt man $M = D := \mathrm{diag}(a_{ii})$ und erhält den Iterationsschritt

$$x_i^{(k+1)} := x_i^{(k)} + \frac{1}{a_{ii}}\Big(b_i - \sum_{j=1}^{n} a_{ij}x_j^{(k)}\Big), \; i \in \{1, \ldots, n\}. \tag{5.2.1}$$

An den Schrittindizes kann man ablesen, dass hier alle n Komponenten unabhängig voneinander, also **parallel** berechnet werden können. Dies wurde auch durch die Vorschrift $i \in \{1, \ldots, n\}$ hervorgehoben. Die Iterationsmatrix ist $B_G = D^{-1}(D - A)$. Bei Standardrechnern wird üblicherweise das Einzelschrittverfahren bevorzugt, da es meist schneller konvergiert, beim Referenzproblem (5.1.4) genau doppelt so schnell wie das GSV. Diese Variante unterscheidet sich dadurch, vom GSV, dass man dessen Teilergebnisse sofort in der nächsten Komponente berücksichtigt,

$$x_i^{(k+1)} := \frac{1}{a_{ii}}\Big(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\Big), \; i = 1, \ldots, n. \tag{5.2.2}$$

Mit dem strikt unteren Anteil L von A, ($l_{ij} = a_{ij}, j < i$, sonst $l_{ij} = 0$) ist die Iterationsmatrix hier $B_E = (D + L)^{-1}(D + L - A)$ und ihr Spektralradius beim Referenzproblem bei Konvergenz tatsächlich $\varrho(B_E) = \varrho(B_G)^2 < \varrho(B_G)$. Hinreichend für Konvergenz ist die Diagonaldominanz von A. Da in beiden Verfahren natürlich nur die nichttrivialen Matrixelemente zu berücksichtigen sind, verringert eine dünnbesetzte Gestalt den Aufwand. Durch Einführung eines zusätzlichen Beschleunigungsparameters läßt sich die Konvergenz sogar noch erheblich beschleunigen ($\rightarrow$ SOR-Iteration, Numerik 1). Leider verhindert die direkte Wiederverwendung von $x_{i-1}^{(k+1)}$ bei $x_i^{(k+1)}$ i.A. jede Parallelisierung.

Beim Referenzproblem (5.1.4) gibt es aber einen Ausweg. Hintergrund ist die Geometrie der Kopplungen im Differenzenstern, da die Variable y_{ij} nur mit den direkten Nachbarn vertikal und horizontal gekoppelt ist. Wenn man die Variablen in einer Schachbrett-Nummerierung anordnet, wie rechts angedeutet, sind alle Nachbarn der roten Felder weiß bzw. umgekehrt. Daher bilden rote und weiße Variable zwei ungefähr gleichgroße Blöcke, zwischen denen es keine Kopplungen in der Matrix gibt. In dieser Nummerierung hat die Matrix $D^{-1}A$ daher folgende Gestalt

22	47	23	48	24	49	25
43	19	44	20	45	21	46
15	40	16	41	17	42	18
36	12	37	13	38	14	39
8	33	9	34	10	35	11
29	5	30	6	31	7	32
1	26	2	27	3	28	4

$$D^{-1}A = \begin{pmatrix} I & -N \\ -N^{\mathsf{T}} & I \end{pmatrix}.$$

Nach Aufteilung der Gesamtvektoren $x^{(k)}$ in x_r und x_w hat das ESV (5.2.2) die Form

$$\left.\begin{array}{rcl} x_r^{(k+1)} &=& b_r + N x_w^{(k)}, \\ x_w^{(k+1)} &=& b_w + N^{\mathsf{T}} x_r^{(k+1)}, \end{array}\right\} \; k = 0, 1, \ldots$$

Hier sind die beiden Teil-Multiplikationen mit N bzw. N^{T} wieder parallel durchführbar. Als eigenständige Iterationsverfahren sind GSV und ESV heutzutage aber nicht mehr brauchbar, da ihre Konvergenz zu langsam ist. Im Referenzproblem (5.1.4) etwa ist die Kondition $\kappa \sim m^2$ und die Konvergenzfaktoren für beide daher $\varrho(B) \cong 1 - C/\kappa$ sehr nahe an eins. Die Verfahren werden aber noch eingesetzt als Bausteine komplexerer Verfahren (s.u.) und da sie einen glättenden Einfluß auf die Gestalt der Fehler haben. Im Zusammenspiel mit anderen Verfahren kann dies vorteilhaft sein.

Ein sehr effizientes Iterationsverfahren ist das Konjugierte Gradienten-Verfahren (CG) von Hestenes und Stiefel, welches allerdings nur bei symmetrisch, positiv-definiten Matrizen A einsetzbar ist. Nicht alle Details dieses Verfahrens werden im Einzelnen hergeleitet. Wir konzentrieren uns auf die Eigenschaften, die für den Einsatz auf Parallelrechnern wichtig ist. Ein wichtiger Hintergrund des Verfahrens ist die Minimierung des Funktionals

$$x \mapsto \phi(x) := \frac{1}{2} x^{\mathsf{T}} A x - b^{\mathsf{T}} x, \tag{5.2.3}$$

dessen Gradient gerade $grad\phi(x) = Ax - b$ der (negative) Defekt im Gleichungssystem ist und deren Hesse-Matrix $\partial^2\phi/\partial x^2 = A$ positiv definit. Daher ist $z := A^{-1}b$ global eindeutiges Minimum von ϕ, der Minimalwert ist $\phi_{\min} = -\frac{1}{2} b^{\mathsf{T}} A^{-1} b$. Ausgehend von einem Startvektor $x^{(0)}$ (z.B. $= 0$) konstruiert das Verfahren *Abstiegsrichtungen* $p^{(k)}$ und minimiert dann das Funktional ϕ auf dem Strahl $x^{(k-1)} + tp^{(k)}$ exakt (*Liniensuche*). Man prüft sofort, dass dieses Minimum wegen $\phi(x + tp) = \phi(x) - tp^{\mathsf{T}}(b - Ax) + \frac{1}{2} t^2 p^{\mathsf{T}} A p$ in

$$\hat{t} = \frac{p^{\mathsf{T}}(b - Ax)}{p^{\mathsf{T}} A p} = \frac{p^{\mathsf{T}} d}{p^{\mathsf{T}} A p} \text{ mit } \quad \phi(x + \hat{t}p) = \phi(x) - \frac{(p^{\mathsf{T}} d)^2}{2 p^{\mathsf{T}} A p} \tag{5.2.4}$$

angenommen wird. Dabei ist $d = b - Ax$ der aktuelle Defekt (=Gradient). Die Effizienz dieses Abstiegsverfahrens wird dadurch dramatisch gesteigert, dass man die verschiedenen Suchrichtungen paarweise orthogonal wählt im A-Innenprodukt gemäß ${p^{(k)}}^{\mathsf{T}} A p^{(j)} = \delta_{kj}$. Dann wird nämlich durch die eindimensionale Liniensuche mit (5.2.4) sogar das globale Minimum

$$x^{(k)} := x^{(k-1)} + t_k p^{(k)} = \operatorname{argmin}\{\phi(x) : \; x \in span(p^{(1)}, \ldots, p^{(k)})\} \tag{5.2.5}$$

im Unterraum aller bisherigen Suchrichtungen bestimmt, denn wegen der A-Orthogonalität der Suchrichtungen sind die Minimierungsschritte entkoppelt. Daher liefert dieses Verfahren nach n Schritten (im Prinzip) sogar die exakte Lösung z! Diese Eigenschaft war das ursprüngliche Ziel

der Erfinder, welches das Verfahren aber i.d.R. nicht erreichen kann wegen Rundungsfehlern. Für moderne Anwendungen ist dagegen das allgemeine Konvergenzverhalten viel interessanter. Die Suchrichtungen werden mit Hilfe des aktuellen Defekts $d^{(k)} = b - Ax^{(k)}$ bestimmt, für den sich die Beziehung $d^{(k)} := d^{(k-1)} - t_k A p^{(k)}$ direkt ergibt. Es läßt sich zeigen, dass man durch Wahl der Suchrichtungen in der Form $p^{(1)} := d^{(0)}$ und $p^{(k+1)} := d^{(k)} + \beta_k p^{(k)}$ die vollständige A-orthogonalität der Richtungen p sicherstellen kann. Da jeder Defektschritt eine Multiplikation mit der Matrix A bedeutet, ist der in (5.2.5) auftretende Unterraum der Suchrichtungen identisch mit den sogenannten *Krylov-Raum*

$$\mathcal{K}_k(A, d) := span(d, Ad, \ldots, A^{k-1}d) = span(p^{(1)}, \ldots, p^{(k)}).$$

Dies ist der Unterraum aller Vektoren, die aus $d = d^{(0)}$ durch Multiplikation mit einem *Polynom* der Matrix A hervorgehen, $\mathcal{K}_k(A, d) = \{q(A)d : \, q \in \Pi_{k-1}\}$, wobei Π_m die Menge aller Polynom mit Maximalgrad m bezeichnet. Das CG-Verfahren sieht recht einfach aus und lautet:

$$\begin{aligned}
&d^{(0)} := b - Ax^{(0)};\ p^{(1)} := d^{(0)};\ m := 0;\\
&\text{while } (d^{(k)} \neq 0)\ \{\\
&\quad k = k + 1;\\
&\quad t_k = \|d^{(k-1)}\|_2^2 / {p^{(k)}}^{\mathsf{T}} A p^{(k)};\\
&\quad x^{(k)} = x^{(k-1)} + t_k p^{(k)};\\
&\quad d^{(k)} = d^{(k-1)} - t_k A p^{(k)};\\
&\quad \beta_k = \|d^{(k)}\|_2^2 / \|d^{(k-1)}\|_2^2;\\
&\quad p^{(k+1)} = d^{(k)} + \beta_k p^{(k)};\\
&\}
\end{aligned} \tag{5.2.6}$$

Die wesentlichen Operationen sind die Matrix-Vektor-Multiplikation Ap, bei der wieder eine dünnbesetzte Struktur den Aufwand verringert, drei Vektor-Kombinationen und 2 Innenprodukte, welche alle ganz gut parallelisierbar sind mit $\mathcal{O}(n)$ Prozessoren. Im folgenden Satz werden die wesentlichen Eigenschaften zusammengefaßt, der Beweis konzentriert sich insbesondere auf die Konvergenzaussage, welche der Ansatzpunkt für weitere Effizienzsteigerungen ist. Denn die Konvergenz wird wesentlich durch die Spektral-*Kondition* $\hat{\kappa}(A) := |\lambda_{\max}(A)/\lambda_{\min}(A)| \geq 1$ bestimmt. Dazu nutzt man die Eigenschaft, dass für definites A das Funktional

$$2\phi(x) - 2\phi_{\min} = x^{\mathsf{T}} A x - 2b^{\mathsf{T}} x + b^{\mathsf{T}} A^{-1} b = (x - z)^{\mathsf{T}} A (x - z) =: \|x - z\|_A^2 \tag{5.2.7}$$

eine Norm ist und man den Fehler $x - z$ in dieser Norm darstellen kann.

Satz 5.2.1 *Ausgehend von einem Startwert $x^{(0)} \in \mathbb{R}^n$ sei im CG-Verfahren (5.2.6) $d^{(j)} \neq 0, j < k$. Dann gelten für die berechneten Größen folgende Aussagen.*

a) Die Suchrichtungen $p^{(1)}, \ldots, p^{(k)}$ sind paarweise A-orthogonal, die Defekte $d^{(0)}, \ldots, d^{(k-1)}$ orthogonal und jeweils Basen des Krylovraums

$$\mathcal{K}_k(A, d^{(0)}) = span\{p^{(1)}, \ldots, p^{(k)}\} = span\{d^{(0)}, \ldots, d^{(k-1)}\}.$$

b) Die Näherung $x^{(k)}$ minimiert das Funktional ϕ (5.2.3) über $x^{(0)} + \mathcal{K}_k$, es gilt

$$\phi(x^{(k)}) = \min\{\phi(x) : \ x \in x^{(0)} + \mathcal{K}_k\}.$$

Daher ist $x^{(k)}$ exakt, wenn $z \in x^{(0)} + \mathcal{K}_k$ gilt, also spätestens für $k = n$.

c) In der A-Norm (5.2.7) gilt die Fehlerschranke

$$\begin{aligned} \|x^{(k)} - z\|_A &= \min\{ \|g_k(A)\big(x^{(0)} - z\big)\|_A : \ g_k \in \Pi_k,\, g_k(0) = 1\} \\ &\leq 2\left(\frac{\sqrt{\hat\kappa(A)} - 1}{\sqrt{\hat\kappa(A)} + 1}\right)^k \|x^{(0)} - z\|_A. \end{aligned} \tag{5.2.8}$$

Beweis Die Behauptungen in a) und b) folgen mit einigem Aufwand aus der Verfahrensvorschrift.

c) Da der Startdefekt das Bild des Startfehlers $f^{(0)} := z - x^{(0)}$ ist, $d^{(0)} = b - Ax^{(0)} = A(z - x^{(0)}) = Af^{(0)}$, hat der Fehler von $x^{(k)}$ wegen a) mit einem Polynom $q_{k-1} \in \Pi_{k-1}$ die Darstellung

$$z - x^{(k)} = z - x^{(0)} + q_{k-1}(A)d^{(0)} = \underbrace{\big(I + q_{k-1}(A)A\big)}_{=:g_k}(z - x^{(0)}) = g_k(A)f^{(0)}.$$

Der Startfehler wird also mit $g_k(A)$ multipliziert, wobei $g_k(t) = 1 + tq_{k-1}(t)$ ein Polynom vom Grad k ist mit dem Wert $g_k(0) = 1$. Nach Teil b) ist g_k aber auch ein Polynom, das das Funktional ϕ und damit auch die A-Norm des Fehlers minimiert. Bei der symmetrischen Matrix A sei $A = U\Lambda U^{\mathsf T}$ die Jordan-Normalform mit $U^{\mathsf T} = U^{-1}$ und $\Lambda = \operatorname{diag}(\lambda_i)$. Damit ist

$$\begin{aligned} \|z - x^{(k)}\|_A^2 =& (z - x^{(k)})^{\mathsf T} A(z - x^{(k)}) = {f^{(0)}}^{\mathsf T} g(A)Ag(A)f^{(0)} = {f^{(0)}}^{\mathsf T} U\Lambda g(\Lambda)^2 U^{\mathsf T} f^{(0)} \\ \leq& \max_j g(\lambda_j)^2 {f^{(0)}}^{\mathsf T} U\Lambda U^{\mathsf T} f^{(0)} = \max_j g(\lambda_j)^2 \|f^{(0)}\|_A^2. \end{aligned}$$

Wie gesagt, minimiert das CG-Verfahren die Fehlernorm auf der linken Seite dieser Ungleichung. Eine obere Schranke bekommt man daher, indem man rechts irgendein "gutes" Polynom g_k einsetzt. Hier bieten sich die *Tschebyscheff-Polynome* aus der Numerik 1 an wegen ihrer Minimaleigenschaft, wobei man sie aber so verschiebt, dass sie die Nebenbedingungen $g_k(1) = 0$ erfüllen und dem Intervall $[a, b] = [\min \lambda_j, \max \lambda_j]$ der Eigenwerte von A klein sind. Tatsächlich erfüllt das Polynom

$$g_k(t) := T_k\Big(\frac{b + a - 2t}{b - a}\Big)\Big/T_k\Big(\frac{b + a}{b - a}\Big)$$

mit $T_k(t) = \cos(k \arccos t)$ diese Bedingungen und wegen $b/a = \hat\kappa(A)$ gilt die Schranke $|g_k(\lambda_j)| \leq 1/T_k(q)$ mit dem Wert $q := \big(\hat\kappa(A) + 1\big)/\big(\hat\kappa(A) - 1\big)$. Für den Normierungsfaktor $T_k(q)$ gilt die untere Schranke (vgl. Stoer/Bulisch 2, §8.7)

$$T_k(q) \geq \frac{1}{2}\left(\frac{\sqrt{\hat\kappa(A)} + 1}{\sqrt{\hat\kappa(A)} - 1}\right)^k.$$

Daraus folgt die Konvergenzaussage (5.2.8). ∎

Bis auf den unwichtigen Vorfaktor 2 ähnelt die Fehlerschranke (5.2.8) den Konvergenzaussagen für andere Iterationsverfahren wie etwa beim Gesamt- und Einzelschrittverfahren, wo man einen Fehlerverlauf mit Vorfaktoren der Form $(1 - 1/\hat\kappa)^k$ bekommt. Der entscheidende Vorteil beim CG-Verfahren ist dabei, dass der Konvergenzfaktor nur noch von $\sqrt{\hat\kappa}$ abhängt und für große

$\hat{\kappa} \gg 1$ erheblich kleiner ist. Im Referenzbeispiel mit Gittergröße m etwa hat man $\hat{\kappa} \sim m^2$ und daher

$$\frac{\sqrt{\hat{\kappa}}-1}{\sqrt{\hat{\kappa}}+1} = 1 - \frac{2}{\sqrt{\hat{\kappa}}+1} \cong 1 - \frac{2}{\sqrt{\hat{\kappa}}} \cong 1 - \frac{2}{m}.$$

Für moderne Anwendungen ist das allerdings noch nicht gut genug. Dazu greift man die Überlegung am Anfang des Abschnitts auf und betrachtet Umformungen des Gleichungssystems, d.h. man löst statt $Ax = b$ das äquivalente Problem

$$M^{-1}Ax = M^{-1}b.$$

Die Anforderungen an die Matrix M sind hier aber größer, sie muß symmetrisch und positiv definit sein. Dann existiert die symmetrische Cholesky-Zerlegung $M = LL^{\mathsf{T}}$ (Numerik 1) mit einer regulären unteren Dreieckmatrix L und das umgeformte System ist formal äquivalent mit

$$(LL^{\mathsf{T}})^{-1}Ax = (LL^{\mathsf{T}})^{-1}b \quad \Longleftrightarrow \quad \Big(\underbrace{L^{-1}A(L^{-1})^{\mathsf{T}}}\Big)(L^{\mathsf{T}}x) = L^{-1}b.$$

Die geklammerte Matrix ist wieder symmetrisch und positiv definit und das CG-Verfahren daher hier einsetzbar. Es gibt sogar eine Umformulierung, die direkt mit M und A arbeitet ohne L, wobei Gleichungssysteme der Form $My = r$ zu lösen sind. Entscheidend für die Konvergenz ist jetzt die neue Kondition $\hat{\kappa}(L^{-1}A(L^{-1})^{\mathsf{T}}) = \hat{\kappa}(M^{-1}A)$. Daher nennt man diesen Vorgang Präkonditionierung. Bei einer guten Präkonditionierung sollte $\hat{\kappa}(M^{-1}A) \ll \hat{\kappa}(A)$ sein für eine bessere Konvergenz, allerdings dürfen die Parallel-Kosten zur Auflösung von Systemen $My = r$ diesen Vorteil nicht zunichte machen. Gängige Verfahren werden in §5.7 behandelt.

Die Voraussetzungen für die Anwendung des CG-Verfahrens, Symmetrie und Definitheit der Matrix, sind natürlich sehr einschränkend. Für allgemeine Matrizen gibt es aber eine ganze Klasse von *Krylov-Verfahren*, die auf verwandten Prinzipien beruhen. Auch dort ist Präkonditionierung (mit allgemeinem M) ein gängiges Mittel zur Verbesserung der Konvergenz. Allerdings ist bei Nicht-Symmetrie der Verfahrensaufwand höher und/oder Konvergenzaussagen schwächer (vgl. Numerik 2A bzw. Spezialvorlesungen).

5.3 M-Matrizen

Für den Einsatz auf Parallelrechnern werden v.a. die auf regulären Zerlegungen beruhenden Verfahren modifiziert. Dabei kann man besonders weitgehende Aussagen bei einer speziellen Klasse von Matrizen machen, zu denen auch die Matrix des Referenzbeispiels (5.1.4) gehört. Eine wesentliche Eigenschaft dieser Matrixklasse ist eine bestimmte Vorzeichenverteilung der Elemente. Zur Beschreibung gelten Ungleichungen zwischen Matrizen bzw. Vektoren elementweise. Mit $\varrho(A)$ wird der Spektralradius einer Matrix A bezeichnet.

Definition 5.3.1 *Eine reguläre $n \times n$-Matrix A heißt* monoton*, wenn $A^{-1} \geq 0$ ist. Die Matrix heißt* M-Matrix*, wenn gilt*

$$A = \mu I - N, \quad N \geq 0, \quad \varrho(N) < \mu.$$

Das wichtigste Argument, das man bei monotonen Matrizen anwenden kann, ist die Folgerung $Ax = b \geq 0 \Rightarrow x = A^{-1}b \geq 0$. Die Definition über die Inverse ist natürlich schwer nachzuprüfen. Es gibt aber eine hinreichende Charakterisierung über die Ausgangsmatrix A.

Satz 5.3.2 *Jede M-Matrix ist monoton.*

Beweis Wegen $\varrho(\frac{1}{\mu}N) = \frac{1}{\mu}\varrho(N) < 1$ konvergiert die Neumannreihe

$$\sum_{j=0}^{\infty} \underbrace{\mu^{-j}N^j}_{\geq 0} = \mu(\mu I - N)^{-1} = \mu A^{-1}.$$

Da alle Reihenelemente nichtnegativ sind, gilt dies auch für A^{-1}. ∎

Bei M-Matrizen sind offensichtlich die Nebendiagonalelemente nicht positiv, hinreichend für die M-Matrix-Eigenschaft ist, z.B., zusätzlich strenge oder schwache Diagonaldominanz. Eine exakte Charakterisierung enthält der

Satz 5.3.3 *Eine Matrix mit der Vorzeichenverteilung*

$$a_{ii} > 0, \quad a_{ij} \leq 0 \, \forall j \neq i, \; i, j = 1, \ldots, n,$$

ist genau dann M-Matrix, wenn ein positiver Vektor $w > 0$ existiert mit $Aw > 0$.

Beweis a) Sei A monoton (z.B. M-Matrix) und $w = A^{-1}\mathbb{1}$ mit $\mathbb{1} := (1, .., 1)^T$. Dann ist $Aw = \mathbb{1} > 0$ und $w \geq 0$. Dabei kann keine Komponente von w verschwinden, da mit $w_i = 0$ die i-te Zeile von A^{-1} null wäre und A^{-1} daher nicht regulär. Für eine M-Matrix $A = \mu I - N$ sind wegen $N \geq 0$ die Nebendiagonalen nicht positiv und für jedes i ist $(\mu - n_{ii})w_i = (Aw)_i + \sum_{j \neq i} n_{ij} w_j > 0$, also $a_{ii} = \mu - n_{ii} > 0$.

b) Sei $Aw > 0$ und $w > 0$. Mit $W := \mathrm{diag}(w_i)$ werden gewichtete Vektor- und Matrix-Normen definiert durch

$$\|x\|_w := \|W^{-1}x\|_\infty, \quad \|B\|_w := \|W^{-1}BW\|_\infty.$$

Diese haben die Eigenschaft, dass für nichtnegative $B \geq 0$ die Matrixnorm $\|B\|_w$ gleich der Norm $\|Bw\|_w = \|B\|_w$ des Vektors Bw ist. Sei nun $\mu \geq \max_i a_{ii}$ gewählt. Dann ist $N = \mu I - A \geq 0$ und es gilt

$$Aw = \mu w - Nw > 0 \iff \mu w > Nw = NW\mathbb{1} \iff \mu\mathbb{1} > W^{-1}NW\mathbb{1}.$$

Die letzte Ungleichung bedeutet aber, dass $\mu > \|W^{-1}NW\mathbb{1}\|_\infty = \|W^{-1}NW\|_\infty \geq \varrho(N)$ gilt. ∎

Die w-Norm ist auch monoton, denn aus $0 \leq C \leq B$ folgt $\|C\|_w \leq \|B\|_w$.

Beispiel 5.3.4 Die Matrix zur diskreten Poisson-Gleichung (5.1.4) ist eine M-Matrix. Mit den Doppelindizes aus §5.1 wird der Testvektor $w_{ij} := a - (2i - m)^2 - (2j - m)^2$ betrachtet. Für genügend großes $a > 0$ ist $w_{ij} > 0$, $0 \leq i, j \leq m + 1$. Für w gilt außerdem

$$\begin{aligned}
&- w_{i-1,j} - w_{i,j-1} + 4w_{ij} - w_{i+1,j} - w_{i,j+1} \\
&= (2i - 2 - m)^2 + (2j - 2 - m)^2 - 2(2i - m)^2 - 2(2j - m)^2 + (2i + 2 - m)^2 + (2j + 2 - m)^2 \\
&= 4 - 4(2i - m) + 4 - 4(2j - m) + 4 + 4(2i - m) + 4 + 4(2j - m) = 16
\end{aligned}$$

in inneren Gitterpunkten. In Randnähe fehlen Nachbarwerte $-w_{i\pm1,j\pm1} < 0$, dort ist 16 eine untere, positive Schranke, also gilt $Aw > 0$.

Im Rahmen der Iterationsverfahren hat die M-Matrix-Eigenschaft eine große Bedeutung wegen der folgenden Sätze. Zur Präzisierung werden einige Begriffe definiert.

Definition 5.3.5 *Für eine reguläre Matrix A heißt die Aufspaltung $A = M - N$ eine* Zerlegung, *wenn M regulär ist. Eine Zerlegung heißt außerdem*

konvergent, wenn	$\varrho(M^{-1}N) < 1$,
regulär, wenn	$M^{-1} \geq 0$, $N \geq 0$,
schwach regulär, wenn	$M^{-1} \geq 0$, $M^{-1}N \geq 0$.

Bei M-Matrizen führt fast jede sinnvolle (z.B. disjunkte) Aufteilung der Matrixelemente auf die beiden Teile einer regulären Zerlegung auf eine konvergente Iteration. Außerdem wird die Konvergenz nicht schlechter, wenn man mehr Elemente aus A in M berücksichtigt.

Satz 5.3.6 *Es sei A eine M-Matrix.*

a) Jede reguläre Zerlegung ist auch konvergent, es gilt $\varrho(M^{-1}N) < 1$.

b) Gegeben seien zwei reguläre Zerlegungen $A = M_1 - N_1 = M_2 - N_2$ von A. Dann folgt

$$N_1 \leq N_2 \;\Rightarrow\; \varrho(M_1^{-1}N_1) \leq \varrho(M_2^{-1}N_2).$$

Beweis a) Da die minimale Komponente eines positiven Vektors positiv ist, existiert $w > 0$ mit $Aw = Mw - Nw \geq \epsilon\mathbb{1} > 0$. Mit $M^{-1} \geq 0$ folgt aus $MW\mathbb{1} \geq Nw + \epsilon\mathbb{1}$, dass

$$\mathbb{1} - \epsilon W^{-1}M^{-1}\mathbb{1} \geq W^{-1}M^{-1}NW\mathbb{1} \;\Rightarrow\; 1 - \delta \geq \|M^{-1}N\|_w \geq \varrho(M^{-1}N)$$

mit einem $\delta > 0$ so, dass $\delta w \leq \epsilon M^{-1}\mathbb{1}$.

b) Für jede reguläre Zerlegung $A = M - N$ ist $M = A + N$ und daher $B := M^{-1}N = (A+N)^{-1}N = (I+G)^{-1}G \geq 0$ mit $G := A^{-1}N \geq 0$. Die Eigenwerte λ von G und β von B hängen zusammen gemäß $\beta = \lambda/(1+\lambda)$. Nach dem Satz von Perron-Frobenius [R.S.Varga, Matrix iterative analysis] ist der Spektralradius $\varrho(B)$ ein reeller, nichtnegativer Eigenwert von B, also $0 \leq \varrho(B) = \lambda/(1+\lambda) < 1$ (nach Teil a)). Da die Funktion $\lambda \mapsto \lambda/(1+\lambda)$ wachsend ist in $\mathbb{R}_+$, wird $\varrho(B)$ durch den größten Eigenwert $\lambda = \varrho(G)$ bestimmt,

$$\varrho(B) = \varrho(G)/(1+\varrho(G)).$$

Dieser ist monoton in G und mit $N_1 \leq N_2 \;\Rightarrow\; G_1 = A^{-1}N_1 \leq A^{-1}N_2 = G_2 \;\Rightarrow\; \varrho(G_1) \leq \varrho(G_2)$ folgt die Behauptung. ∎

Zur Illustration sei an Gesamtschritt- und Einzelschrittverfahren erinnert, bei denen man die Matrix $A = D - L - R$ mit Diagonale D, sowie linker und rechter Dreieckmatrix darstellt. Das ESV entspricht der Zerlegung $M_1 = D - L$, $N_1 = R$ und das GSV verwendet $M_2 = D$, $N_1 = L + R$. Bei einer M-Matrix A ist natürlich $L, R \geq 0$ und daher tasächlich $M_1 = D - L \leq D = M_2$ und $N_1 = L \leq L + R = N_2$, hier konvergiert das ESV nicht schlechter als das GSV. Tatsächlich kann man beim Referenzproblem sogar zeigen, dass $\varrho(M_1^{-1}N_1) = \varrho(M_1^{-1}N_1)^2 < 1$ ist.

5.4 Mehrfachzerlegungen der Matrix

Wenn man $p > 1$ Prozessoren zur Verfügung hat, kann man anstelle einer einzelnen Iteration zu einer regulären Zerlegung mehrere unabhängige Schritte dieser Form überlagern. Dazu werden in jedem Iterationsschritt mehrere Teile mit verschiedenen Zerlegungen $A = M_j - N_j$, $j = 1, \ldots, p$, ausgeführt und die Resultate mit Matrizen $E_j \geq 0$ zu einem Ergebnis zusammengefügt. Man betrachtet also die Iteration

$$x^{(k+1)} := \sum_{j=1}^{p} E_j M_j^{-1}(N_j x^{(k)} + b), \quad \sum_{j=1}^{p} E_j = I,\ M_j - N_j = A,\ j = 1, \ldots, p. \tag{5.4.1}$$

Die in (5.4.1) genannten Nebenbedingungen garantieren, dass die Lösung z auch Fixpunkt der Iteration ist. Aus Effizienzgründen wählt man üblicherweise die Matrizen E_j als Diagonalmatrizen mit nur wenigen nichtnegativen Elementen (Maskierungs-, Gewichtungs-Matrizen). Dann berechnet Prozessor P_j auch nur diejenigen Komponenten von $M_j^{-1}(N_j x^{(k)} + b)$, für die das Diagonalelement in E_j positiv ist.

Ein besonders einfaches Beispiel einer solchen Mehrfachzerlegung ist das Block-Gesamtschrittverfahren (Block-Jacobi-Iteration). Man betrachtet wieder ein System mit Block-Matrix,

$$\begin{pmatrix} A_{11} & \cdots & A_{1p} \\ \vdots & & \vdots \\ A_{p1} & \cdots & A_{pp} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_p \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix}, \tag{5.4.2}$$

und $A_{ij} \in \mathbb{R}^{r \times r}$, $x_i, b_i \in \mathbb{R}^r$. Analog zum normalen Gesamtschritt-Verfahren (5.2.1) kann man eine Block-Version formulieren, wenn die Hauptdiagonalblöcke regulär sind,

$$x_i^{(k+1)} := A_{ii}^{-1}\Big(b_i - \sum_{j \neq i} A_{ij} x_j^{(k)}\Big),\ i \in \{1, .., p\}. \tag{5.4.3}$$

In der Formulierung wurde beim Index i keine Laufanweisung verwendet um zu betonen, dass die p Teile des Iterationsschritts (5.4.3) voneinander unabhängig sind und daher parallel durchgeführt werden können. Dies wurde schon als Vorteil des Gesamtschrittverfahren gegenüber dem Einzelschrittverfahren genannt. In der Darstellung (5.4.1) ist hierbei z.B. $M_j \equiv \mathrm{diag}(A_{11}, .., A_{pp})$ als Block-Diagonalmatrix zu wählen, die Gewichtsmatrizen E_j besitzen jeweils einen Identitätsblock an der Stelle j, $E_j = \mathrm{diag}(0, .., 0, I_r, 0, .., 0)$. Tatsächlich berechnet Prozessor P_i aber nur seinen Teilschritt aus (5.4.3). Zur Untersuchung der Konvergenz von (5.4.1) wird der Fehler $f^{(k)} := x^{(k)} - z$ zur Lösung z betrachtet. Für ihn gilt folgende einfache Beziehung

$$f^{(k+1)} = \sum_{j=1}^{p} E_j M_j^{-1}(N_j x^{(k)} + b - N_j z - b) = B f^{(k)}, \quad B := \sum_{j=1}^{p} E_j M_j^{-1} N_j. \tag{5.4.4}$$

In jedem Schritt wird der alte Fehler mit der Matrix B multipliziert, also ist $f^{(k)} = B^k f^{(0)}$. Mit den Beweistechniken des letzten Abschnitts folgt hierzu der

Satz 5.4.1 *Die Matrix A sei monoton, die E_j seien nicht-negativ und alle verwendeten Zerlegungen schwach regulär, d.h.*

$$\det M_j \neq 0,\ M_j^{-1} \geq 0,\ B_j := M_j^{-1} N_j \geq 0,\ j = 1, .., p.$$

Dann konvergiert (5.4.1), denn in (5.4.4) gilt $\varrho(B) < 1$ und $B \geq 0$.

Beweis Mit dem Vektor $w > 0$, $Aw \geq \epsilon \mathbb{1} > 0$ folgt für jede Zerlegung und $B_j = M_j^{-1} N_j$ dass

$$w - M_j^{-1} N_j w = M_j^{-1} A w \geq \epsilon M_j^{-1} \mathbb{1} \ \Rightarrow\ B_j w = M_j^{-1} N_j w \leq w - \epsilon M_j^{-1} \mathbb{1} < w.$$

Dies bedeutet wieder $W^{-1} B_j W \mathbb{1} < \mathbb{1}$, also $\|B_j\|_w < 1$ und damit ist $\beta := \max_j \|B_j\|_w < 1$. Für die Gesamt-Iterationsmatrix B aus (5.4.4) folgt daraus

$$0 \leq |B| w = \sum_{j=1}^{p} E_j B_j w \leq \beta \sum_{j=1}^{p} E_j w = \beta w \ \Rightarrow\ \|B\|_w \leq \beta < 1. \qquad \blacksquare$$

Bemerkung Der Beweis nutzt die auch in Satz 5.3.6 formulierte Tatsache, dass bei monotonen Matrizen jede schwach reguläre Zerlegung konvergiert wegen $\varrho(B_j) = \varrho(M_j^{-1} N_j) < 1$. Die Konvergenzaussage überträgt sich natürlich auf allgemeine Matrizen, wenn man für die Zerlegungen direkt die Annahme $\varrho(|B_j|) < 1$ macht. Diese muß dann aber unabhängig verifiziert werden.

Beispiel 5.4.2 Konvergenzfaktoren für Gesamtschritt-, Einzelschritt- und Block-Gesamtschritt-Verfahren bei der diskreten Poisson-Gleichung (5.1.4) der Dimension $n = m^2$. Mit Zeilenblöcken bei lexikografischer Nummerierung ist die Verbesserung der Konvergenz recht dürftig. Daher wird das Block-Verfahren mit 16 Kacheln (vgl. S.40) durchgeführt.

$n =$	256	1024	4096	16384
GSV $\varrho \doteq$	0.9230	0.99547	0.998827	0.999701
ESV $\varrho \doteq$	0.9663	0.99098	0.997663	0.999403
BJac $\varrho \doteq$	0.9357	0.96566	0.982145	0.990810

Das Ende eines Iterationsschrittes in (5.4.1) stellt einen gemeinsamen Synchronisationspunkt (*Barriere*) für alle beteiligten Prozessoren dar. Der j-te Prozessor muß nach Berechnung seines Teilvektors $E_j M_j^{-1}(N_j x^{(k)} + b)$ die zugehörigen Komponenten einer zentralen Stelle oder all denjenige Prozessoren mitteilen, die sie benötigen. Bei allgemeiner Block-Tridiagonalgestalt von A etwa sind dies nur der linke und rechte Nachbar. Bei einer geeigneten (geometrischen) Problemaufteilung muß mglw. sogar nur ein Bruchteil der Daten übermittelt werden, vgl. § 5.1. Der nächste Schritt kann erst nach Abschluß des aktuellen Iterationsschritts in allen Prozessoren beginnen. Daher sollten alle Prozessoren Teilaufgaben mit gleich viel Aufwand zugewiesen bekommen, bei regelmäßiger Struktur der Matrix (z.B. Bandgestalt) also Teile gleicher Blockgröße. Eine solche Aufteilung ist aber nicht immer einfach möglich, Gründe dafür werden im nächsten Abschnitt genannt.

5.5 Asynchrone Iteration

In einigen Anwendungsbereichen ist die Struktur der zuletzt behandelten Verfahren zu starr, da die Synchronisation am Ende eines Iterationsschritts viele Prozessoren warten lassen kann. Dies gilt z.B., in folgenden Situationen.

- Unterschiedlich aufwändige Teile: Bei technischen Problemen mit zusammengesetzten Baugruppen kann es im Gesamtalgorithmus am günstigsten sein, jedem Prozessor einen bestimmten geometrischen Teil zuzuordnen (mit speziellen Materialparametern, Teilebeschreibung,..). Aber auch bei einfacher Geometrie und einer gleichmäßigen Kachelung können die einzelnen Teile verschiedenen Aufwand verursachen, etwa wenn wegen der Gestalt der Lösung an bestimmten Stellen lokal feinere Gitter erforderlich sind. Unterschiede im Aufwand pro Iterationsteil (5.4.1) sind in diesen Fällen wahrscheinlich.
- Unterschiede der verfügbaren Prozessoren: Ein interessantes Denkmodell ist der Einsatz von fremden Rechnern in verteilten Systemen (Workstations im Netz, *Grid Computing*) für rechenintensive Probleme, wenn und solange diese unbenutzt sind. Diese Rechner sind dann meist unterschiedlich leistungsfähig. Auch kann der Rechner jederzeit wieder von seinem Besitzer in Anspruch genommen werden und fällt dann möglicherweise für die aktuell zugeteilte Aufgabe (teilweise) aus.

In beiden Fällen wäre es günstig, die schnelleren Prozessoren unabhängig von den Ergebnissen der anderen weitere Iterations-Teilschritte ausführen zu lassen, statt sie untätig zu lassen. Zwei Modelle sind bei Durchführung in einem gemeinsamem Speicher oder in einer Master-Slave-Anordnung denkbar:

1. Jeder Prozessor führt eine individuelle Zahl von lokalen Iterationen aus, bis alle Prozessoren bereit sind, eine neue globale Lösung zu berechnen.
2. Ein zur Zeit k freiwerdender Prozessor erhält einen neuen Teil j der Iteration (5.4.1) zugewiesen und berechnet dann $E_j M_j^{-1}(N_j x^{(k)} + b)$. Dieser Teil wird aber erst zu einem (viel) späteren Zeitpunkt $\geq k+1$ in die neue Näherung eingebracht.

Beide Möglichkeiten kann man mathematisch formulieren und dazu unterschiedlich genaue Aussagen herleiten. Im ersten Fall werden folgende affinen Iterations-Abbildungen definiert,

$$G_j(x) := M_j^{-1}(N_j x + b), \quad G_j^{\mu} := \underbrace{G_j \cdot .. \cdot G_j}_{\mu - mal}.$$

Die Iterationsvorschrift für die erste Version kann dann lauten,

$$x^{(k+1)} := \sum_{j=1}^{p} E_j G_j^{\mu_{jk}}(x^{(k)}), \tag{5.5.1}$$

wobei $\mu_{jk} \geq 0$ die Anzahl der lokal in P_j und im Schritt k durchgeführten Iterationen angibt. Diese Anzahl kann also nicht nur von der verwendeten Zerlegung, sondern auch vom Zeitpunkt, etwa der aktuellen Rechnerauslastung, abhängen. Die Analyse dieser Variante erfordert nur eine geringfügige Verallgemeinerung von Satz 5.4.1 und erlaubt sogar eine quantitative Aussage.

Satz 5.5.1 *Die Matrix A sei monoton, die E_j seien nicht-negativ, alle Zerlegungen schwach regulär und es gelte $\mu_{jk} \geq 1 \forall j,k$ in (5.5.1). Dann konvergiert die Iteration (5.5.1). Mit einem Vektor $w > 0$, für den $Aw > 0$ ist, sei $\beta_j := \|B_j\|_w < 1$, $j = 1,..,p$. Dann gilt die Fehlerschranke*

$$\|x^{(k+1)} - z\|_w \leq \left(\max_{j=1}^{p} \beta_j^{\mu_{jk}} \right) \|x^{(k)} - z\|_w.$$

Beweis Für die Fehler $d^{(k)} = x^{(k)} - z$ erhält man wieder mit $B_j = M_j^{-1} N_j$ die einfache Darstellung

$$d^{(k+1)} = B^{(k)} d^{(k)}, \quad B^{(k)} = \sum_{j=1}^{p} E_j B_j^{\mu_{jk}}.$$

Die Iterationsmatrix ist also nicht mehr unabhängig vom Zeitpunkt. Aber aus $B_j \geq 0$ und $\|B_j\|_w \leq \beta_j$, d.h. $B_j w \leq \beta_j w$ folgt wie bei Satz 5.4.1

$$0 \leq B^{(k)} w = \sum_{j=1}^{p} E_j B_j^{\mu_{jk}} w \leq \sum_{j=1}^{p} \beta_j^{\mu_{jk}} E_j w \leq \big(\max_{j=1}^{p} \beta_j^{\mu_{jk}} \big) w.$$

Dies führt auf die Fehlerschranke und die Konvergenzaussage. ∎

Der Satz gibt auch gleichzeitig einen Hinweis zur Wahl der lokalen Iterationsanzahlen $\mu_j = \mu_{jk}$ unabhängig von k: Bei Aufteilung der Iteration sollten erstens die Teilprobleme so gewählt werden, dass der Zeitbedarf zur Durchführung der jeweils μ_j Iterationen für alle Teile gleich groß ist, es sollten aber auch alle Faktoren $\beta_j^{\mu_j}$ ungefähr den gleichen Wert haben.

Das zweite Iterationsmodell ist allgemeiner als das erste, dabei setzt man den Iterationszähler nicht mit einer gleichmäßig ablaufenden Zeit gleich. Die einzelnen Teilschritte von (5.5.1) können auf folgende Weise zeitlich entzerrt werden. Hier ist es nicht mehr notwendig, von einer gleichen Anzahl von Prozessoren und Zerlegungen auszugehen. Die Anzahl der Zerlegungen sei jetzt m und $\{j_k\}$ eine Folge mit $j_k \in \{1,..,m\}$. Außerdem seien $r(j,k)$ Zahlen mit $0 \leq r(j,k) \leq k$. Da es jetzt keinen globalen Takt mehr gibt, wird der Iterationszähler k mit jedem eintreffenden Teilergebnis $j = j_k$ weitergezählt. Die Iteration lautet dann so:

$$\begin{aligned} E_i y^{(k)} &:= E_i x^{(k-r(i,k))}, \quad i = 1,..,m, \\ x^{(k+1)} &:= (I - E_j) x^{(k)} + E_j M_j^{-1} (N_j y^{(k)} + b). \end{aligned} \tag{5.5.2}$$

Diese Vorschrift hat folgenden Hintergrund: Das Teilergebniss eines Zeitschritts $k \to k+1$ wird von einem freien Prozessor mit der Zerlegung j_k berechnet. Dieser Prozessor kannte zu Beginn seiner Berechnung aber unterschiedlich aktuelle Teile der Näherungslösung $x^{(\cdot)}$. Die Vorschrift (5.5.2) kann auch eine vollkommen dezentrale Iteration beschreiben, bei der (analog zu früher besprochenen Verfahren) in einem Netz dem Prozessor P_j eine Zerlegung E_j (bzw. mehrere Zerlegungen) fest zugeordnet wird. Die in P_j zum Zeitpunkt k vorhandenen Komponenten $E_j x$ sind

die aktuellsten. Die anderen Prozessoren arbeiten dagegen mit älteren Versionen $E_j x^{(k-r(j,k))}$ dieser Komponenten, die im Netz kursieren.

Eine Konvergenzaussage ist bei diesem Verfahren natürlich nur möglich, wenn alle Zerlegungen genügend oft berücksichtigt werden und eine absolute obere Schranke für die Verzögerungen vorausgesetzt wird. Diese Schranke kann man bei praktischer Realisierung durch eine (zentrale oder kooperative) Überwachung erzwingen: Trifft ein Teilergebnis nicht innerhalb einer bestimmten Zeitspanne ein, so wird die Teilaufgabe neu vergeben und eine spätere Meldung des säumigen Prozessors ignoriert. Eine Verkleinerung des Fehlers kann dann in genügend großen Zeitspannen nachgewiesen werden.

Satz 5.5.2 *Die Matrix A sei monoton, die E_j nicht-negativ, $\sum_{j=1}^{m} E_j = I$, und alle m Zerlegungen von A seien schwach regulär. Falls mit einem $T > m$ in jedem Abschnitt der Folge (j_k) der Länge T jeder Index $1, .., m$ mindestens einmal vorkommt, und die Verzögerungen in (5.5.2) beschränkt sind, $r(j,k) < T$, dann konvergiert die Iterationsfolge aus (5.5.2) für jeden Startvektor $x^{(0)}$.*

Beweis: Bru/Elsner/Neumann, Linear Algebra Applcs 103:175-192(1988).

Diese qualitative Aussage ist natürlich nur dann ausreichend, wenn durch die verteilte Verarbeitung Probleme behandelt werden können, die auf einem einzelnen Prozessor nicht durchführbar waren. Im anderen Fall wäre eine quantitative Aussage über die erreichte Beschleunigung (wenn überhaupt) von Interesse. Als Beispiel wird ein verzögertes Gesamtschrittverfahrens zu $A = I - B,\ B = (B_{ij})_{i,j=1}^{m}$ betrachtet. Die Zerlegungen können dann allein durch die E_j beschrieben werden. Im einfachsten Fall, bei dem jeder von p Prozessoren einen gleich aufwändigen Teil erhält, ergibt sich

$$x^{(k)} := (I - E_{j_k})x^{(k-1)} + E_{j_k}(Bx^{(k-p)} + b), k = p, p+1, \ldots .$$

Für die zyklische Steuerungsfolge $j_k := k \bmod m + 1$ und $B \geq 0,\ \varrho(B) < 1$, läßt sich zeigen, dass der asymptotische Konvergenzfaktor dieser Iteration *größer* wird für wachsende Prozessorzahl p, die Konvergenz wird also schlechter. Dies ist nicht verwunderlich, da die jeweils verwendete Information mit wachsendem p immer älter wird. Dennoch ergibt sich für kleinere p zunächst meist eine Beschleunigung.

Die Aussage wird illustriert durch ein Beispiel aus [Elsner/Neumann/Venner, Lin.Alg.Applcs 154:311-330(1991)]: Matrixgröße 80×80, Blöcke der Größe 8, d.h. $m = 10$ Zerlegungen. Der Eintrag K bezeichnet denjenigen Iterationsindex, bei dem erstmalig der Maximalfehler um 10^{-5} verkleinert wurde.

p	K	K/p	pK_1/K_p
1	101	101.0	1.00
2	134	67.0	1.51
3	162	54.0	1.97
4	194	48.5	2.08
5	204	40.8	2.48
6	212	35.3	2.86
7	244	34.9	2.89
8	261	32.6	3.10
..	..	..	
17	434	25.5	3.96
18	454	25.3	3.99
19	474	24.9	4.06
20	484	24.2	4.17

Die letzte Spalte enthält den Beschleunigungsfaktor. Zunächst ergibt sich also eine merkliche Beschleunigung bis zu ca. 6 Prozessoren, bei größeren $p \cong 17$ wächst die Steigerung dann praktisch überhaupt nicht mehr.

Bemerkung: Bei nichtlinearen Gleichungssystemen $F(x) = 0$, etwa aus der Diskretisierung eines nichtlinearen Randwertproblems, denkt man zunächst an den Einsatz der besprochenen Iterationsverfahren zur Lösung der linearen Gleichungssysteme im *Newton-Verfahren*. Unter geeigneten Monotonie-Voraussetzungen an die Funktion F können die Zerlegungs-Verfahren aber auch direkt beim nichtlinearen System eingesetzt werden. Dabei würde man etwa den Schritt (5.5.2) ersetzen durch die nichtlineare Variante

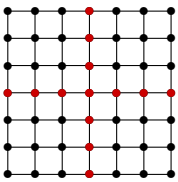
$$x^{(k+1)} := x^{(k)} - E_j M_j^{-1} F(y^{(k)}),$$

die direkt eine verbesserte Approximation des nichtlinearen Problems liefert.

5.6 Gebietszerlegungs-Verfahren

Der Name *Gebietszerlegung* (*domain decomposition*) deutet an, dass man beim Differenzenverfahren (5.1.4) oder anderen Diskretisierungen primär das Gebiet unterteilt und die Freiheitsgrade geeignet nummeriert, die geometrischen Zusammenhänge stehen bei der Aufteilung im Vordergrund. Gebietszerlegung kann man sowohl zum Einsatz mit Eliminations- als auch Iterationsverfahren betrachten, im letzteren Fall entspricht ihr Arbeitsprinzip oft den besprochenen Mehrfachzerlegungen. Bei der Unterteilung verwendet man unterschiedliche Strategien. Wenn man das Gebiet in Kacheln zerteilt, aber Trennfugen läßt, welche als letztes nummeriert werden (rot), bekommt man eine Matrix in Pfeilform.

Kacheln mit Trennfuge

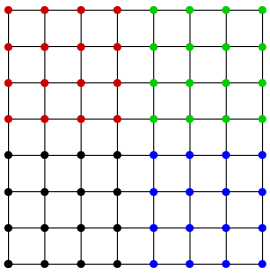


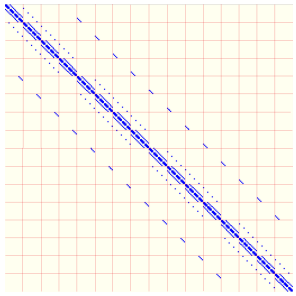
$$\begin{pmatrix} D_1 & & & & B_1 \\ & D_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & D_{m-1} & B_{m-1} \\ C_1 & C_2 & \dots & C_{m-1} & D_m \end{pmatrix}$$

Diese hat beim Gauß-Algorithmus den Vorteil, dass die Struktur erhalten bleibt, hier sind $m-1$ Systeme $D_k E_k = B_k$ mit dünnbesetzten Blöcken D_k zu lösen und ein (i.d.R. kleineres) mit der i.a. vollbesetzten Matrix $\hat{D}_m = D_m - C_1 E_1 - \ldots - C_{m-1} E_{m-1}$. Die Berechnung von $\hat{D}_m$ ist aber schlecht parallelisierbar.

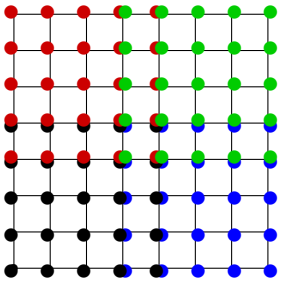
Iterationsverfahren mit geometrischem Hintergrund werden meist als *Schwarzsche Verfahren* bezeichnet, nach H.A. Schwarz (1870) der das Prinzip in der Funktionentheorie einsetzte. Hier betrachtet man Kachelungen wie in §5.1 ohne Trennfugen oder sogar mit Überlappungen. Die fugenlose Kachelung erzeugt recht dicht besetzte Diagonalblöcke und einige verstreute, aber sehr dünn besetzte Nebendiagonalen (linkes und mittleres Diagramm).

Kachelung fugenlos



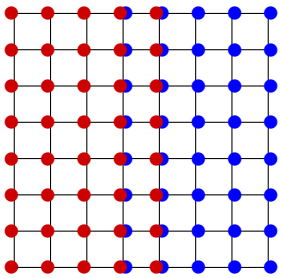


Kachelung überlappend



Die Block-Jacobi-Iteration (5.4.3) bearbeitet die Kacheln parallel und es müssen dabei nur sehr wenige Elemente der aktuellen Näherung $x^{(k)}$ mit anderen Prozessoren ausgetauscht werden. Vor dem Hintergund des Ausgangsproblems, der Poisson-Gleichung (5.1.1), kann man das Verfahren auch so interpretieren, dass man parallel auf jeder Kachel ein eigenes Randwertproblem löst, wobei Nachbarwerte aus anderen Kacheln dabei als Randwerte eingesetzt werden. Bei dieser Interpretation können sich die Kacheln jetzt auch überlappen wie im rechten Diagramm oben, erst Nachbarwerte, die außerhalb der bearbeiteten Kachel liegen, werden als Randwerte von Nachbarkacheln übernommen. Der Grund für die Nutzung größerer Überlappungen ist die Erwartung einer besseren Konvergenz. In den Überlappungsbereichen liegen nach einem parallelen Schritt mehrere Lösungswerte vor, von denen man einen Mittelwert bildet.

Als Beispiel wird eine Zerlegung des Quadrats $\Omega = [0,1]^2$ in zwei Kacheln Ω_1, Ω_2 diskutiert, welche sich in der Mitte überlappen, $\Omega_1 \cap \Omega_2 \neq \emptyset$, $\Omega = \Omega_1 \cup \Omega_2$. Dadurch werden die Variablen in 3 Gruppen unterteilt, x_1 enthält die rein roten Werte aus $\Omega \setminus \Omega_2$, x_3 die blauen aus $\Omega \setminus \Omega_1$ und x_2 die aus dem Schnitt $\Omega_1 \cap \Omega_2$. Da hier keine Kopplung zwischen x_1 und x_3 existiert, hat das Gleichungssystem die Block-Tridiagonalform



$$\begin{pmatrix} A_1 & B_1 & 0 \\ C_2 & A_2 & B_2 \\ 0 & C_3 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}. \tag{5.6.1}$$

Bei geringer Überlappung ist der Block A_2 kleiner als die anderen. Im parallelen Schwarz-Schritt werden mit einer aktuellen Näherung $x = x^{(k)}$ die Teilsysteme für die Kacheln gelöst

$$\begin{pmatrix} A_1 & B_1 \\ C_2 & A_2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 - B_2 x_3 \end{pmatrix}, \quad \begin{pmatrix} A_2 & B_2 \\ C_3 & A_3 \end{pmatrix} \begin{pmatrix} v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} b_2 - C_2 x_1 \\ b_3 \end{pmatrix}. \tag{5.6.2}$$

Der Vollständigkeit halber kann man noch $y_3 := x_3$ und $v_1 := x_1$ definieren. Da im Überlappungsbereich jetzt zwei Näherungen vorliegen, ist dort für die neue Lösung ein Mittelwert zu bilden:

$$x^{(k+1)} := \begin{pmatrix} y_1 \\ \frac{1}{2}(y_2 + v_2) \\ v_3 \end{pmatrix} = E_1 y + E_2 v, \quad (E_j)_{ii} := \begin{cases} 1 & \text{Gitterpunkt } i \text{ in } \Omega_j \setminus \Omega_{3-j} \\ \frac{1}{2} & \text{Gitterpunkt } i \text{ in } \Omega_1 \cap \Omega_2 \\ 0 & \text{sonst} \end{cases} \tag{5.6.3}$$

Dann liegt tatsächlich eine Mehrfachzerlegung (5.4.1) vor mit den eben definierten Gewichtsmatrizen E_j. In den Blockzerlegungen ist in N_1 nur der Block B_2 nichttrivial und in N_2 nur C_2.

Zur Diskussion der Konvergenz gilt mit der Lösung z im ersten System

$$\begin{pmatrix} A_1 & B_1 \\ C_2 & A_2 \end{pmatrix} \begin{pmatrix} y_1 - z_1 \\ y_2 - z_2 \end{pmatrix} = \begin{pmatrix} b_1 - b_1 \\ b_2 - B_2 x_3 - b_2 + B_2 z_3 \end{pmatrix} = \begin{pmatrix} 0 \\ -B_2(x_3 - z_3) \end{pmatrix}.$$

Dies ist die homogene Variante von (5.6.2). Für die Fehler betrachtet man (5.6.2) jetzt also mit $b = 0$. Wenn die Blöcke A_i regulär sind, kann man die Gleichungen formal auflösen:

$$\begin{array}{lll} A_1 y_1 + B_1 y_2 = 0 & \to & y_1 = -A_1^{-1} B_1 y_2 \qquad\qquad y_1 = A_1^{-1} B_1 (A_2 - C_2 A_1^{-1} B_1)^{-1} B_2 x_3 \\ C_2 y_1 + A_2 y_2 = -B_2 x_3 & \Downarrow & (A_2 - C_2 A_1^{-1} B_1) y_2 = -B_2 x_3 \quad \Uparrow \end{array}$$

Die Matrix $A_2 - C_2 A_1^{-1} B_1$ nennt man auch *Schur-Komplement*. Analog läßt sich auch der zweite Schritt durchführen. Man bekommt so für die Übergänge $x \mapsto y$, $x \mapsto v$, sowie $x \mapsto x^{(k+1)}$ die Matrizen

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 0 & Y_1 \\ 0 & 0 & Y_2 \\ 0 & 0 & I \end{pmatrix}}_{=:P_1} x, \quad \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \underbrace{\begin{pmatrix} I & 0 & 0 \\ Z_2 & 0 & 0 \\ Z_3 & 0 & 0 \end{pmatrix}}_{=:P_2} x, \quad x^{(k+1)} = \underbrace{\begin{pmatrix} 0 & 0 & Y_1 \\ \frac{1}{2} Z_2 & 0 & \frac{1}{2} Y_2 \\ Z_3 & 0 & 0 \end{pmatrix}}_{:=Q} x.$$

Dabei ist $Y_1 = A_1^{-1} B_1 (A_2 - C_2 A_1^{-1} B_1)^{-1} B_2$ und $Z_3 = A_3^{-1} C_3 (A_2 - B_2 A_3^{-1} C_3)^{-1} C_2$. Die Matrizen Y_2, Z_2 spielen keine Rolle. Die auftretenden Matrizen habe folgende Eigenschaften.

Satz 5.6.1 *Die Hauptdiagonalblöcke A_i der Systemmatrix in* (5.6.1) *und die Schurkomplemente $A_2 - C_2 A_1^{-1} B_1$, $A_2 - B_2 A_3^{-1} C_3$ seien regulär. Dann sind die Matrizen P_1, P_2 Projektoren, es gilt $P_j^2 = P_j$, $j = 1, 2$. Das Verfahren* (5.6.2), (5.6.3) *konvergiert genau dann für alle Startwerte, wenn $\varrho(Y_1 Z_3) < 1$ ist. Hinreichend dafür ist $q_1 + q_3 < 1$ mit $\|C_2 A_1^{-1} B_1 A_2^{-1}\| \le q_1$ und $\|B_2 A_3^{-1} C_3 A_2^{-1}\| \le q_3$. In diesem Fall gilt für den Konvergenzfaktor*

$$\varrho(Q) \le \sqrt{\frac{q_1 q_3}{(1 - q_1)(1 - q_3)}} < 1.$$

Beweis Nach Vertauschung der beiden ersten Variablengruppen bekommt man für die Iterationsmatrix

$$Q \sim \begin{pmatrix} 0 & \frac{1}{2} Z_2 & \frac{1}{2} Y_2 \\ 0 & 0 & Y_1 \\ 0 & Z_3 & 0 \end{pmatrix}.$$

Diese ist hat offensichtlich einen Kern in der Dimension des Matrixblocks A_2, insbesondere haben Y_2, Z_2 keinen Einfluß auf die Eigenwerte. Die anderen Eigenwerte sind die Wurzeln der Eigenwerte von $Y_1 Z_3$. Also ist $\varrho(Q) = \sqrt{\varrho(Y_1 Z_3)} = \sqrt{\varrho(Z_3 Y_1)}$. Für das Produkt bekommt man die Darstellung

$$Y_1 Z_3 = A_1^{-1} B_1 (A_2 - C_2 A_1^{-1} B_1)^{-1} B_2 A_3^{-1} C_3 (A_2 - B_2 A_3^{-1} C_3)^{-1} C_2.$$

Da der Spektralradius eines Produkts von 2 Matrizen unabhängig von der Reihenfolge der Faktoren ist, gilt

$$\begin{aligned}\varrho(Y_1Z_3) =&\varrho\Big(\underbrace{C_2A_1^{-1}B_1}(A_2-\underbrace{C_2A_1^{-1}B_1})^{-1}\underbrace{B_2A_3^{-1}C_3}(A_2-\underbrace{B_2A_3^{-1}C_3})^{-1}\Big)\\ =&\varrho\Big(Q_1(I-Q_1)^{-1}Q_3(I-Q_3)^{-1}\Big),\quad Q_1:=C_2A_1^{-1}B_1A_2^{-1},\ Q_3:=B_2A_3^{-1}C_3A_2^{-1}.\end{aligned}$$

Dies sind die Matrizen aus der Formulierung des Satzes, vorausgesetzt war $\|Q_j\| \le q_j$, $j = 1,3$. Wenn γ ein Eigenwert von Q_j ist, dann gehört zu $Q_j(I-Q_j)^{-1}$ der Eigenwert $\gamma/(1-\gamma)$ mit $|\gamma| \le q_j$. Für $q_1+q_3<1$ ist $q_1q_3 < (1-q_1)(1-q_3)$, also $\varrho(Q)<1$. ∎

Die Bedeutung der Voraussetzungen wurde noch nicht ganz klar. Dazu werden die im Satz auftretenden Matrix-"Quotienten" wieder in eine Blockmatrix eingetragen:

$$\begin{pmatrix}0 & B_1A_2^{-1} & 0\\ C_2A_1^{-1} & 0 & B_2A_3^{-1}\\ 0 & C_3A_2^{-1} & 0\end{pmatrix} = \begin{pmatrix}0 & B_1 & 0\\ C_2 & 0 & B_2\\ 0 & C_3 & 0\end{pmatrix}\begin{pmatrix}A_1 & & \\ & A_2 & \\ & & A_3\end{pmatrix}^{-1}.$$

Dieses ist aber i.w. die Iterationsmatrix des Block-Jacobi-Iteration bei (5.6.1), deren Spektralradius durch

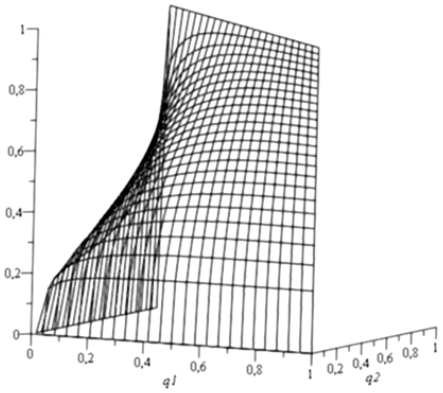
$$\sqrt{\|C_2A_1^{-1}B_1A_2^{-1}\| + \|B_2A_3^{-1}C_3A_2^{-1}\|} = \sqrt{q_1+q_2}$$

abgeschätzt werden kann. Denn man kann die erste und dritte Zeile $B_1A_2^{-1}x_2 = \lambda x_1$, $C_3A_2^{-1}x_2 = \lambda x_3$ der Eigenwertgleichung für $\lambda \neq 0$ in die zweite einsetzen zu $(Q_1+Q_2)x_2 = \lambda^2 x_2$.

Daher lassen sich die Konvergenzraten mit Überlappung (2 Kacheln) und ohne (3 diskjunkte Kacheln) direkt vergleichen. Aufgrund der Produktform der Schranke aus Satz 5.6.1 ist die Konvergenz dort sicher besser, wenn eine der Größen q_j klein ist. Im Detail zeigt die Graphik den Quotienten der Schranken

$$\sqrt{\frac{q_1q_3}{(1-q_1)(1-q_3)(q_1+q_3)}}$$

der Konvergenzfaktoren im Bereich $0 \le q_1+q_2 \le 1$ aber überall die wesentlich bessere Konvergenz bei Überlappung.



5.7 Parallele Präkonditionierung

Die besprochenen Iterationsverfahren zur Lösung eines LGS $Ax = b$ arbeiteten nach grundlegenden, allgemeinen Prinzipien weitgehend unabhängig von der Gestalt des Ausgangsproblems. Allerdings sind ihre Konvergenzvoraussetzungen i.d.R. sehr einschränkend und die Konvergenzgeschwindigkeit bei großen Problemdimensionen gering. Die moderne Sichtweise ist daher, dass man die Verfahren sowieso auf äquivalente, präkonditionierte Systeme

$$M^{-1}Ax = M^{-1}b, \qquad \big(AM^{-1}\big)(Mx) = b$$

anwendet und durch eine geeignete Wahl des Präkonditionierers M (z.B. in einer regulären Zerlegung) eine genügend gute Konvergenz erreicht. Man kann dabei noch zwischen Links-Präkonditionierung $M^{-1}A$ und Rechts-Präkonditionierung AM^{-1} unterscheiden. Bei der Konstruktion von M muss man jetzt aber meist spezielle Eigenschaften der Aufgabenstellung berücksichtigen, insbesondere natürlich eine dünne Besetzung. Generell einsetzbare Präkonditionierer gibt es kaum. Um eine etwas genauere Vorstellung zu vermitteln, werden zunächst einige gängige Präkonditionierungen behandelt, die insbesondere auch beim Referenzbeispiel einsetzbar ist. Danach wird eine Methode zur parallelen Präkonditionierung vorgestellt. Bei der Diskussion ist zu beachten, dass man oft durchaus einigen Aufwand in die Konstruktion von Präkonditionierern investieren kann, nämlich wenn die ursprüngliche Konvergenz sehr langsam ist, oder wenn ähnlich aufgebaute Gleichungssysteme mehrfach zu lösen sind. Letzteres ist bei nichtlinearen Problemen im Rahmen der Newton-Iteration der Fall. Bekannte Standard-Präkonditionerer:

- Überraschenderweise kann man die Standardverfahren aus §5.2 wechselseitig zur Präkonditionierung einsetzen, indem man Iterationszyklen des einen in das andere einbaut, wenn die Verfahren auf unterschiedlichen Prinzipien beruhen. Ein wichtiger Fall ist die Verwendung von Einzelschritt- oder SOR-Verfahren zur Präkonditionierung beim CG-Verfahren. Zur Bewahrung der Symmetrie sind dabei Vorwärts- und Rückwärtsschleifen durch die Matrix zu kombinieren ($\rightarrow$ SSOR). Bezeichnet man wieder mit $A = D - L - L^{\mathsf{T}}$, die Aufteilung der symmetrischen Matrix, dann entspricht der Einsatz des SSOR-Verfahrens mit Parameter $\omega > 0$ der Präkonditionierung mit

  $$M_\omega = (D - \omega L)D^{-1}(D - \omega L^{\mathsf{T}}).$$

  Beim Referenzbeispiel (5.1.4) mit Schachbrettnummerierung ist dies sogar ganz gut parallelisierbar.

- Ein wichtiger Grund für den relativ hohen Rechenaufwand des Gauß-Algorithmus bei dünnbesetzten Matrizen ist das Auffüllen ("fill-in") bei der Elimination. Denn beim Basisschritt $a_{ij}^{(k+1)} := a_{ij}^{(k)} - a_{ik}^{(k+1)} a_{kj}^{(k)}$ ensteht an einer leeren Stelle mit $a_{ij}^{(k)} = 0$ immer dann ein neues nichttriviales Element, wenn beide Faktoren $a_{ik}^{(k+1)} a_{kj}^{(k)} \neq 0$ sind. Bei Matrizen mit großen Diagonalelementen (z.B. definit, diagonaldominant) macht man in diesem Fall wegen der Beziehung

  $$a_{ij}^{(k+1)} := 0 - \frac{a_{ik}^{(k)} a_{kj}^{(k)}}{a_{kk}^{(k)}} \tag{5.7.1}$$

  aber folgende Beobachtung: da die Diagonaldominanz bei der Elimination für die Restmatrizen $(a_{ij}^{(k)})_{i,j \geq k}$ nicht verloren geht (Numerik 1, S.4.3.4), überwiegt der Nenner und die neu eingeführten nichttrivialen Elemente werden immer kleiner. Dadurch kam man auf die Idee, Auffüllelemente einfach zu vernachlässigen in einer *unvollständigen LR-Zerlegung* (ILU=*incomplete lower-upper*). Dabei unterscheidet man anhand der "Generation" der

Elemente die Verfahren

ILU(0): es werden nur Elemente aus der ursprünglichen Besetzungsstruktur der Matrix A berücksichtigt,

ILU(1): $a_{ij}^{(k+1)}$ in (5.7.1) wird nur berücksichtigt, wenn $a_{ik}a_{kj} \neq 0$ in A.

ILU(m): beide Faktoren stammen aus ILU($0 \dots m-1$).

Ein wichtiger Vorteil der einfachsten Version ILU(0) ist, dass für die Matrix A eine statische Datenstruktur verwendet werden kann, da keine neuen Elemente hinzukommen. Auch der Speicherbedarf für ILU(1) läßt sich mit etwas Aufwand vorab formal bestimmen. Zur Untersuchung sei jetzt

$$A = \tilde{L}\tilde{R} - N = M - N, \quad M = \tilde{L}\tilde{R}$$

die unvollständige Zerlegung mit Fehler N, zur Präkonditionierung verwendet man das Produkt der Faktoren. Gleichungssysteme der Form $My = \tilde{L}\tilde{R}y = z$ sind natürlich einfach auflösbar. Bei einer einfachen Iteration ist dann

$$\varrho\big(M^{-1}N\big) = \varrho\big((A+N)^{-1}N\big) = \varrho\big((I + A^{-1}N)^{-1}A^{-1}N\big) \leq \frac{\varrho(A^{-1}N)}{1-\varrho(A^{-1}N)}.$$

Dieser Wert ist kleiner eins, wenn $\varrho(A^{-1}N) < \frac{1}{2}$ ist, anzustreben sind aber natürlich viel kleinere Werte. Bei Krylov-Verfahren ist im umformulierten System $M^{-1}Ax = M^{-1}b$ die Matrix i.w.

$$M^{-1}A = (A+N)^{-1}A = \big(I + A^{-1}N\big)^{-1} \;\Rightarrow\; \lambda_j(M^{-1}A) = \frac{1}{1+\lambda_j(A^{-1}N)}$$

Die Kondition ist der Quotient zwischen größtem und kleinstem Eigenwert und daher beschränkt durch $\hat{\kappa}(M^{-1}A) \leq (1+\rho)/(1-\rho)$ mit $\rho = \varrho(A^{-1}N) \overset{!}{<} 1$. Das präkonditionierte CG-Verfahren konvergiert besser als vorher, wenn diese Kondition kleiner als $\hat{\kappa}(A)$ selber ist. Dies läßt sich erzwingen, indem man genügend viele Element-Generationen verwendet. Dabei ist der Übergang zu einem direkten Verfahren (alle Generationen) fließend. Grundvoraussetzung für die Funktionsweise ist natürlich eine Art Diagonaldominanz.

Beide Methoden haben für Parallelrechner den Nachteil einer begrenzten Effizienz auch wegen der Verwendung von Dreieckmatrizen. Interessant ist aber, dass sich durch die Verwendung dünnbesetzter (Dreieck-) Faktoren eine erhebliche Aufwands-Ersparnis ergibt. Bei der direkten Verwendung der Inversen $M^{-1} = \tilde{R}^{-1}\tilde{L}^{-1}$, bei der die Multiplikation gut parallelisierbar wäre, geht die dünne Besetzung fast immer verloren.

5.7.1 Approximativ Inverse

Dennoch kann man diese Ideen kombinieren durch Verwendung von Präkonditionierung mit Approximativ-Inversen, die direkt $P = M^{-1}$ konstruieren, um die Abhängigkeiten bei der Auflösung von Hilfs-Systemen mit M zu vermeiden. Dazu gibt man eine feste Belegungsstruktur $\mathcal{S} \subseteq \{(i,j) : 1 \leq i,j \leq n\}$ vor und sucht nach einer Matrix P mit dieser Struktur, die möglichst

nahe an A^{-1} liegt ($\rightarrow$ Name). Da man die Inverse nicht kennt, betrachtet man den Defekt in den definierenden Bedingungen $AP - I = 0$ oder $PA - I = 0$. Aus pragmatischen Gründen mißt man die Abweichung meist in der Frobeniusnorm, da diese leicht auswertbar ist und außerdem das Minimierungsproblem dann in kleinere Teile zerfällt. Nach Einführung der Menge $\mathcal{G}_\mathcal{S}$ aller Matrizen deren nichttriviale Elemente Indizes in $\mathcal{S}$ haben, lautet die Aufgabe also

$$\min\{\|AP - I\|_F^2 : P \in \mathcal{G}_\mathcal{S}\}, \tag{5.7.2}$$

oder analog die Minimierung von $\|PA-I\|_F^2$. Da in der Frobeniusnorm $\|AP-I\|_F^2 = \sum_{j=1}^n \|Ap_j - e_j\|_2^2$ die n Defekte mit den einzelnen Spalten $p_j = Pe_j$ unabhängig voneinander sind, zerfällt das Problem in unabhängige Teile. Bei der Wahl von $\mathcal{S}$ kann man zunächst einmal eine explizite Vorgabe anhand problemspezifischer Erkenntnisse vorsehen oder nur die Paare für eine Diagonalmatrix. Entscheidender sind aber adaptive Strategien, wo man kleine Matrixelemente einer berechneten (Näherungs-) Lösung verwirft (*dropping strategy*) oder bei einem zu großen Residuum $\|Ap_j - e_j\|_2^2$ neue Indizes zu $\mathcal{S}$ hinzunimmt.

Die Einschränkung an p_j aus der Nebenbedingung $P \in \mathcal{G}_\mathcal{S}$ wird durch die Indexmenge $\mathcal{S}(j) := \{i : \ (i,j) \in \mathcal{S}\}$ ausgedrückt. Zu einem Vektor x oder einer Matrix A und einer Indexmenge $J \subseteq \{1,\ldots,n\}$ bezeichnet x_J bzw. A_J denjenigen Teil, der nur die Komponenten bzw. Spalten mit Indizes aus J enthält (vgl. Lineare Optimierung). Damit ist für die Spalte $x = p_j$ das Minimum des Problems $\min_x \|A_J x_J - e_j\|_2$ zu bestimmen. Dies ist ein Kleinste-Quadrate-Problem zum überbestimmten System

$$A_{\mathcal{S}(j)} x_{\mathcal{S}(j)} = Q \begin{pmatrix} R \\ 0 \end{pmatrix} x_{\mathcal{S}(j)} = Q_1 R x_{\mathcal{S}(j)} = e_j, \tag{5.7.3}$$

das man mit einer *QR-Zerlegung* von $A_{\mathcal{S}(j)}$ effizient lösen kann, wenn die Zahl $k := |\mathcal{S}(j)|$ der Indizes klein ist. Insbesondere kann man auch verschwindende *Zeilen* aus $A_{\mathcal{S}(j)}$ streichen. In (5.7.3) bezeichnet $Q = (Q_1, Q_2) \in \mathbb{R}^{n\times n}$ den unitären Faktor und $Q_1 \in \mathbb{R}^{n\times k}$ seine ersten k Spalten, die eine orthogonale Matrix $Q_1^\mathsf{T} Q_1 = I_k$ bilden. Die Kleinste-Quadrate-Lösung von (5.7.3) ist damit $x_{\mathcal{S}(j)} = R^{-1} Q_1^\mathsf{T} e_j$ bei vollem Rang der Teilmatrix $A_{\mathcal{S}(j)}$. Sehr kleine Elemente dieser Lösung $x_{\mathcal{S}(j)}$ sollte man wieder entfernen und so $\mathcal{S}$ verkleinern. Schwieriger ist aber die Frage, welche Indizes man sinnvollerweise zu $\mathcal{S}$ hinzunimmt, wenn das Residuum $r := A_{\mathcal{S}(j)} x_{\mathcal{S}(j)} - e_j$ zu groß ist. Dazu kann man für $r \neq 0$ zunächst einen einzelnen Index $\ell \notin J := \mathcal{S}(j)$ betrachten und das Minimum von

$$\|e_j - A_{J\cup\{\ell\}} x_{J\cup\{\ell\}}\|_2^2 = \|r - x_\ell A e_\ell\|_2^2 = \|r\|_2^2 - 2x_\ell r^\mathsf{T} A e_\ell + x_\ell^2 \|Ae_\ell\|_2^2$$

bezüglich x_ℓ untersuchen. Dieses wird in $x_\ell = r^\mathsf{T} Ae_\ell / \|Ae_\ell\|_2^2$ angenommen, interessanter ist aber die neue Residuumsnorm

$$\rho_\ell^2 = \|r\|_2^2 - \left(\frac{r^\mathsf{T} Ae_\ell}{\|Ae_\ell\|_2}\right)^2.$$

Man überlegt sich leicht, dass $r^\mathsf{T} Ae_\ell \neq 0$ für mindestens ein $\ell \notin \mathcal{S}(j)$ gilt und dort also $\rho_\ell < \|r\|_2$ ist. Dann erweitert man $\mathcal{S}$ um diejenigen Paare (ℓ, j), für die

$$|r^\mathsf{T} Ae_\ell| > \delta \|Ae_\ell\|_2 > 0$$

mit einer geeigneten Schwelle δ. Details dazu werden nicht besprochen. Wichtiger ist, dass im nächsten Schritt wieder das Kleinste-Quadrate-Problem (5.7.3) zu lösen ist mit einer vergrößerten Indexmenge $J \cup J'$, $J = \mathcal{S}(j)$. Insbesondere, wenn J' nur wenige zusätzliche Indizes enthält, ist es sehr wichtig, die alte QR-Zerlegung wiederzuverwenden. Dazu sei L die Menge aller Zeilenindizes mit nichttrivialen Elementen von A_J, mit $A_J^{(L)}$ sei dann dieser nichttriviale Abschnitt von A bezeichnet. Genauso sei $L \cup L'$ die entsprechende Menge für die erweiterte Matrix $A_{J\cup J'}$. Dann ist aus (5.7.3) die QR-Zerlegung von $A_J^{(L)}$ bekannt und es gilt

$$\begin{aligned} A_{J\cup J'}^{(L\cup L')} &= \begin{pmatrix} A_J^{(L)} & A_{J'}^{(L)} \\ 0 & A_{J'}^{(L')} \end{pmatrix} = \begin{pmatrix} Q & \\ & I \end{pmatrix} \begin{pmatrix} R & Q_1^\mathsf{T} A_{J'}^{(L)} \\ 0 & Q_2^\mathsf{T} A_{J'}^{(L)} \\ 0 & A_{J'}^{(L')} \end{pmatrix} \\ &= \begin{pmatrix} Q & \\ & I \end{pmatrix} \begin{pmatrix} I & \\ & U \end{pmatrix} \begin{pmatrix} R & Q_1^\mathsf{T} A_{J'}^{(L)} \\ 0 & R' \\ 0 & 0 \end{pmatrix}, \end{aligned} \tag{5.7.4}$$

wenn man eine QR-Zerlegung des südöstlichen Blocks

$$\begin{pmatrix} Q_2^\mathsf{T} A_{J'}^{(L)} \\ A_{J'}^{(L')} \end{pmatrix} = U \begin{pmatrix} R' \\ 0 \end{pmatrix}, \quad U = (U_1, U_2)$$

berechnet. Man beachte, dass bei den Blockdiagonalmatrizen in (5.7.4) unterschiedliche Blockgrößen auftreten, die Blöcke Q und U überlappen sich im Produkt, welches eine unitäre Matrix ist. Daher ist (5.7.4) tatsächlich eine QR-Zerlegung. Mit einer vorgegebenen Toleranz $\varepsilon > 0$ kann man mit diesem Verfahren eine Approximativ-Inverse P bestimmen mit Spalten $p_j = Pe_j$ und Residuen

$$\|r_j\|_2 < \varepsilon, \quad r_j := Ap_j - e_j,\ j = 1, \ldots, n.$$

Für genügend kleines ε häufen sich dann die Eigenwerte der präkonditionierten Matrix AP tatsächlich bei eins und erzeugt bei den besprochenen Iterationsverfahren schnelle Konvergenz.

Satz 5.7.1 *Es sei q die Maximalzahl nichttrivialer Elemente in r_j, $j = 1, \ldots, n$, und es gelte $\varepsilon\sqrt{q} < 1$. Dann häufen sich die Eigenwerte von AP bei Eins und liegen in einem Kreis mit Radius $\varepsilon\sqrt{q}$. Außerdem gilt*

$$\hat{\kappa}(AP) \le \frac{1 + \varepsilon\sqrt{q}}{1 - \varepsilon\sqrt{q}}.$$

Beweis Auch die Frobeniusnorm ist invariant unter unitären Transformationen. Es sei $U(\Lambda + R)U^\mathsf{T} = AP$ die Schur-Normalform von AP mit der Eigenwertmatrix $\Lambda = \operatorname{diag}(\lambda_j)$. Dann gilt

$$\sum_{j=1}^{n} |\lambda_j - 1|^2 = \|\Lambda - I\|_F^2 \le \|\Lambda + R - I\|_F^2 = \|AP - I\|_F^2 = \sum_{j=1}^{n} \|r_j\|_2^2 \le n\varepsilon^2.$$

Daher ist die mittlere Abweichung der Eigenwerte von 1 (Varianz) beschränkt durch ε. Das Residuum r_j hat n.V. nur q nichttriviale Elemente. Daher gilt $\|r_j\|_1 \le \|r_j\|_2\sqrt{q} \le \varepsilon\sqrt{q}$ und damit $\|AP - I\|_1 = \max_j \|r_j\|_1 \le \varepsilon\sqrt{q}$. Wenn $x \ne 0$ ein Eigenvektor von AP ist zum Eigenwert λ, dann gilt

$$|\lambda - 1|\|x\|_1 = \|(\lambda - 1)x\|_1 = \|(AP - I)x\|_1 \le \|AP - I\|_1\|x\|_1 \le \varepsilon\sqrt{q}\,\|x\|_1.$$

Damit folgen auch die Schranken $1 - \varepsilon\sqrt{q} \leq |\lambda| \leq 1 + \varepsilon\sqrt{q}$ und so die Schranke für die Kondition $\hat{\kappa}$. ■

Bemerkungen a) Man beachte, dass für die Funktion des Algorithmus außer der dünnen Besetzung keine weiteren Voraussetzungen wie etwa Diagonaldominanz gemacht wurden im Gegensatz zu den anderen Präkonditionierern.

b) Bei einer symmetrischen Matrix A liefert das Verfahren i.d.R. kein symmetrisches P. Zur Präkonditionierung sollte man dann den symmetrischen Anteil $\frac{1}{2}(P + P^{\mathsf{T}})$ verwenden.

c) Man kann auch Links- und Rechts-Präkonditionierung kombinieren, indem man etwa die Inversen der LR-Zerlegung direkt dünn approximert. Die präkonditionierte Matrix ist dann $L'AR'$, wobei man mit der Wahl $R' = L'$ den symmetrischen Fall direkt berücksichtigen kann. Zur Lastverteilung bei der Multiplikation mit Dreieckmatrizen sei dabei an §3.2.2 erinnert.

Literatur:

M.J. Grote, T.Huckle, *Parallel preconditioning with sparse approximate inverses*, SIAM J. Sci. Comp. 18, 838-853 (1997),

M. Bycklin, M. Huhtanen, *Approximate factoring of the inverse*, Numer. Math. 117, 507-528 (2011).

6 Parallel-Verfahren für gewöhnliche Differentialgleichungen

Bei einem gewöhnlichen Anfangswertproblem (AWP) ist eine Lösungskurve $t \mapsto u(t) \in \mathbb{R}^n$ gesucht für das Problem

$$u'(t) = f\big(t, u(t)\big),\ t \in [t_0, t_e], \tag{6.0.1}$$

$$u(t_0) = u_0, \tag{6.0.2}$$

mit einer zumindestens stetigen Funktion $f : \mathbb{R}^{n+1} \to \mathbb{R}^n$. Die Lösung u beginnt also zum "Zeitpunkt" t_0 im Punkt $u_0 \in \mathbb{R}^n$ und ihre Tangentenrichtung u' zeigt zu jedem $t \in [t_0, t_e]$ genau in die Richtung, die durch den Funktionswert $f(t, u(t))$ im aktuellen Kurvenpunkt $(t, u(t))$ vorgegeben ist. Numerische Verfahren zur Lösung solcher Anfangswertprobleme gehen meist schrittweise vor in Zeitschritten von Punkten t_m, $m \in \mathbb{N}_0$, zu $t_{m+1} = t_m + h_m$, die durch eine *Schrittweite* $h_m > 0$ voneinander getrennt sind. Dabei sind die klassischen Standardverfahren leider kaum parallelisierbar. Grundlegende Begriffe zur Analyse wurden aber bei ihnen entwickelt. Parallelität kann man bei Systemen von AWPen auf 2 Weisen betrachten:

1. Parallelität im System: Bei großen, regelmäßig aufgebauten rechten Seiten f kann man die Funktionsauswertung $f(t_m + .., y_m + ..)$ evtl. auf mehrere Prozessoren verteilen. Das gilt z.B. für das zeitabhängige Referenzbeispiel, die Wärmeleitungsgleichung $u_t = \Delta u + g$. Dieser Ansatz ist aber problemabhängig und vom Anwender zu implementieren.

2. Parallelität in der Methode: Das Integrationsverfahren für die DGl besitzt eine parallele Struktur, die keine speziellen Eigenschaften der rechten Seite f voraussetzt. Der Parallelisierungsgrad ist dabei zunächst überschaubar, eine Parallelität im System (Punkt 1) kann aber zusäzlich genutzt werden.

6.1 Sequentielle Standardverfahren

Ansatzpunkt der Standardverfahren ist die Umformulierung des Anfangswertproblems in die Integralgleichung aus dem Satz von Picard-Lindelöf. Allerdings betrachtet man dabei am besten schon einen der oben genannten Zeitschritte. Dort gilt für die Lösung u die Gleichung

$$u(t_{m+1}) = u(t_m) + h_m \int_0^1 f\big(t_m + h_m r, u(t_m + h_m r)\big)\, dr \tag{6.1.1}$$

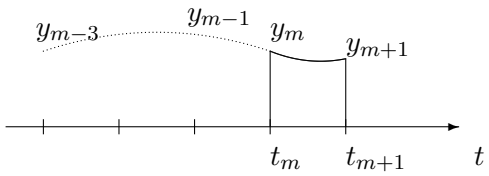
Das Integral stellt dabei einen Mittelwert von Funktionswerten f auf dem Intervall $[t_m, t_{m+1}]$ dar. Er hängt allerdings auch noch von der unbekannten Lösungskurve u ab. In einem ersten Schritt approximiert man dieses Integral durch eine *Quadraturformel*, d.h. man interpoliert den Integranden $f(t_m + .. , u(t_m + ..))$ durch ein Polynom (vgl. Numerik 1) und berechnet dessen Integralwert. Diese Quadratur-Näherug besteht aus einer Linearkombination von Funktionswerten, $\sum_{j=0}^{m} \beta_j g(r_j) \cong \int_0^1 g(r)\, dr$ in Stützstellen r_j. Die einfachste Variante ist in (6.1.1) die Ersetzung des Integrals durch den Funktionswert $f(t_m, u(t_m))$ am linken Intervallrand (Polynomgrad 0).

Dies führt auf das einfachste Näherungsverfahren (für Näherungen $y_m \cong u(t_m)$), das *explizite Euler-Verfahren*:

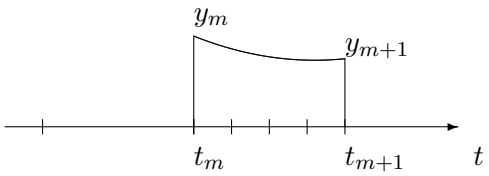
$$y_{m+1} = y_m + h_m f(t_m, y_m),\ m \geq 0, \quad y_0 := u_0. \tag{6.1.2}$$

Zur Herleitung besserer Verfahren, die höhere Polynomgrade und mehr Stützstellen verwenden, muß man in einem zweiten Schritt aber das Problem lösen, dass im Integranden die unbekannte Lösung u auftaucht. Hierfür gibt es zwei unterschiedliche Lösungsansätze, die auf verschiedene Verfahrensklassen führen.

Mehrschrittverfahren



Einschrittverfahren



1. *Mehrschrittverfahren:* Das Interpolationspolynom verwendet nur Funktionswerte auf dem Zeitgitter $t_0, \ldots, t_m, t_{m+1}$, also insbesondere solche links *außerhalb des Integrationsintervalls* $[t_m, t_{m+1}]$! Dies hat den Vorteil, dass bis auf den neuen Wert $u(t_{m+1})$ alle anderen in der Quadraturformel tatsächlich schon bekannt sind. Das Verfahren lautet mit der Abkürzung $f_k := f(t_k, y_k),\ k \geq 0$, und Gewichten $\beta_j \in \mathbb{R}$ einfach

$$y_{m+1} - y_m = h_m \sum_{j=0}^{s} \beta_j f_{m+1-j},\ m \geq s. \tag{6.1.3}$$

   Es werden also zusätzlich zu y_m noch weitere $s-1$ alte Näherungswerte aus mehreren Zeitschritten verwendet. Zu beachten ist hierbei, dass für $\beta_0 \neq 0$ der neue Lösungswert y_{m+1} auf der linken *und der rechten Seite* auftritt. In diesem Fall ist ein i.a. nichtlineares Gleichungssystem der Form $y_{m+1} - h_m f(t_{m+1}, y_{m+1}) = \ldots$ für y_{m+1} zu lösen, das Verfahren heißt dann *implizit*. Für $\beta_0 = 0$ ist das Verfahren *explizit*. Da man auf der rechten Seite des Verfahrens (6.1.3) eine Linearkombination alter (Funktions-) Werte verwendet, kann man dies auch auf der linken Seite tun. Man bekommt so die allgemeinen linearen Mehrschrittverfahren ($\alpha_0 = 1$)

$$\sum_{j=0}^{s} \alpha_j y_{m+1-j} = h_m \sum_{j=0}^{s} \beta_j f_{m+1-j},\ m \geq s.,\ m \geq s. \tag{6.1.4}$$

2. *Einschrittverfahren:* Hier verwendet man tatsächlich nur Werte aus dem Intervall $[t_m, t_{m+1}]$, indem man Zwischenknoten $r_i = t_m + h_m c_i$, $c_i \in [0,1]$, $i = 1, \ldots, s$ einführt. Mit der üblichen Abkürzung $k_i = f(t_m + h_m c_i, ..)$ bekommt man als Approximation für (6.1.1) mit Koeffizienten b_i den Schritt

$$y_{m+1} = y_m + h_m \sum_{i=1}^{s} b_i k_i. \tag{6.1.5}$$

Die grundlegende Schwierigkeit ist immer noch, dass man mit k_i die Lösungsableitung an einer Stelle $t_m + h_m c_i$ im neuen Intervall benötigt. Man löst das Problem dadurch, dass man schrittweise immer bessere Approximationen aufbaut, indem man mit Koeffizienten a_{ij} die einzelnen *Stufen* berechnet

$$k_i = f\Big(t_m + h_m c_i, y_m + h_m \sum_{j=1}^{s} a_{ij} k_j\Big),\ i = 1, \ldots, s. \tag{6.1.6}$$

Das Verfahren (6.1.5), (6.1.6) heißt *Runge-Kutta-Verfahren*. Wenn $A = (a_{ij})$ eine streng untere Dreieckmatrix ist, ist das Verfahren wieder *explizit*, da man die Stufen der Reihe nach berechnet, die erste Stufe $k_1 = f(t_m, y_m)$ stimmt hier immer mit dem Euler-Verfahren (6.1.2) überein.

Die Verfahrensklassen (6.1.3) und (6.1.5) kann man einheitlich beschreiben in der Form

$$y_{m+1} = y_m + h_m \Phi_{h_m}(t_m, y_m, \ldots). \tag{6.1.7}$$

Dabei nennt man Φ_h die Verfahrensfunktion, welche bei Einschrittverfahren nur von y_m, bei Mehrschrittverfahren aber auch von y_{m-1} etc. abhängt. Man will natürlich wissen, wie genau die numerischen Approximationen y_m im Vergleich zur Lösung u sind. Die Schrittweite(n) h ist die bestimmende Größe für die Genauigkeit. Daher erwartet man, dass die Näherung y gegen u konvergiert für feinere Schrittweiten $h \to 0$. Bei Differentialgleichung erfolgt dieser Nachweis prinzipiell über den Grundsatz

Aus Konsistenz und Stabilität folgt Konvergenz.

Konsistenz des Verfahrens bedeutet, dass beim Einsetzen der Lösung u in die Verfahrensvorschrift nur ein kleines Residuum T_h bleibt. Bei glatten Lösungen gilt dabei typischerweise

$$hT_h(t) := u(t_{m+1}) - u(t_m) - h\Phi_h(t_m, u(t_m), ..) = \mathcal{O}(h^{p+1}). \tag{6.1.8}$$

Man nennt T_h auch den *lokalen Fehler* (truncation error), der Exponent p heißt (Konsistenz-) *Ordnung* des Verfahrens. Bei Mehrschrittverfahren ist der Konsistenznachweis recht einfach, da in der Lösung u gilt $f(t_m, u(t_m)) = u'(t_m)$. Daher bekommt man die Konsistenzordnung einfach aus der Taylorentwicklung des Residuums

$$hT_h(t_m) = \sum_{j=0}^{s} \alpha_j u(t_{m+1-j}) - h_m \sum_{j=0}^{s} \beta_j u'(t_{m+1-j}).$$

Das 4-Schrittverfahren von Adams-Bashforth für konstante Schrittweite

$$y_{m+1} - y_m = \frac{h}{24}\big(55 f_m - 59 f_{m-1} + 37 f_{m-2} - 9 f_{m-3}\big),$$

z.B., besitzt Ordnung 4, denn $T_h = ch^4 u^{(5)}(\tau)$ mit τ bei t_m.

Der Konsistenznachweis für Runge-Kutta-Verfahren ist schwieriger, denn man muß den Lösungswert $u(t_{m+})$ mit derjenigen Näherung $y_{m+1} = u(t_m) + h_m \sum_{i=1}^{s} b_i k_i$ vergleichen, welche mit dem exakten Anfangswert $u(t_m)$ auch in $k_i = f(t_m + h_m c_i, u(t_m) + \ldots)$ bestimmt wurde. Die Rechnungen sind aufwändig, da verschiedene Taylorentwicklungen ineinander eingesetzt werden müssen und darin insbesondere Ableitungen von f auftreten. Das bekannteste Verfahrensbeispiel ist das folgende 4-stufige Verfahren der Ordnung $p = 4$, das durch sein *Butcher-Tableau* beschrieben wird

$$\begin{array}{c|c} c & A \\ \hline & b^{\mathsf{T}} \end{array} = \begin{array}{c|cccc} 0 & 0 & & & \\ \frac{1}{2} & \frac{1}{2} & 0 & & \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \qquad \text{mit } hT_h = O(h^5).$$

Stabilität bedeutet, dass kleine Störungen der Verfahrensfunktion sich im Verlauf der Rechnungen nur überschaubar (Lipschitz-stetig) auswirken. Man benötigt für die gestörte Lösung

$$z_{m+1} = z_m + h_m \Phi_{h_m}(t_m, z_m, \ldots, z_{m-s}) + h_m g_m$$

dazu im Vergleich zu (6.1.7) eine Abschätzung der Form

$$\|z_m - y_m\| \le K\Big(\|z_0 - y_0\| + \max_{i \le m} \|g_m\|\Big), \quad t_0 \le t_m \le t_e. \tag{6.1.9}$$

Dieser Nachweis ist bei Adams- und Runge-Kutta-Verfahren für Lipschitz-stetige Funktionen f recht einfach, für das allgemeine Mehrschrittverfahren (6.1.4) aber eine gravierende Einschränkung. Man beachte, dass in der Schranke (6.1.9) bei g_m der Vorfaktor h_m verloren ging. Dies ist der Grund, warum man in der Konsistenz-Definition (6.1.8) eine h-Potenz abgetrennt wurde.

Konvergenz folgt mit den beiden Begriffen direkt. Denn die Konsistenzdefinition bedeutet gerade, dass obige Gleichung für $z_m := u(t_m)$ gilt mit $g_m = T_h(t_m) = \mathcal{O}(h^p)$ (ohne Vorfaktor h!). Und die Stabilität zeigt folgende Fehlerschranke für den *globalen Fehler*

$$\|u(t_m) - y_m\| \le K \max_{i \le m} \|T_h(t_i)\| = \mathcal{O}(h^p).$$

Für stabile Verfahren ist daher insbesondere die Konvergenzordnung mindestens so hoch wie die Konsistenzordnung. Eine hohe Konvergenzordnung p hat den Vorteil, dass eine Halbierung der Schrittweite $h \to h/2$ den Fehler stärker verkleinert $h^p \to 2^{-p} h^p$ als bei einer niederen Ordnung, während sich der Rechenaufwand unabhängig von der Ordnung verdoppelt. In der Praxis wird die Schrittweite h_m durch Schrittweitensteuerung an den aktuellen Verlauf der Lösung angepaßt. Dies ist bei Einschrittverfahren problemlos möglich, bei Mehrschrittverfahren aber mit einigem Aufwand und Vorsichtsmaßnahmen verbunden. Zur Anpassung der Schrittweite verwendet man Fehlerschätzungen, die man mit den berechneten Daten y_{m+1-j} bzw. k_j berechnet.

An den Verfahrensvorschriften (6.1.3) und (6.1.6) erkennt man sofort, dass die expliziten klassischen Verfahren kaum einen Ansatzpunkt für parallele Berechnung bieten. Bei den Mehrschrittverfahren müssen die y_m der Reihe nach berechnet werden und auch in den Stufengleichungen von expliziten Runge-Kutta-Verfahren bestehen direkte Abhängigkeiten. Man hat zwar versucht, bei RK-Verfahren durch Lücken in der Matrix A etwas Unabhängigkeit der Stufen und damit Parallelität einzubauen. Man gewinnt dadurch aber wenig, da dann mehr Stufen für die gleiche Ordnung benötigt werden. Alternative Ansätze bei allen Klassen waren parallelisierbare Iterationsverfahren zur Lösung der auftretenden Gleichungen mit mäßigem Erfolg.

6.2 Peer-Zweischritt-Methoden

Zur Konstruktion von parallelisierbaren Verfahren sollte man die genannten Verfahrensklassen verlassen und Verfahren betrachten, die sowohl mehrere *Stufen* als auch mehrere *Zeitschritte* verwenden. Solche Verfahren lassen sich in die übergreifende Klasse der *Allgemeinen Linearen Verfahren* (GLM=General Linear Methods) von Butcher einordnen. Hier gibt es verschiedene Ansätze. Sehr einfach aufgebaut ist die Klasse der *Peer-Methoden*. Diese benutzen wie Runge-Kutta-Verfahren pro Zeitschritt s Stufen, die Stufenlösungen $Y_{m,i} \cong y(t_{m,i})$ in Zwischenpunkten $t_{m,i} = t_m + h_m c_i$, $i = 1, \ldots, s$, werden aber nach einem Schritt nicht verworfen, sondern beim Zeitschritt im nächsten Intervall verwendet. Dafür gibt es keinen ausgezeichneten Lösungswert y_{m+1}, alle Stufen $Y_{m,i}$ sind gleich gute Approximationen der Lösung ($\rightarrow$ "peer"), allerdings an unterschiedlichen Stellen im Teilintervall. Ein Zeitschritt besteht aus den Gleichungen

$$Y_{m,i} - h_m \gamma_i f(t_{m,i}, Y_{m,i}) = \sum_{j=1}^{s} b_{ij} Y_{m-1,j} + h_m \sum_{j=1}^{s} a_{ij} f\big(t_{m-1,j}, Y_{m-1,j}\big), \quad i \in \{1, \ldots, s\}, \tag{6.2.1}$$

wobei γ_i, a_{ij}, b_{ij} Verfahrenskoeffizienten sind. An dieser Schreibweise kann man einen wesentlichen Aspekt sofort sehen. Auf der linken Seite stehen die *voneinander unabhängigen* Gleichungen für die neuen Stufenlösungen $Y_{m,i}$, während alle Daten auf der rechten Seite aus dem *vorherigen Zeitschritt* stammen. Daher sind alle s Stufen *parallel* ausführbar, das Verfahren ist aber ein Zweischrittverfahren, das Daten aus 2 Zeitintervallen verknüpft. Daher werden viele Ergebnisse von dem Schrittweitenverhältnis

$$\sigma_m = \frac{h_m}{h_{m-1}}$$

abhängen (aber nur von einem σ_m, nicht von mehreren wie bei MSVen). In (6.2.1) wurde auch die implizite Version aufgenommen. Für $\gamma_i > 0$ sind s voneinander unabhängige Gleichungssysteme zu lösen, im expliziten Fall ($\gamma_i \equiv 0$) ist $Y_{m,i}$ direkt durch die rechte Seite bestimmt. Implizite Verfahren werden für einen schwierigen Typ von DGln benötigt (s.u.), hier verursacht die Auflösung der $n \times n$-Systeme in den Stufen den Hauptaufwand, welche aber auf $\geq s$ Prozessoren parallel ablaufen kann.

Für eine übersichtlichere Darstellung kann man die Stufen und die Ableitungen in flachen Matrizen $Y_m = (Y_{m,1}, \ldots, Y_{m,s})^\mathsf{T}$ und F_m unterbringen und die Koeffizienten in Matrixform

sammeln $B := (b_{ij})$, $A := (a_{ij})$, $\Gamma := \mathrm{diag}(\gamma_i)$. Dann lautet der Verfahrensschritt (6.2.1) einfach

$$\begin{aligned} Y_m - h_m\Gamma_m F_m = &\quad B_m Y_{m-1} + h_m A_m F_{m-1} \\ = &\quad \Big(B_m\ ,\ A_m\Big)\begin{pmatrix} Y_m \\ h_m F_m \end{pmatrix}, \qquad \text{mit } F_m = \begin{pmatrix} f(t_{m,1}, Y_{m,1})^{\mathsf{T}} \\ \vdots \\ f(t_{m,s}, Y_{m,s})^{\mathsf{T}} \end{pmatrix}. \end{aligned} \tag{6.2.2}$$

Die rechte Seit besteht also aus der parallelen Auswertung aller f-Werte und der anschließenden Multiplikation mit der Matrix (B_m, A_m). Der Index m an den Matrizen deutet an, dass diese mglw. vom aktuellen Zeitschritt abhängen, was bei wechselndem σ_m erforderlich ist.

6.2.1 Ordnung

Die *Konsistenz* der Peer-Methoden überprüft man genauso einfach wie bei Mehrschrittverfahren, indem man alle Stufenwerte durch die Werte der Lösung u ersetzt. Die Auswertung ist leicht, da $f(t,u)$ in der Lösung gerade u' ist. Für die einzelnen Stufen bekommt man daher durch Taylor-Entwicklung um t_{m-1} mit $t_{m,i} = t_{m-1} + h_{m-1} + h_m c_i = t_{m-1} + h_{m-1}(1+\sigma_m c_i)$ die Gestalt

$$\begin{aligned} h_m T_{m,i} =& u(t_{m,i}) - h_m\gamma_i u'(t_{m,i}) - \sum_{j=1}^{s} b_{ij} u(t_{m-1,j}) - h_m \sum_{j=1}^{s} a_{ij} u'(t_{m-1,j}) \\ =& \sum_{k=0} h_{m-1}^k \frac{u^{(k)}}{k!}\big((1+\sigma_m c_i)^k - \sum_{j=1}^{s} b_{ij} c_j^k\big) - \sum_{k=0} h_{m-1}^{k+1}\frac{u^{(k+1)}}{k!}\big(\gamma_i(1+\sigma_m c_i)^k + \sum_{j=1}^{s} a_{ij} c_j^k\big) \\ =& \sum_{k=0} h_{m-1}^k \frac{u^{(k)}}{k!}\Big((1+\sigma_m c_i)^k - \sigma_m\gamma_i k(1+\sigma_m c_i)^{k-1} - \sum_{j=1}^{s}(b_{ij}c_j^k + k\sigma_m a_{ij} c_j^{k-1})\Big). \end{aligned}$$

Da dieser Ausdruck für beliebige Lösungen eine bestimmte Ordnung besitzen soll, müssen die einzelnen Klammern verschwinden. Die Ordnungsbedingungen werden mit $AB(1), AB(2), \ldots$ bezeichnet und lauten

$$AB(k+1): \quad (1+\sigma_m c_i)^k - \sigma_m\gamma_i k(1+\sigma_m c_i)^{k-1} - \sum_{j=1}^{s}(b_{ij}c_j^k + k\sigma_m a_{ij} c_j^{k-1}) = 0, \tag{6.2.3}$$

$i = 1, \ldots, s$. Die Bedingung $AB(1)$ zum Polynomgrad null heißt *Präkonsistenz*, ist sehr einfach,

$$\sum_{j=1}^{s} b_{ij} = 1 \quad \Longleftrightarrow \quad B\mathbb{1} = \mathbb{1},$$

und bedeutet, dass jeder konstante Vektor ein Eigenvektor von B zum Eigenwert eins ist. Ordnungsbedingungen benutzt man nicht nur, um die Ordnung eines gegebenen Verfahrens zu überprüfen, sondern vor allem dazu, diese Verfahren überhaupt zu konstruieren. Dazu schreibt man die Bedingungen zunächst spaltenweise mit Vektoren $\mathbf{c} = (c_i)_{i=1}^s$ und der Rechenregel $\mathbf{c}^k := (c_i^k)$ und bekommt für eine Serie von Bedingungen (6.2.3) die Darstellung

$$\begin{aligned} AB(1\ldots k+1): \quad 0 =& \big(\mathbb{1}, \mathbb{1}+\sigma_m\mathbf{c}, \ldots, (\mathbb{1}+\sigma_m\mathbf{c})^k\big) - \sigma_m\Gamma\big(0, \mathbb{1}, \ldots, k(\mathbb{1}+\sigma_m\mathbf{c})^{k-1}\big) \\ & - B\big(\mathbb{1}, \mathbf{c}, \ldots, \mathbf{c}^k\big) - \sigma_m A\big(0, \mathbb{1}, \ldots, k\mathbf{c}^{k-1}\big) \end{aligned} \tag{6.2.4}$$

Offensichtlich sind dies gut überschaubare Beziehungen zwischen den Matrizen A, B, Γ, insbesondere wenn deren rechte Faktoren quadratische Matrizen sind, etwa bei $AB(1\dots s)$. Dazu werden zunächst die Matrizen

$$V := \left(c_i^{j-1}\right)_{i,j=1}^{s} = \begin{pmatrix} 1 & c_1 & \dots & c_1^{s-1} \\ 1 & c_2 & \dots & c_2^{s-1} \\ \vdots & \vdots & & \vdots \\ 1 & c_s & \dots & c_s^{s-1} \end{pmatrix}, \quad P := \left(\binom{j-1}{i-1}\right)_{i,j=1}^{s} = \begin{pmatrix} 1 & 1 & 1 & \dots \\ & 1 & 2 & \dots \\ & & 1 & \dots \\ & & & \ddots \end{pmatrix} \tag{6.2.5}$$

eingeführt. Die linke ist die Vandermonde-Matrix $V = (1, \mathbf{c}, \dots, \mathbf{c}^{s-1})$ aus der Polynominterpolation und die rechte die Pascal-Matrix der Binomialkoeffizienten. Zusammen mit $S_m := \operatorname{diag}(1, \sigma_m, \dots, \sigma_m^{s-1})$ hat dann die verschobene Vandermonde-Matrix die Darstellung

$$\left(\mathbb{1}, \mathbb{1} + \sigma_m \mathbf{c}, \dots, (\mathbb{1} + \sigma_m \mathbf{c})^{s-1}\right) = \left((1 + \sigma c_i)^{j-1}\right)_{i,j=1}^{s} = VSP$$

aufgrund des binomischen Satzes. Da in den Ordnungsbedingungen Funktions- und Ableitungswerte miteinander verknüpft werden, treten in (6.2.4) auch verschobene Versionen dieser Matrizen auf und zusätzliche Vorfaktoren. Dies kann man mit den Matrizen

$$F_0 := \begin{pmatrix} 0 & & & \\ 1 & 0 & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{pmatrix} = \left(\delta_{i,j+1}\right)_{i,j=1}^{s}, \quad D := \operatorname{diag}(1, 2, \dots, s),$$

beschreiben. Im Produkt VF_0^{T} sind die Spalten von V um eins nach rechts verschoben. Da die Präkonsistenz $AB(1)$ sehr einfach ist, kann man sie auch vorziehen und dann $AB(2\dots s+1)$ zusammenfassen. Dazu benötigt man noch die Diagonalmatrix der Stützstellen $C = \operatorname{diag}(c_i)$. So lassen sich die Ordnungsbedingungen als Matrixgleichungen zusammengefassen.

Satz 6.2.1 *Wenn die Koeffizienten der Peer-Methode* (6.2.1) *die Bedingungen*

$$\begin{aligned} AB(1\dots s): &\quad VSP - \sigma_m \Gamma VSPDF_0^{\mathsf{T}} - BV - \sigma_m AVDF_0^{\mathsf{T}} = 0, \quad \textit{bzw.} \\ AB(1\dots s+1): &\quad B\mathbb{1} = \mathbb{1}, \quad (I + \sigma_m C)VSP - \sigma_m \Gamma VSPD - BCV - \sigma_m AVD = 0, \end{aligned}$$

erfüllen, dann besitzt das Verfahren die Konsistenzordnung $s-1$ bzw. s. Für genügend glatte Lösungen u gilt für den lokalen Fehler dann $T_h = \mathcal{O}(h_{m-1}^{s-1})$ bzw. $T_h = \mathcal{O}(h_{m-1}^{s})$.

Man erkennt an diesen Bedingungen, dass die Koeffizientenmatrizen meistens von rechts mit der Vandermonde-Matrix V multipliziert werden. Daher ist es naheliegend, entsprechend transformierte Versionen aller Matrizen zu betrachten, wie etwa bei

$$\tilde{B} := V^{-1}BV \iff B = V\tilde{B}V^{-1}.$$

Da die erste Spalte von V der Einservektor ist, $Ve_1 = \mathbb{1}$ vereinfacht sich schon die Präkonsistenz zu $\tilde{B}e_1 = e_1$. Für den Umgang mit dem Produkt CV wird ein weiterer Hilfssatz benötigt.

Satz 6.2.2 *Es sei* $\varphi(t) := (t-c_1)\cdots(t-c_s) = \sum_{j=0}^{s} \phi_j t^j$ *das Stützstellen-Polynom. Dann gilt*

$$CV = VF \quad mit \quad F := \begin{pmatrix} 0 & & & -\phi_0 \\ 1 & 0 & & -\phi_1 \\ & \ddots & \ddots & \\ & & 1 & -\phi_{s-1} \end{pmatrix} = F_0 - \phi e_s^{\mathsf{T}}, \quad \phi := \begin{pmatrix} \phi_0 \\ \phi_1 \\ \vdots \\ \phi_{s-1} \end{pmatrix}.$$

Beweis Die ersten $s-1$ Spalten von CV erhält man durch Linksverschiebung, welche durch die ersten $s-1$ Spalten von F bewirkt werden. Für jede Stützstelle c_i gilt nach Definition $c_i^s = -\sum_{j=0}^{s-1} c_i^j \phi_j \iff CVe_s = \mathbf{c}^s = -\sum_{j=0}^{s-1} \mathbf{c}^j \phi_j = -V\phi$. ∎

Die Matrix F ist die *Frobeniusmatrix* (Begleitmatrix) des Polynoms φ, ihre Eigenwerte sind die Stützstellen, da $F = V^{-1}CV$ gilt. Damit bekommt man für die Peer-Methoden die transformierten Ordnungsbedingungen

$$AB(1\ldots s): \quad S_m P - \sigma_m \tilde{\Gamma} S_m P D F_0^{\mathsf{T}} - \tilde{B} - \sigma_m \tilde{A} D F_0^{\mathsf{T}} = 0, \quad \text{bzw.} \tag{6.2.6}$$

$$AB(1\ldots s+1): \quad \tilde{B}e_1 = e_1, \quad (I + \sigma_m F) S_m P - \sigma_m \tilde{\Gamma} S_m P D - \tilde{B} F - \sigma_m \tilde{A} D = 0. \tag{6.2.7}$$

Durch beide Varianten werden die drei Koeffizientenmatrizen miteinander gekoppelt. Eine der Matrizen ist also dann festgelegt, wenn man die beiden anderen gewählt hat. Welche man dabei am Besten als erste festlegt, hängt von anderen Verfahrensaspekten ab.

6.2.2 Stabilität

Statt eine allgemeine, aber möglicherweise pessimistische Stabilitäts-Schranke der Form (6.1.9) nachzuweisen, betrachtet man das Verhalten von Integrationsverfahren auch gerne anhand von Testgleichungen. Die einfachste ist die skalare lineare $u' = \lambda u$ mit $\lambda \in \mathbb{C}$. Deren Lösungen sind $u(t) = u_0 e^{\lambda t}$ und sie gehen gegen null für $t \to 0$, wenn $Re\lambda < 0$ ist. Wendet man die Peer-Methode auf diese Dgl an, sieht man in der Gestalt (6.2.2) den einfachen Zusammenhang $Y_m = M_m(z) Y_{m-1}$ mit $z := h_m \lambda$. Der Zeitschritt reduziert sich auf eine einfache Multiplikation mit der *Stabilitäts-Matrix*

$$M(z) := (I - z\Gamma_m)^{-1}(B_m + zA_m). \tag{6.2.8}$$

Vor dem Hintergrund der Dämpfungseigenschaften der Exponentialfunktion interessiert man sich insbesondere für die Punkte z der komplexen Ebene, für die auch die Näherungslösungen gegen null gehen $M(z)^m \to 0\,(m \to \infty)$ für konstante Schrittweiten $h_m \equiv h$.

Definition 6.2.3 *Für eine Peer-Methode ist der* Stabilitätsbereich S *definiert durch*

$$S := \{z \in \mathbb{C} : \ \varrho\big(M(z)\big) < 1\}.$$

Der einfachste Fall der Testgleichung ist $\lambda = 0$ und $y' = 0$ hat nur konstante, also global beschränkte Lösungen. Analog muß man für das Verfahren die sogenannte *Nullstabilität* fordern,

welche besagt, dass die Lösungen für $z = 0$ gleichmäßig beschränkt sind, dies allerdings für allgemeine Gitter mit varriierendem σ_m. Wegen $M_m(0) = B_m$ bedeutet das die Existenz einer Konstante $K > 0$ so, dass

$$\|B_m B_{m-1} \cdots B_\ell\| \leq K \ \forall m \geq \ell \geq 0. \tag{6.2.9}$$

Leider gibt es noch keine Theorie, die die Beschränkheit solcher Produkte bei allgemeinen Matrix-*Familien* $\{B_\ell : \ell \in J\}$ sicherstellt. Garantiert werden kann dies für 2 einfache Fälle

$$\begin{array}{ll} a) & B_m \equiv B, \quad \lambda_j(B) < 1,\ j = 2, \ldots, s, \\ b) & |\tilde{B}_m| \leq \hat{B}, \quad \lambda_j(\hat{B}) < 1,\ j = 2, \ldots, s. \end{array} \tag{6.2.10}$$

In diesen beiden Fällen sind jeweils die Potenzen B^m bzw. $\hat{B}^m$ gleichmäßig beschränkt fur allen $m \in \mathbb{N}$. Der Fall (6.2.10a) einer konstanten Koeffizientenmatrix B ist offensichtlich der einfachere. Vor dem Hintergund der Ordnungsbedingungen (6.2.7) bedeutet er aber, dass zuerst B festzulegen und dann die Gleichung nach $\tilde{A}$ aufzulösen ist.

6.2.3 Explizite Peer-Methoden

Für explizite Verfahren ist $\Gamma = 0$ und nach Wahl der festen Matrix B kann man Verfahren der Ordnung s durch Auflösen der Gleichung (6.2.7) nach $\tilde{A}$ konstruieren, d.h. mit

$$\sigma\tilde{A} = ((I + \sigma_m F)SP - \tilde{B}F)D^{-1}.$$

Als Verfahrensparameter sind die Knoten c_j und die Elemente von $\tilde{B}$ frei wählbar. Zur Auswahl geeigneter Verfahren nutzt man daher u.a. folgende Anforderungen:

+ alle Eigenwerte B, außer dem zu $B\mathbb{1} = \mathbb{1}$, sind null ("optimale Nullstabilität")
+ der Stabilitätsbereich S ist möglichst groß
+ die Norm von B ist möglichst klein (aber ≥ 1)
+ der erste, nicht verschwindende Koeffizient in der Taylorentwicklung von T_h ist möglichst klein.

Durch eine automatisierte Suche mit diesen Kriterien wurde z.B. die folgende 4-stufige Methode `epp4y3` gefunden mit Knoten $\mathbf{c} \doteq (1.3388, 1.7038, 1.8682, 1)^\mathsf{T}$ und

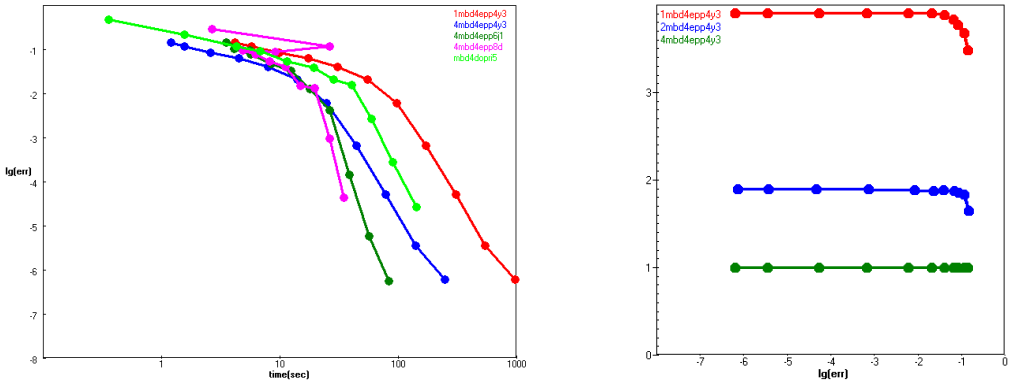
$$B = \begin{pmatrix} 0.0053 & -0.1279 & -0.0135 & 1.1360 \\ 0.0029 & -0.0027 & -0.0718 & 1.0716 \\ 0.0003 & -0.0045 & -0.0026 & 1.0073 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Man erkennt, dass die Knoten c_i auch außerhalb des Intervalls $[0, 1]$ liegen können. Bei numerischen Tests zeigte sich leider, dass zwar die s Funktionsauswertungen im Verfahrensschritt (6.2.2) $Y_m = BY_{m-1} + hAF(Y_{m-1})$ gut parallel ablaufen, der Parallelisierungsgrad bei der Matrixmultiplikation von (B, A) und (Y, hF) aber in der Praxis bei 2 stagniert. Daher wird der theoretische

Speed-up s nur bei sehr aufwändigen Problemen erreicht. Ein solches ist das Vielkörperproblem (Mini-Galaxis) für die Positionen $u_i(t) \in \mathbb{R}^3$ von $m = 400$ Objekten,

$$u_i'' = -\sum_{j \neq i} \frac{m_j(u_j - u_i)}{(\varepsilon + \|u_j - u_i\|_2^2)^{3/2}}, \ i = 1, \ldots, m, \tag{6.2.11}$$

da hier $O(m^2)$ Kräfte zu berechnen sind. Die folgenden Diagramme zeigen die Rechenzeiten verschiedener Verfahren auf 1 bis 4 Prozessoren für eine ganze Serie von Genauigkeitsvorgaben $tol = 10^{-3} \ldots 10^{-12}$. Es wird aber der tatsächlich erreichte Fehler gezeigt.



Im linken Diagramm gehören die rote und die blaue Kurve zum Verfahren `epp4y`, gerechnet mit einem bzw. vier Prozessoren. Dass deren Abstand einem Speed-up nahe bei vier entspricht, entnimmt man der rechten Grafik, die den fast optimalen Speed-up bei Rechnung auf einem, zwei und vier Prozessoren zeigt.

6.2.4 Implizite Peer-Methoden

Bei impliziten Methoden mit $\Gamma \neq 0$ sind in jeder Stufe von (6.2.1) nichtlineare Gleichungssysteme für die Stufenlösung Y_{mi} zu lösen. Dieser, im Vergleich zu expliziten Verfahren wesentlich höhere Aufwand, lohnt sich bei sogenannten steifen Anfangswertproblemen.

Beispiel 6.2.4 Das Anfangswertproblem

$$u'(t) = \lambda(1 - u(t)), \ u(0) = 0,$$

besitzt die Lösung $u(t) = 1 - e^{-\lambda t}$. Für $\lambda \gg 1$ und $t > 0$ nähert sich diese Lösung sehr schnell dem Grenzwert $u(\infty) = 1$ an und ist daher für $t > O(1/\lambda)$ so glatt, dass numerische Verfahren sie mit großen Schrittweiten gut approximieren könnten. Wendet man aber z.B. das explizite Euler-Verfahren (6.1.2) hier an, bekommt man bei festem $h > 0$ die Approximationen

$$\begin{aligned} y_m =& y_{m-1} + h\lambda(1 - y_{m-1}) = h\lambda + (1 - h\lambda)y_{m-1} = h\lambda\big(1 + (1 - h\lambda)\big) + (1 - h\lambda)^2 y_{m-2} = \ldots \\ =& h\lambda \sum_{j=0}^{m-1} (1 - h\lambda)^j = h\lambda \frac{1 - (1 - h\lambda)^m}{1 - (1 - h\lambda)} = 1 - (1 - h\lambda)^m. \end{aligned}$$

Dies ist aber nur für $-z := h\lambda < 2$, also für Schrittweiten $h < 2/\lambda$, eine akzeptable Approximation der Lösung.

Diese Problematik kann man nur durch den Einsatz von *impliziten* Verfahren vermeiden. Wendet man im Beispiel das *implizite* Eulerverfahren mit $y_m = y_{m-1} + hf(t_m, y_m)$ an, bekommt man die Näherungen $y_m = 1 - (1 + h\lambda)^{-m}$, die für jedes $\lambda > 0$ und $m \to \infty$ gegen $u(\infty) = 1$ streben. Der Hintergrund ist, dass die Stabilitätsfunktion $\varphi(z)$ für das explizite Verfahren ein Polynom ist, $\varphi(z) = 1 + z$, das nur im Intervall $(-2, 0)$ kleiner eins ist, während beim impliziten Verfahren $\varphi(z) = 1/(1 - z)$ rational ist und kleiner eins auf der gesamten negativen reellen Achse. Insbesonder ist hier sogar $\varphi(-\infty) = 0$ wie bei der Exponentialfunktion.

Obwohl jetzt der Aufwand für jeden Zeitschritt wegen der Lösung von Gleichungssystemen der Dimension n wesentlich größer ist als bei expliziten Verfahren, lohnt sich der Einsatz impliziter Verfahren, da viel größere Zeitschritte gemacht werden können. Parallelität ist bei impliziten Peer-Methoden hier auch einfacher auszunutzen, da die Granularität von Gleichungssystemen wesentlich gröber ist als die der einfachen Funktionsauswertung $f(Y_{m-1,j})$ bei expliziten. Um bei den impliziten Peer-Verfahren für $z = -\infty$ eine verschwindende Stabilitätsmatrix $M(\infty) = 0$ zu bekommen, wird $A = 0$ gewählt. Dann ist die Stabilitätsmatrix (6.2.8) einfach

$$M_m(z) = (I - z\Gamma_m)B_m.$$

Die Ordnungsbedingungen (6.2.6) kann man jetzt aber nur nach B auflösen für Ordnung $s - 1$, da Γ durch Diagonalgestalt und die Bedingung $\gamma_i > 0$ zu stark eingeschränkt ist. In diesem Fall hängt $B = B_m$ aber vom Schrittweitenverhältnis σ_m ab und ist nicht mehr konstant,

$$\tilde{B}_m = S_m P - \sigma_m \tilde{\Gamma} \underbrace{S_m P D F_0^{\mathsf{T}}}. \tag{6.2.12}$$

Nullstabilität läßt sich hier nur mit dem Fall (6.2.10b) garantieren. Hilfreich ist dabei die Beobachtung, dass in $\tilde{B}_m$ die abgeänderte Pascal-Matrix SP obere Dreieckform hat und der Faktor $S_m P D F_0^{\mathsf{T}}$ bei $\tilde{\Gamma}$ sogar streng obere Dreieckform, da F_0^{T} die Spalten der Dreieckmatrix SPD um eins nach rechts verschiebt. Wenn man daher bei $\tilde{\Gamma}$ nur eine einzige Subdiagonale zuläßt, hat $\tilde{B}_m$ auch wieder obere Dreieckform und (6.2.10b) ist erfüllbar durch Anpassung der Hauptdiagonalen. Dazu sei an Satz 6.2.2 erinnert. Die Transformierte der Knotenmatrix C ist die Frobeniusmatrix $V^{-1}CV = F$ und hat nur direkt unter der Hauptdiagonalen noch nichttriviale Elemente. Wenn man daher die γ_i als eine lineare Funktion der Knoten c_i ausdrückt, bekommt man daher

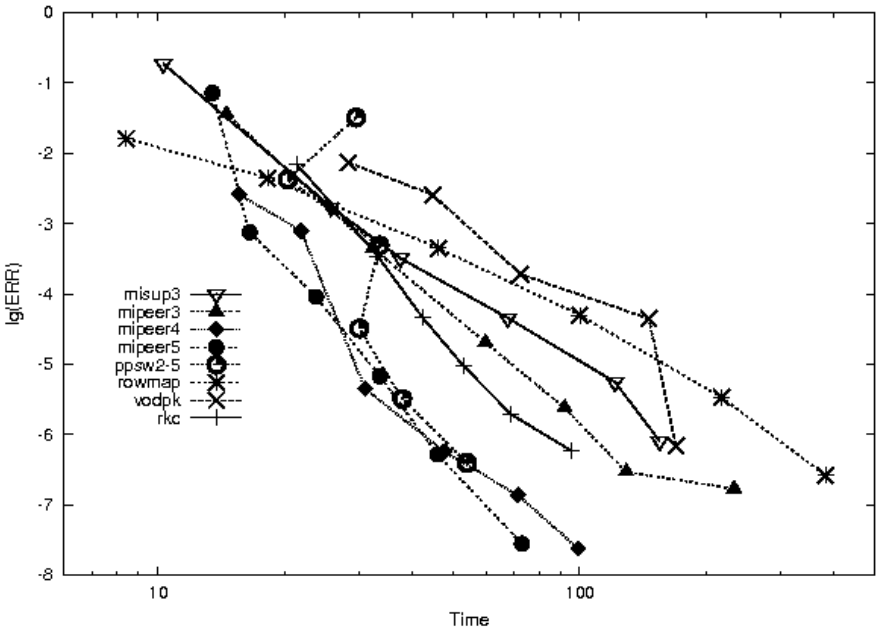
$$\gamma_i = g_0 + g_1 c_i,\ i = 1, \ldots, s,\ \text{d.h. } \Gamma = g_0 I + g_1 C \ \Rightarrow \quad \tilde{\Gamma} = g_0 I + g_1 F$$

in $\tilde{F}$ sogenannten Hessenberg-Gestalt (eine nichttriviale Subdiagonale). Die Diagonalelemente von $\tilde{B}$ sind damit $\tilde{b}_{ii} = (1 - (i-1)g_1)\sigma_m^{i-1}$. Die Bedingung $|b_{ii}| < 1$ für $1 < i \leq s$, welche die Nullstabilität über (6.2.10b) sicherstellt, führt auf folgende Einschränkungen

$s =$	3	4	5	6	7	8
$\sigma <$	2.414	1.678	1.444	1.330	1.262	1.218
$g_1 =$	0.5858	0.4039	0.3075	0.2481	0.2078	0.1788

Die Einschränkungen an das Schrittweitenverhältnis werden für wachsende Ordnung also immer gravierender. Bei dieser Konstruktion spielen die Knoten c_i keine Rolle. Man kann sie unabhängig festlegen, etwa als Tschebyscheffknoetn im Intervall $[-1, 1]$. Dann hat die Vandermonde-Matrix V eine günstige Kondition und die Matrix $B = V\tilde{B}V^{-1}$ eine moderate Norm.

Beispiel 6.2.5 Diese Verfahren `mipeer*` wurden eingesetzt bei der parabolischen Diffusionsgleichung $u_t = \Delta u + g(t, x, y)$ auf dem Gebiet $\Omega = [0, 1]^2$ und einem 100×100 Gitter. Die Problemdimension war also $n = 10^4$. Die zum Vergleich herangezogenen Verfahren VODPK, ROWMAP, RKC sind nicht-parallel. In allen impliziten Verfahren VODPK, ROWMAP und `mipeer*` wurden die Stufensysteme mit Krylov-Iterationsverfahren gelöst.



Index