Technische Universität
Bergakademie Freiberg

Studienarbeit

Lösung großer linearer Gleichungssysteme in Linux-Clustern mittels MPI und PETSc

Jens Oeser

31. Januar 2002

Institut für Geophysik
Gustav-Zeuner-Str. 12
09599 Freiberg

Inhaltsverzeichnis

1 Einführung in die Problematik

In jüngster Vergangenheit ist eine rasante Zunahme der Groß- und Superrechner zu verzeichnen gewesen. Dabei bilden die Parallelrechner den weitaus größeren Teil und haben die bis dahin unangefochtenen Vektorrechner in ihrer Rolle als Spitzenreiter vollständig abgelöst (Meuer, Strohmaier, Dongarra und Simon, 2001). Mit Parallelrechnern sind aber auch immense Kosten nicht nur in der Anschaffung, sondern auch in der Unterhaltung verbunden. Auf Grund dieser Tatsache entstehen gegenwärtig immer mehr Cluster aus PCs oder Workstations. Am Beispiel des Chemnitzer Linux-Cluster (CLiC) sieht man anschaulich, dass dieses Bemühen auch von Erfolg gekrönt ist, denn dieser Cluster aus 528 PCs nimmt Platz 156 in der Liste der 500 schnellsten Rechner ein (Meuer et al., 2001, Stand Juni). Angesichts solcher Resultate stellt sich natürlich die Frage, inwieweit sich diese auch mit einem kleinen Cluster aus handelsüblichen PCs bei der dreidimensionalen geoelektrischen Modellierung und Inversionsrechnung erreichen lassen. In Zukunft werden immer feiner diskretisierte Modelle des Untergrundes erstellt werden, um der Komplexität des Systems Erde Rechnung zu tragen. Gleichzeitig gewinnen mit der immer feiner werdenden Diskretisierung die zu lösenden linearen Gleichungssysteme an Größe. Damit ist eine überlineare Zunahme des Rechenaufwandes verbunden und genau an diesem Punkt setzen die parallelen iterativen Gleichungslöser an. Man erwartet in der Theorie eine Reduktion der Rechenzeit auf $1/P$ beim Einsatz von P Rechnern. Mit Blick auf die stochastischen Inversionsverfahren, bei denen bis zu 200 000 Vorwärtsrechnungen schon allein im eindimensionalen Fall nötig sind, wird der Einsatz solcher Cluster auch in diesem Gebiet immer mehr an Bedeutung gewinnen.

2 Zielstellung

Lassen die Erfahrungen mit dem Chemnitzer Linux-Cluster erwarten, dass schon kleine Cluster Parallelisierungspotential besitzen? Um dies herauszufinden, sollten mit einem Linux Cluster aus 7 PCs Erfahrungen gesammelt werden. Die Vorbereitungen für die verschiedenen Untersuchungen umfasste:

- Installation des Betriebssytems und der benötigten Pakete
- Einrichten des Nutzers *mpi2*.

Um Aussagen über das Parallelisierungsvermögen eines kleinen Clusters treffen zu können, mussten folgende Schritte abgearbeitet werden:

- Einbindung der Software in *GEO4D* (Börner, Günther und Käppler, 1998)
- Bewertung der Effizienz über Skalierungstests.

3 Linux-Cluster

Auf dem Cluster von 7 Rechnern wurde als Betriebssytem ein *SuSE* Linux sowie die beiden Pakete *MPICH* (Ashton et al., 2000) und *PETSc* (Balay, Gropp, McInnes und Smith, 1998) installiert. Darüber hinaus musste noch ein Account (*mpi2*) auf jedem Rechner eingerichtet und das Homeverzeichnis per "Network File System", kurz NFS, auf jeden PC exportiert werden. Mit diesen Schritten waren die Grundlagen gelegt, um mit dem Linux-Cluster parallel rechnen zu können. Es wurde bewusst ein Parallelrechner aus unterschiedlich ausgestatteten PCs erstellt, um die Möglichkeiten auszuloten, ob eine solches System, wie es vielleicht in jedem Ingenieurbüro oder Institut vorhanden ist, auch später bei dem Anwender Vorteile bringen kann. Darüber hinaus sind die Rechner jederzeit von den Benutzern im Nichtclusterbetrieb einsetzbar, somit ist eine effiziente Nutzung der begrenzten Resourcen möglich.

3.1 Hard- und Software

Detaillierte Informationen über die unterschiedlich ausgestatteten PCs können der Tabelle 3.1 entnommen werden. Darin wird die Heterogenität des gesamten Clusters noch einmal verdeutlicht. Neben den bereits aufgeführten Versionen des *SuSE* Linux Betriebssystems

Rechner	**Prozessor**	**Arbeitspeicher**	**Netzanschluss**	**Betriebssytem**
matlab	Athlon 800*MHz*	768*MB* SDRAM	100*MBit*	*SuSE* 7.1
boreale	Athlon 800*MHz*	512*MB* SDRAM	100*MBit*	*SuSE* 7.0
bud	Athlon 800*MHz*	256*MB* SDRAM	100*MBit*	*SuSE* 7.0
paddy	Athlon 800*MHz*	256*MB* SDRAM	100*MBit*	*SuSE* 7.0
pub03	Duron 800*MHz*	128*MB* SDRAM	100*MBit*	*SuSE* 7.2
pub04	Duron 800*MHz*	128*MB* SDRAM	100*MBit*	*SuSE* 7.2
pub05	Duron 800*MHz*	128*MB* SDRAM	100*MBit*	*SuSE* 7.2

Tabelle 3.1: Rechnerausstattung

wurden noch die *MPICH*-Implementierung des "Message Passing Interface", kurz *MPI*, in der Version 1.2.1-10 und die Version 2.1.0.1 des "Portable Extensible Toolkit for Scientific Computation", kurz *PETSc*, installiert.

3.2 Performance

Die einzelnen PCs unterscheiden sich nur geringfügig in der Leistungsfähigkeit ihrer Prozessoren. Aber auf die Gesamtleistung eines Systems hat nicht nur die Leistung der CPU Einfluss, sondern in großem Maße der vorhandene Arbeitsspeicher. Dieser beeinflusst die Performance des Rechners erheblich. Um diesen Einfluss quantitativ beurteilen zu können,

ist auf drei Rechnern ein Benchmark[1] durchgeführt wurden. Die Ergebnisse sind in Tabelle 3.2 aufgeführt. Die Ergebnisse zeigen deutlich, dass mit zunehmendem Arbeitsspeicher die Laufzeit $\overline{t}$ des Benchmarks abnimmt und damit die Performance des Gesamtsystems zunimmt.

Rechner	$\overline{\mathbf{t}}$ [s]	$\mathbf{s}_{\overline{\mathbf{t}}}$ [s]
matlab	272	2,0
bud	312	1,8
pub04	363	7,2

Tabelle 3.2: Performancetest ausgewählter Rechner

Bei der parallelen Lösung eines Gleichungssystems in einem Cluster beeinflussen nicht nur die bereits genannten Faktoren die Leistung des Clusters, sondern auch in starkem Maße die Netzverbindung der PCs untereinander. Um die Frage nach der Latenzzeit und maximalen Übertragungsrate klären zu können, wurde ein Test des Netzwerkes mit dem Programm *NetPIPE* (Snell, Helmer und andere, 1998) durchgeführt. Die Ergebnisse dieses Tests sind in Abbildung 3.1 dargestellt. Für die Übertragung von Daten innerhalb eines Netzwerkes

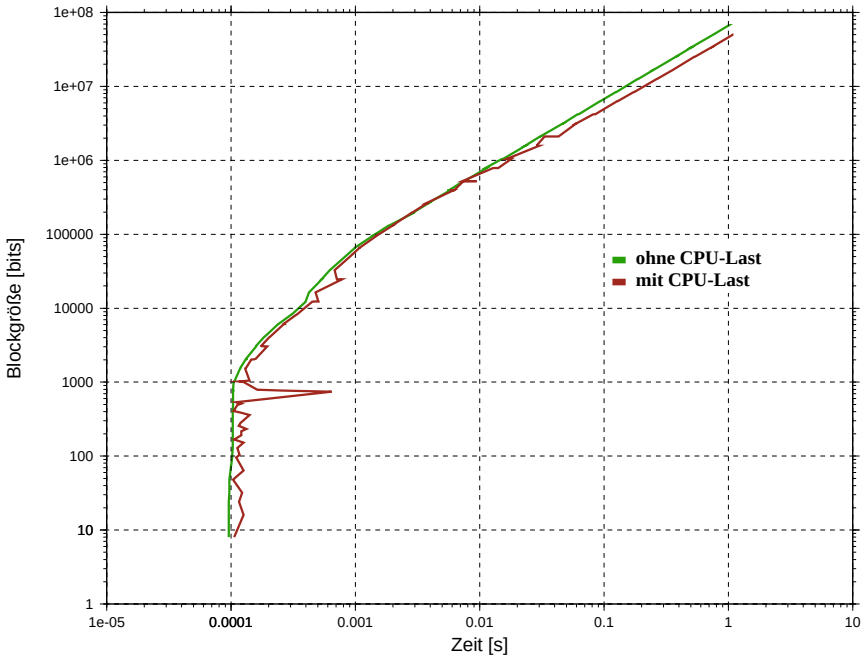

Abbildung 3.1: Ergebnis des Netzwerkbenchmark

gibt es eine Reaktionszeit, diese wird minimal benötigt, um Datenblöcke zu übertragen. Man bezeichnet diese Zeit als Latenzzeit. Für den verwendeten Linux-Cluster kann man aus

[1] Compilierung eines Linux-Kernel gleicher Konfiguration

Abbildung 3.1 eine Latenzzeit von $0,1\mu s$ ablesen. Diese Zeit wird mindestens benötigt, um Datenblöcke der Größe 10 bis $1000bit$ zu übertragen. Des Weiteren lässt sich aus dem mittleren Anstieg zwischen $1\mu s$ und $1s$ der Kurve ohne CPU-Last eine maximale Übertragungsrate von $67Mbit/s$ ableiten. Dieser Wert entspricht dem aus der Praxis zu erwartenden Wert von 70% eines 100–$Mbit$-Netzwerkes. Die zweite Kurve innerhalb des Diagramms veranschaulicht den Einfluss einer bereits geringen CPU-Auslastung auf die Ergebnisse des Benchmarks und der damit steigenden Latenzzeit und geringeren maximalen Übertragungsrate.

4 Leistungsbewertung

Eine quantitative Bewertung der Leistungsfähigkeit eines Parallelrechners schließt nicht nur die Leistungfähigkeit der Hardwarekomponenten und Rechnerarchitektur, sondern insbesondere die Algorithmen und deren Implementierung mit ein. Deshalb genügt es nicht, die für einen Einzelrechner überlicherweise angegebene Mips-Rate (Million instructions per second) oder Mflops-Rate (Million floating point operations per second) anzugeben, da bei diesen Werten die Parallelität der Hardware, die Systemsoftware, die Lastparameter, der Speicherausbau pro Prozessor und eventuelle Speicherhierarchien keine Berücksichtigung finden. Eine vergleichbare Basis bieten Benchmarkprogramme für Parallelrechner, welche für diese Untersuchungen jedoch keine praktische Relevanz haben, da die in dieser Studienarbeit verwendeten numerischen Algorithmen nicht in die Benchmarksoftware eingebunden sind. Auf Grund dessen ist es zweckmäßig, andere Kriterien zur Bewertung heranzuziehen. Dies soll Gegenstand der nächsten Abschnitte sein. Dabei lehnen sich die Ausführungen stark an Huber (1997) und Mönch (WS 00/01) an.

4.1 Speedup

Idealerweise könnte man erwarten, mit P Rechnern P-mal schneller eine Berechnung zu beenden. Dies ist aber nur für ganz einfache Probleme zu erreichen, wie später im Abschnitt zu Amdahl's Gesetz (4.4) erklärt wird. Im Normalfall wird der Geschwindigkeitszuwachs geringer ausfallen. Der Faktor des Geschwindigkeitsgewinnes bzw. Laufzeitgewinnes wird als *Speedup* bezeichnet. Es existieren verschiedene Definitionen des Speedup. Man unterscheidet zwischen Modellen mit fester Problemgröße und mit fester Laufzeit. Der Grund für diese Unterscheidung liegt in der unterschiedlichen Motivation, mit der man einen Parallelrechner nutzen kann. Zum einen kann man versuchen, die Laufzeit für eine Problemgröße zu reduzieren, oder man versucht in gleicher Laufzeit, ein größeres Modell zu berechnen.

4.1.1 Speedup für eine feste Problemgröße

Mittels der Gesamtlaufzeit t eines parallelen Algorithmus lässt sich der *parallele Speedup* $S_{par}(P)$ berechnen über

$$S_{par}(P) = \frac{t(1)}{t(P)}, \tag{4.1}$$

wobei P die Anzahl der verwendeten Rechner beschreibt. Die Bestimmung der Laufzeit erfolgt für $t(1)$ und $t(P)$ mit identischem parallelen Algorithmus. Der parallele Speedup ist damit durch P nach oben hin beschränkt. Er wird häufig auch als *relativer Speedup* bezeichnet. Durch die Definition des parallelen Speedup ist es möglich, die verschiedenen Implementierungen eines Algorithmus für eine Rechnerarchitektur zu vergleichen. Allerdings sind große Speedup-Werte noch kein Hinweis auf eine gute Parallelisierung, da die Speedup-Zahlen für die Implementierung eines Algorithmus auf verschiedenen Architekturen unterschiedlich sein können. Es muss immer noch ein Vergleich der Laufzeiten vorgenommen werden.

In der bis jetzt verwendeten Definition des Speedup wird derselbe Algorithmus sowohl auf einem als auch auf P Rechnern ausgeführt. Ein effizienter paralleler numerischer Code ist nicht immer auch der schnellste sequentielle. Diese Überlegung führt zur Definition des auf den schnellsten bekannten sequentiellen Algorithmus normierten Speedup

$$S_{ges}(P) = \frac{\text{Laufzeit des schnellsten sequentiellen Algorithmus}}{\text{Laufzeit des parallelen Algorithmus auf } P \text{ Rechnern}}. \tag{4.2}$$

$S_{ges}(P)$ bezeichnet den *gesamten Speedup*, der kleiner oder gleich P ist.

4.1.2 Speedup für eine feste Laufzeit

Mit steigender Rechnerzahl ist auch eine Steigerung der gesamten Problemgröße wünschenswert. Damit lässt sich das Problem nicht mehr auf einem Rechner ausführen, was zu Problemen mit der bisherigen Definition des Speedup führt. Es ergibt sich eine Definition des Speedup normiert auf eine feste Laufzeit.

$$S_{skal}(P) = \frac{\text{Modellgröße auf } P \text{ Rechnern bei fester Laufzeit } t}{\text{Modellgröße auf 1 Rechner bei fester Laufzeit } t} \tag{4.3}$$

Ziel ist nicht eine Reduzierung der Gesamtlaufzeit, sondern eine Erhöhung der Problemgröße bei gleich bleibender Laufzeit. Daher erscheint P=1 im Nenner. Dies wird als *skalierbarer Speedup* $S_{skal}(P)$ bezeichnet.

4.2 Effizienz

Der Begriff der Effizienz ist eng verbunden mit dem des Speedup. Man kann ebenfalls zwei Definitionen unterscheiden.

4.2.1 Effizienz für eine feste Problemgröße

Die *parallele Effizienz* lässt sich wie folgt definieren

$$E_{par}(P) = \frac{t(1)}{P \cdot t(P)} = \frac{S_{par}(P)}{P}. \tag{4.4}$$

Sie ist durch $E_{par}(P) = 1$ nach oben beschränkt. Nach der Definition des gesamten Speedup ergibt sich die Definition für die *gesamte Effizienz* $E_{ges}(P)$.

$$E_{ges}(P) = \frac{S_{ges}(P)}{P} \ . \tag{4.5}$$

4.2.2 Effizienz für eine feste Laufzeit

Analog zu obigen Definitionen ergibt sich die Effizienz für eine feste Laufzeit zu

$$E_{skal}(P) = \frac{S_{skal}(P)}{P} \ . \tag{4.6}$$

$E_{skal}(P)$ wird als *skalierbare Effizienz* bezeichnet und wird maximal 1.

4.3 Skalierbarkeit

Ein paralleler Algorithmus skaliert perfekt, wenn sich beim Einsatz von P Rechnern der gesamte Lösungsaufwand um den Faktor P reduziert. Dies ist identisch mit perfektem parallelen Speedup ($S_{par}(P) = P$). Analog gibt es eine Definition für die Skalierbarkeit für eine feste Problemgröße, d.h. ein paralleler Algorithmus ist skalierbar, falls trotz Anwachsen der Unbekannten und Rechneranzahl die Lösungszeit konstant bleibt. Perfekte Skalierbarkeit setzt aber eine Minimierung des Kommunikationsaufwandes voraus. Dies alles lässt sich im Allgemeinen nur sehr schwer erreichen, so dass man gezwungen ist, zwischen den verschiedenen beeinflussenden Faktoren Kompromisse einzugehen.

4.4 Amdahl's Gesetz

Viele parallele Algorithmen lassen sich nicht vollständig parallelisieren, d.h. es existieren Codeteile, die nur auf einem Rechner ausgeführt werden. Dies könnte zum Beispiel eine Multiplikation oder Division zweier skalarer Werte sein. Wie sich der nicht parallelisierbare Anteil eines Programms auf die Gesamtrechenzeit auswirkt, besagt das Gesetz von Amdahl. Mit Hilfe dieses Gesetzes lässt sich eine erweiterte Definition des Speedup S angeben.

$$S = \frac{1}{A_s + \frac{A_p}{P}} \tag{4.7}$$

Dabei bezeichnet A_s den sequentiellen Anteil und A_p den parallelen Anteil am Algorithmus. Es ist im Allgemeinen schwierig, den sequentiellen Anteil eines Algorithmus zu bestimmen. Demnach erzielen Parallelrechner nur Vorteile bei Algorithmen mit geringem sequentiellen Anteil.

5 Strategien zur Parallelisierung

Einen Einblick in die verschiedenen Rechnerarchitekturmodelle erhält man in Huber (1997) und Mönch (WS 00/01). Darin wird genauer auf die verschiedenen Architekturen und die sich daraus ableitenden Vor- und Nachteile eingegangen. Der für die weiteren Tests verwendete Linux-Cluster lässt sich als MIMD-Rechner mit verteiltem Speicher charakterisieren. Die Abkürzung MIMD steht für "multiple instruction stream, multiple data stream". Jeder Prozessor im Cluster verfügt über seinen eigenen Speicher und ist über ein Netzwerk mit den anderen verbunden.

5.1 Ebenen der Parallelität

In Huber (1997) wird eine Unterteilung der parallel ablaufenden Programme in verschiedene Ebenen oder Schichten vorgenommen. Dabei unterscheidet man die 4 Ebenen, wie sie in Abbildung 5.1 dargestellt sind, hinsichtlich ihres Parallelisierungsgrades. Bei der höchs-

Abbildung 5.1: Schichtenmodell für verschiedene Parallelisierungsgrade

ten Schicht der Parallelität, der Ebene 3, spricht man von dem sogenannten Multitasking. Dies bedeutet, dass mehrere Benutzerprozesse parallel abgearbeitet werden. Diese Art der Parallelität wird durch das Betriebssystem zur Verfügung gestellt. Damit ist jeder Benutzer selbst für die Parallelisierung verantwortlich, demgegenüber ist der maximal zu erreichende Parallelisierungsgrad sehr hoch.

Bei Ebene 2 handelt es sich um die Datenparallelität, d.h. man parallelisiert mittels Datenstrukturen. Diese Art tritt oftmals in Schleifen auf und findet bei der Lösung linearer Gleichungssyteme Anwendung, wobei dort die Vektoren und Matrizen für die Datenparallelität sorgen. Analog zu Schicht 3 ist der Parallelisierungsgrad sehr hoch, erfordert aber auch vom Anwender ein tieferes Wissen über die verwendeten Methoden.

In der Ebene 1 liegt eine Parallelität in Form von Instruktionen oder Anweisungen vor. Dies wird im Allgemeinen nur von parallelisierenden Compilern unterstützt. Der zu erreichende Parallelisierungsgrad ist im Vergleich zu den bereits erwähnten Ebenen eher gering.

Bei der letzten Ebene der Parallelität ist ebenfalls nur ein kleiner Parallelisierungsgrad erreichbar und wird nur durch die entsprechenden Compiler unterstützt, erfordert aber auch wenig Wissen vom Anwender.

Aus diesen Schichten leiten sich verschiedene Empfehlungen für die unterschiedlichen Rechnerarchitekturen ab. Demnach wäre eine Parallelisierung im verwendeten Cluster durch parallel auf den verschiedenen Rechnern ablaufende Prozesse möglich. Diese Art erfordert aber keine explizite Umstellung der Programme und ist relativ einfach durch entsprechende Shellskripte zu implementieren, deshalb findet sie im Rahmen der Studienarbeit keine weitere Beachtung. Die zweite Ebene ist hingegen viel interessanter und erfordert eine explizite Parallelisierung des Programms und damit der Gleichungslöser. Für die Implementierung bieten sich die verschiedenen Programmiersprachen wie High Performance Fortran und pC++ an. Es ist aber auch möglich, bereits vorhandene Kommunikationsbibliotheken und darauf aufbauende Bibliotheken zu verwenden. Auf diese Möglichkeiten wird in den folgenden Abschnitten noch genauer eingegangen.

Weiterhin kann zwischen expliziter und impliziter Parallelität unterschieden werden. Bei der impliziten Variante erfolgt die Umsetzung durch entsprechende parallelisierende Compiler, dazu muss aber eine Analyse der Datenabhängigkeiten vorgenommen werden. Für den verwendeten Linux-Cluster sind solche Compiler nicht erhältlich. Deshalb muss die explizite Parallelität Verwendung finden. Dabei kann man nur eine geringe Unterstützung durch den Compiler erwarten und muss selbst für die Parallelisierung sorgen. Um aber trotzdem einen hohen Parallelisierungsgrad zu erreichen, bietet sich die Verwendung von entsprechenden Bibliotheken an.

Für die zu entwickelnden Programme ergeben sich nun zwei verschiedene Parallelisierungsansätze, einmal der datenparallele und zum anderen der instruktionsparallele Ansatz. Beim ersten führt die Realisierung auf MIMD-Rechnern zum sogenannten SPMD-Modus, was für "single programm, multiple data" steht. Dabei führen alle Prozessoren denselben Programmcode aus, wenden ihn aber auf unterschiedliche Daten an. Eine hohe Datenparallelität ist für eine effektive Parallelisierung unablässig, da der Parallelisierungsgrad von der Problemgröße abhängt. Die Instruktionsparallelität ist hingegen völlig unabhängig von der Problemgröße und damit von der Anzahl der Unbekannten.

5.2 Komplikationen

Bei den Komplikationen, die im Verlauf der Entwicklung und Implementierung parallel ablaufender Algorithmen auftreten können, handelt es sich im Wesentlichen um die Synchronisation und Verklemmung. Durch Kommunikation, d.h. Austausch von Daten, lassen sich

Algorithmen synchronisieren. Dabei verwendet man die Synchronisation nicht nur um Daten auszutauschen, zum Beispiel am Ende eines Iterationsschrittes, sondern auch um die Beendigung eigener Berechnungen mitzuteilen oder um den weiteren Ablauf des Algorithmus simultan zu gestalten. Die Problematik der Verklemmung ist wiederum sehr eng mit der der Synchronisation verbunden. Sie entsteht, wenn parallel arbeitende Prozesse untereinander auf Daten warten. Dies tritt ausschließlich auf MIMD-Rechnern auf und kann durch Datenabhängigkeitsuntersuchungen gelöst werden.

5.3 Message Passing Interface

Bei dem Message Passing Interface handelt es sich um eine Kommunikationsbibliothek, die auf dem nachrichtenorientierten Programmiermodell (engl. "message passing model") aufbaut. Neben dem bereits erwähnten Modell existieren noch das datenparallele und das mit gemeinsamen Speicher. Allen drei Modellen ist der SPMD-Modus übergeordnet. Obwohl das nachrichtenorientierte Modell am weitesten verbreitet ist, hat es als großen Nachteil den hohen Implementierungsaufwand.

Bei der Umsetzung des nachrichtenorientierten Programmiermodells in MPI erfolgte bereits von Anfang an eine Standardisierung unter Einbezug der Hersteller und Universitäten. Auf Grund dessen hat sich heute MPI zum Quasi-Standard entwickelt. Die Parallelisierung erfolgt auf den Ebenen 2 und 3 und erreicht damit ein hohes Maß an Effizienz. Als Grundfunktionen stehen Anweisungen für die Synchronisation und das Senden und Empfangen von Daten oder Nachrichten bereit. Eine umfangreiche Einführung zu MPI erhält man in Pacheco (1997).

5.4 Portable Extensible Toolkit for Scientific Computation

PETSc ist eine große Bibliothek für die Lösung partieller Differentialgleichungen und verwandter Probleme. Sie basiert auf MPI und diversen Bibliotheken zur lineraen Algebra wie LAPACK[2] und BLAS[3]. Es lassen sich dicht und dünn besetzte lineare und nichtlineare Gleichungssysteme lösen. Des Weiteren stehen umfangreiche Fehlerfunktionen und einfache Grafikfunktionen zur Verfügung. Weiterhin ist die Verwendung der Bibliothek in Fortran, C und C++ möglich, wobei sie primär für C und C++ erstellt wurde. Laufzeitoptionen für die einzelnen Funktionen lassen sich während des Programmstarts einfach durch Kommandozeilenparameter übergeben. Eine detailliertere Einführung in PETSc erfolgt zu einem späteren Zeitpunkt.

[2] Linear Algebra PACKage
[3] Basic Linear Algebra Subprogram

6 Parallelisierung numerischer Algorithmen

Im Folgenden wird auf Aspekte bei der Parallelisierung iterativer numerischer Verfahren eingegangen. Tiefer in diese Problemstellung wird in den Ausführungen von Demmel, Heath und van der Vorst (1993) und Barrett et al. (1994) eingedrungen. Beim Lösen linearer Gleichungssysteme mit Hilfe von iterativen Methoden verursachen

- die Matrix-Vektor-Produkte
- die Vektor-Vektor-Produkte
- die Vektoraktualisierungen
- die Vorkonditionierung

den größten numerischen Aufwand. In diesen vier Punkten liegt der Ansatz zur effizienten Parallelisierung numerischer Algorithmen.

6.1 Matrix-Vektor-Produkt

Die Berechnung des Matrix-Vektor-Produktes ist für Parallelrechner mit verteiltem Speicher nicht einfach zu implementieren. Eine schematische Darstellung dessen ist in Abbildung 6.1 zu sehen. Durch die Farbkodierung wird die Verteilung der verschiedenen Zeilen auf die Prozessoren kenntlich gemacht. Es befinden sich immer auf ein und demselben Rechner die gleichen Zeilen des Vektors und der Matrix. Zur Berechnung des Matrix-Vektor-Produktes werden aber alle Zeilen des Vektors auf jedem Prozessor benötigt. Entweder es wird der gesamte Vektor im Speicher des jeweiligen Rechners gehalten oder man muss durch Kommunikation dafür sorgen, dass alle Prozessoren die benötigten Elemente des Vektors erhalten. Die zweite Variante wird von PETSc genutzt. Bei der Berechnung des Produktes werden die

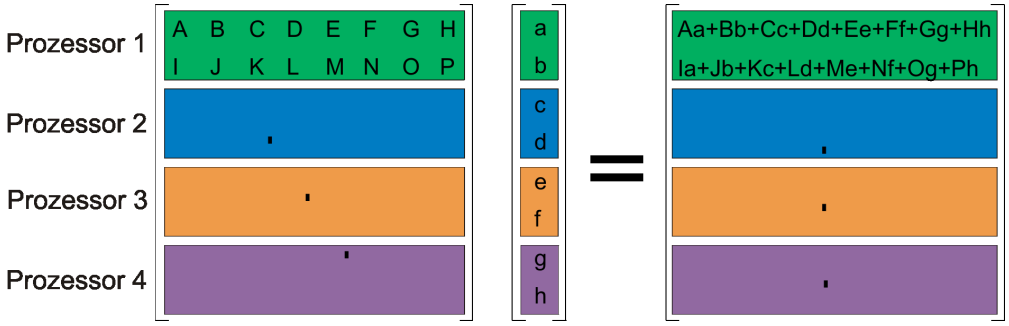

Abbildung 6.1: Berechnung des Matrix-Vektor-Produktes

benötigten Elemente des Vektors durch die anderen Prozessoren versendet. Dies bedeutet, dass die Elemente des Prozessors 2 an die Prozessoren 1, 3 und 4 versendet werden müssen.

Gleiches muss analog für die anderen Prozessoren erfolgen. Danach werden die Multiplikationen und Additionen durch jeden Prozessor ausgeführt. Damit sind die Zeilen des neu berechneten Vektors auf den jeweiligen Rechnern vorhanden und die empfangenen Elemente des Vektors können aus dem Speicher gelöscht werden.

6.2 Vektor-Vektor-Produkt

Die Berechnung des Vektor-Vektor-Produktes (inneres Produkt) ist einfach zu parallelisieren. Jeder Prozessor berechnet das innere Produkt seiner Elemente, das sogenannte lokale innere Produkt (LIP) und verschickt dieses. Daraus wird dann das globale innere Produkt gebildet. Dies kann wie in Abbildung 6.2 schematisch dargestellt erfolgen. In diesem Schema

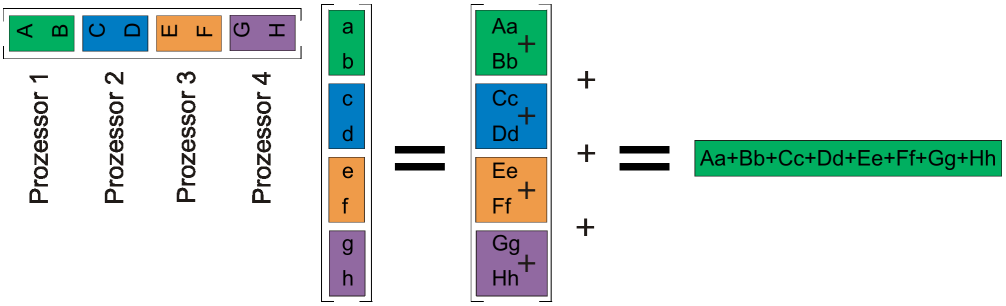

Abbildung 6.2: Berechnung des Vektor-Vektor-Produktes

werden alle LIP auf Prozessor 1 angesammelt. Er berechnet nach Erhalt aller LIP das globale innere Produkt und versendet es an die anderen Prozessoren. Die Berechnung ist noch in einer weiteren Variante möglich. Dabei senden alle Prozessoren ihr LIP an alle anderen. Danach berechnet jeder Prozessor für sich das globale innere Produkt. Die Entscheidung für eines der beiden Schemata hängt von der Wahl der Iterationsmethode ab. Das Vektor-Vektor-Produktes fungiert bei Iterationsverfahren vom Typ Conjugate Gradient (CG) als Synchronisationspunkt, da alle weiteren Rechnungen erst dann starten können, wenn das globale innere Produkt auf jeden Prozessor vorhanden ist. Damit ist es nicht möglich, die Kommunikationszeit durch Berechnungen zu überbrücken. Aus diesem Grund existieren abgewandelte CG Schemata für Parallelrechner bei denen es möglich ist, durch eine geschickte Umsortierung die Kommunikation mit Berechnungen zu überlappen.

6.3 Vektoraktualisierung

Die Aktualisierung der Elemente eines Vektors ist trivial parallelisierbar, da jeder Rechner seine eigenen Elemente aktualisiert.

6.4 Vorkonditionierung

Die Parallelisierung des Vorkonditionierers gestaltet sich schwierig. Da die meisten Vorkonditionierer nicht im Hinblick auf eine Verwendung mit Parallelrechnern entwickelt wurden, kann es zu Problemen bei der Umsetzung kommen. Einfach zu implementierende parallele Vorkonditionierer sind jene, bei denen auf jedem Rechner ein sequentieller Vorkonditionierer (SOR, SSOR und ILU) auf die lokale Submatrix angewendet wird. Sie werden auch als Block-Jacobi-Vorkonditionierer bezeichnet. Eine Verbesserung dieser stellen die Additiv-Schwarz-Vorkonditionierer dar, bei denen durch eine Überschneidung eine Verbesserung der Effizienz erreicht werden kann. Unter Überschneidung versteht man die Anwendung eines sequentiellen Vorkonditionierers nicht nur auf die lokale Submatrix, sondern auch auf gewisse Bereiche (meist Zeilen) der lokalen Submatrix der benachbarten Rechner. Durch die Überschneidung werden Teile der Matrix besser vorkonditioniert, was bei der Lösung des lineraren Gleichungssystems in geringeren Iterationszahlen resultiert. Auf die Problematik der Vorkonditionierung wird in Barrett et al. (1994), Demmel et al. (1993) und Heath et al. (1991) genauer eingegangen.

7 Nutzung von PETSc

Als Ausgangspunkt für die Tests wird die Numerikbibliothek PETSc des Argonne National Laboratory verwendet. Wie unter 5.4 kurz vorgestellt, sind umfangreiche Funktionen zur Entwicklung numerischer Verfahren vorhanden. Dabei wird keine Beschränkung hinsichtlich der Prozessoranzahl vorgenommen. In gleichem Umfang sind Funktionen für die sequentielle (sprich Einzelrechner) und parallele Abarbeitung von Problemen vorhanden. Dabei ähneln die Funktionsaufrufe stark denen in MATLAB. Da man nicht immer vom Anwender ein tiefes Verständnis der parallelen Konzeption vorraussetzen kann, werden die einzelnen Funktionen hinsichtlich ihrer Komplexheit unterschieden. Beim geringsten Grad an Komplexheit wird von der Bibliothek auf eine effiziente Auswahl der numerischen Methoden durch sinnvolle Defaultwerte geachtet. Im weiteren Verlauf dieses Kapitels werden die von der Programmiersprache C her bekannten Nomenklaturen benutzt. Für die Umsetzung in Fortran bietet sich das Studium des Manuals (Balay, Gropp, McInnes und Smith, 2001) und der Hilfeseiten (Balay et al., 1998) an. Es können nicht alle Funktionen Gegenstand dieser Ausführungen sein, so dass wieder auf die beiden erwähnten Quellen verwiesen werden muss.

7.1 Grundlagen

Bei der Verwendung von PETSc bestehen keine Einschränkungen hinsichtlich der von C bekannten Funktionen und Datentypen. Es werden lediglich Erweiterungen vorgenommen. Neue Datentypen für Matrizen, Vektoren, Gleichungslöser und Verwaltungsaufgaben werden

wie folgt definiert:

- `Mat` für Matrizen
- `Vec` für Vektoren
- `SLES` für lineare Gleichungslöser
- `PC` für Vorkonditionierer
- `KSP` für Krylov-Unterraummethoden
- `IS` für Indexerzeugung
- `VecScatter` für Vektorumwandlung
- `PetscViewer` für Ausgabe
- `PetscLogDouble` für Zeit.

Die mittels dieser Datentypen definierten Variablen fungieren als Handler zur Änderung der Eigenschaften von Funktionen oder Methoden.

Zum ordnungsgemäßem Starten und Beenden des Programmes werden die beiden Funktionen:

- `PetscInitialize()`
- `PetscFinalize()`

benötigt. Durch `PetscInitialize()` wird für eine korrekte Parameterübergabe an das Programm und PETSc selbst gesorgt. Mittels `PetscFinalize()` werden MPI und die PETSc-Bibliotheken korrekt beendet. Des Weitern werden bei eventuell aufgerufenen Log-Optionen die entsprechenden Ausgaben gestartet.

Bei der Verwendung der Standardausgabefunktionen in C schreibt jeder Rechner in die Ausgabe. Dies führt zu mehrfachen Ausgabe des selben Textes, womit bei 4 Prozessoren bereits die Übersichtlichkeit verlorengeht. Um dieses Problem zu beseitigen, ersetzen die folgenden Funktionen die originalen C-Funktionen:

- `PetscPrintf()` an Stelle von `printf()`
- `PetscFPrintf()` an Stelle von `fprintf()`.

Um eine realistische Zeitmessung vornehmen zu können, ist es erforderlich, zum gleichen Zeitpunkt innerhalb des Codes die Zeitmessung zu starten. Auf Grund der Heterogenität innerhalb des Clusters würde der eine Rechner diesen Punkt eher erreichen als ein anderer,

so dass für eine Synchronisation gesorgt werden muss. Diesem Umstand trägt die Funktion `PetscGetTime()` Rechnung und ersetzt `clock()`.

Zur Fehlerdiagnose innerhalb des Programmes sollten alle Funktionen stets wie folgt aufgerufen werden:

```
ierr=PetscFUNCTION();CHKERRQ(ierr);
```

Der Variablen `ierr` wird ein Integerwert durch `PetscFUNCTION()` zugeordnet. Je nach Variablenwert wird durch die Funktion `CHKERRQ()` keine oder eine entsprechende Fehlermeldung ausgegeben. Anhand dieser ist es möglich, innerhalb des Codes die Probleme zu lokalisieren und zu beseitigen.

7.2 Vektoren

Bevor sich ein Vektor abspeichern lässt, muss der benötigte Speicherplatz reserviert werden. Dies geschieht über

```
VecCreateSeq(PETSC_COMM_SELF,Vektorlänge,&Vektorname)
```

oder

```
VecCreateMPI(PETSC_COMM_WORLD,LokaleVektorlänge,Vektorlänge,&Vektorname).
```

Dabei charakterisiert das Kürzel `Seq` die sequentielle und `MPI` die parallele Variante. Mit `PETSC_COMM_SELF` wird ein Rechner (der auf dem das Programm gestartet wird) angesprochen, wohingegen mit `PETSC_COMM_WORLD` alle Rechner angesprochen werden. Nachdem der Speicher reserviert wurde, müssen die einzelnen Elemente des Vektors innerhalb einer Schleife übergeben werden. Dies kann über

```
// Schleife über LokaleVektorlänge
   VecSetValues(Vektorname,1,&Zeile,&Wert,INSERT_VALUES)
```

realisiert werden. Nachdem alle Werte eingefügt wurden, müssen diese noch auf die richtigen Rechner verteilt werden. Dazu wird das Funktionspaar

```
VecAssemblyBegin(Vektorname)
VecAssemblyEnd(Vektorname)
```

genutzt. Zwischen diesen beiden Funktionsaufrufen können Berechnungen erfolgen, da in diesem Zeitraum nur Kommunikation anfällt. Am Ende jedes Programmes muss der reservierte Speicher wieder freigegeben werden. Dazu wird

```
VecDestroy(Vektorname)
```

verwendet. Neben diesen Funktionen existieren noch diverse andere zur Veränderung von Vektoren.

7.3 Matrizen

Für die verschiedenen Matrixspeicherformate existieren unterschiedliche Funktionsaufrufe. Anhand des Compressed Row Storage Formates (in PETSc als AIJ Format bezeichnet, mehr dazu in Barrett et al. (1994)) wird die prinzipielle Vorgehensweise erklärt. Dazu muss zuerst der benötigte Speicherplatz reserviert werden über

```
MatCreateSeqAIJ(PETSC_COMM_SELF,Zeilenzahl,Spaltenzahl,
   nz,nnz,&Matrixname)
```

oder

```
MatCreateMPIAIJ(PETSC_COMM_WORLD,LokaleZeilenzahl,LokaleSpaltenzahl,
   Zeilenzahl,Spaltenzahl,d_nz,d_nnz,o_nz,o_nnz,&Matrixname).
```

Bei der sequentiellen Variante des Befehls bezeichnet `nz` die maximal zu erwartenden Nichtnullelemente. Mit `nnz` wird ein Feld übergeben, wo jedes Feldelement die Anzahl der Nichtnullelemente der jeweiligen Zeile beinhaltet. Es ist aber nur notwendig, eines der beiden anzugeben, da jeweils nur eins genutzt wird. Dies ist ebenso bei der parallelen Version der Funktion der Fall, nur bezeichnen hier die Buchstaben `d` und `o` die diagonale Submatrix und die nichtdiagonale Submatrix. Als diagonale Submatrix wird der lokal auf jedem Rechner vorhandene quadratische Teil der Matrix (Anzahl der Zeilen gleich Anzahl der Spalten) bezeichnet, der sich um die Diagonale befindet. Die restlichen Elemente der Zeile gehören der nichtdiagonalen Submatrix an. Wenn der benötigte Speicher im richtigen Umfang reserviert wurde, können die einzelnen Werte innerhalb einer Schleifenkonstruktion übergeben werden. Dazu kann folgendes Schema verwendet werden:

```
// Schleife über LokaleZeilenzahl
   MatSetValues(Matrixname,1,&Zeilenzahl,1,&Spaltenzahl,
     &Wert,INSERT_VALUES)
```

Bevor die einzelnen Werte übergeben werden, ist es möglich, Matrixeigenschaften dem Programm mitzuteilen. Zu solchen Eigenschaften gehört u. a. die Symmetrie. Dies erfolgt mittels

```
MatSetOption(Matrixname,Eigenschaft).
```

Wie bei Vektoren müssen im Folgenden die einzelnen Elemente der Matrix richtig auf die Rechner verteilt werden. Dazu wird ein analoges Funktionspaar

```
MatAssemblyBegin(Matrixname)
```

und

```
MatAssemblyEnd(Matrixname)
```

verwendet. Es ist wiederrum sinnvoll, zwischen beiden Funktionen Berechnungen auszuführen. Innerhalb des Programmes steht eine Vielzahl an Operationen für Matrizen zur Verfügung. Bevor das Programm ordnungsgemäß beendet werden kann, muss der Speicher wieder freigegeben werden. Dazu findet die Funktion

```
MatDestroy(Matrixname)
```

Verwendung.

7.4 Lösen des linearen Gleichungssystems

Beim Lösen eines linearen Gleichungssystems $A\overrightarrow{x} = \overrightarrow{b}$ werden mehrere Funktionen benötigt. Zuerst wird der Löserkontext `SLES` erzeugt. Danach werden die Matrix A und die Vorkonditionierungsmatrix dem Löserkontext übergeben. Als nächstes wird über den Krylovkontext `KSP` die verwendete Krylov-Unterraummethode bestimmt, gefolgt von der Festlegung der Konvergenzkriterien, der Auswahl des Vorkoniditionierers und dem eigentlichen Starten des Lösungsprozesses. Dabei werden noch die Vektoren $\overrightarrow{b}$ und $\overrightarrow{x}$ übergeben. Bevor der Löserkontext wieder gelöscht wird, ist es u.a. möglich, die Norm des Residuums auszulesen. Dies alles ist im folgendem Schema zusammengefasst.

```
SLESCreate(PETSC_COMM_WORLD,&SLES)
SLESSetOperators(SLES,Matrixname,VorkonditionsMatrixname,Eigenschaft)
SLESGetKSP(SLES,&KSP)
KSPSetType(KSP,Krylov-Unterraummethode)
KSPSetTolerances(KSP,rtol,atol,dtol,MaxIterationen)
SLESGetPC(SLES,&PC)
PCSetType(PC,Vorkontitionierer)
SLESSolve(SLES,Vektorb,Vektorx,&Iterationen)
KSPGetResidualNorm(KSP,&ResidualNorm)
SLESDestroy(SLES)
```

7.5 Konvergenzkriterien

Um einen Iterationsprozeß bei ausreichender Genauigkeit abbrechen zu können, müssen Konvergenzkriterien definiert werden. Dazu wird in PETSc die l_2-Norm des Residuums verwendet. Mit Hilfe von 3 Größen wird entschieden, ob Konvergenz oder Divergenz des Algorithmus vorliegt. Innerhalb des Befehls

```
KSPSetTolerances(KSP,rtol,atol,dtol,MaxIterationen)
```

werden diese 3 Größen [`rtol` (relative Abnahme des Residuums), `atol` (absolute Größe des Residuums) und `dtol` (relative Zunahme des Residuums)] übergeben. Anhand des in 7.1

aufgeführten Kriteriums wird Konvergenz ermittelt. Falls ein Algorithmus divergiert, wird Formel 7.2 angewendet. Dabei berechnet sich $\overrightarrow{r_k}$ über die Beziehung $\overrightarrow{r_k} = \overrightarrow{b} - A\overrightarrow{x_k}$.

$$\|\overrightarrow{r_k}\|_2 < \max(rtol * \|\overrightarrow{r_0}\|_2, atol) \tag{7.1}$$

$$\|\overrightarrow{r_k}\|_2 > dtol * \|\overrightarrow{r_0}\|_2 \tag{7.2}$$

Für `rtol`, `atol` und `dtol` existieren folgende Defaultwerte:

- $rtol = 10^{-5}$
- $atol = 10^{-50}$
- $dtol = 10^{5}$

7.6 Zugriff auf die verteilte Lösung

Nach dem Lösen des Gleichungssystems befindet sich die Lösung verteilt auf allen beteiligten Rechnern. Um weiterführende Berechnungen auf einem Rechner durchzuführen, muss die Lösung eingesammelt und in einem lokalen Vektor gespeichert werden. Dazu dient folgendes Schema:

```
VecCreateSeq(PETSC_COMM_SELF,Zeilenzahl,&seq_x)
// Schleife über Zeilenzahl
   IndexScatter[i]=i;
ISCreateGeneral(PETSC_COMM_SELF,Zeilenzahl,IndexScatter,&IS)
VecScatterCreate(x,IS,seq_x,IS,&SCATTER)
VecScatterBegin(x,seq_x,INSERT_VALUES,SCATTER_FORWARD,SCATTER)
VecScatterEnd(x,seq_x,INSERT_VALUES,SCATTER_FORWARD,SCATTER)
ISDestroy(IS)
VecScatterDestroy(SCATTER)
```

Zuerst wird für einen sequentiellen Vektor der Speicherplatz reserviert. Als nächstes muss ein Index über die gesamte Vektorlänge erzeugt werden. Mit diesem ist es möglich, die verteilte Lösung `x` in den sequentiellen Lösungsvektor `seq_x` zu schreiben. Zum Schluss werden die nicht mehr gebrauchten Kontexte gelöscht. Nun ist die Lösung in dem sequentiellen Vektor vorhanden und kann für weitere Rechnungen benutzt werden. Am Ende des Programmes sollte aber nicht vergessen werden, den reservierten Speicherplatz für die verteilte Lösung und die sequentielle Lösung freizugeben.

8 Skalierungstests

Für die folgenden Tests wurde das Programm *GEO4D* unter Einsatz der Bibliothek PETSc umgeschrieben. In Börner et al. (1998) sind die Ansätze des Programmes zur Lösung des geoelektrischen Vorwärtsproblems näher beschrieben. Für die verschiedenen Untersuchungen wurde ein dreidimensionales Modell einer Dikestruktur erzeugt. Die Ausdehnung des Dike wurde während der gesamten Tests nicht verändert (siehe Tabelle 8.1). Für die verschiedenen Untersuchungen wurden jeweils die Größe des inneren äquidistanten Gitters und die Diskretisierungsabstände abgeändert. Auf die jeweiligen Zahlenwerte wird in den entsprechenden Abschnitten näher eingegangen, wobei sich die Angaben zur Ausdehnung in x-, y-, z-Richtung auf das innere äquidistante Gitter beziehen. Um dieses Gitter werden 3 sich um einen Faktor vergrößernde Zellen erzeugt. Ebenso wie der spezifische elektrische Widerstand des Dike von $\rho = 10\,\Omega \cdot m$ konstant blieb, lag der spezifische elektrische Widerstand des Hintergrundes konstant bei $\rho = 100\,\Omega \cdot m$. In Abbildung 8.1 ist eine schematische Darstellung des Dike zu sehen. Bei allen Tests lagen die Einspeisungspunkte bei $A = (0, 0, 0)$ und $B = (20, 0, 0)$.

Richtung	**Ausdehnung [m]**
x	60 bis 70
y	$-\infty$ bis ∞
z	$-\infty$ bis 0

Tabelle 8.1: Position des Dike

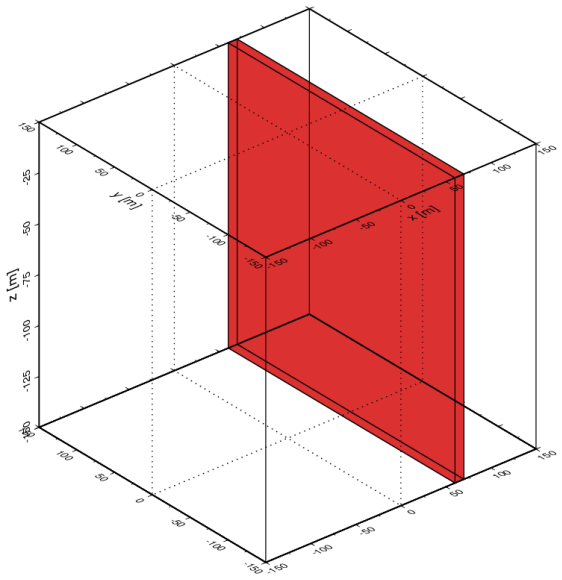

Abbildung 8.1: Modell der dreidimensionalen Dikestruktur

8.1 Bestimmung des Konvergenzkriteriums

Wie in Gleichung 7.1 eingeführt, existieren bei PETSc zwei Kriterien zur Ermittlung von Konvergenz. Beim Vergleich dieser Kriterien mit dem aus LASPack[4] stellte sich heraus, dass als Abbruchkriterium die Größe *rtol* Verwendung finden muss.

$$\|\overrightarrow{r_k}\|_2 < rtol * \|\overrightarrow{r_0}\|_2 \tag{8.1}$$

$$\|\overrightarrow{b} - A\overrightarrow{x_k}\|_2 < rtol * \|\overrightarrow{b} - A\overrightarrow{x_0}\|_2 \tag{8.2}$$

Durch Genauigkeitstests musste herausgefunden werden, ob der in *GEO4D* verwendete Wert von 10^{-3} weiterhin benutzt werden kann. Da für den Dike eine analytische Lösung existiert, wurde dieses Modell wieder als Grundlage genommen. Die Dimensionen in den verschiedenen Richtungen und der Diskretisierungsabstand sind Tabelle 8.2 zu entnehmen. Für die

Modellgröße	**x-, y-Richtung [m]**	**z-Richtung [m]**	**Δ [m]**	**Unbekannte**
$71 \times 71 \times 34$	-320 bis 320	0 bis 300	10	171 394

Tabelle 8.2: Modellgröße — Bestimmung von *rtol*

Untersuchung des Abbruchkriteriums wurde eine Schlumberger-Sondierung simuliert. Die Ergebnisse der Schlumberger-Sondierung werden in Tabelle 8.3 zusammengefasst. Um Konvergenz zu erreichen, muss das Residuum $\overrightarrow{r_k}$ unter den Wert von $rtol * \|\overrightarrow{b}\|_2$ sinken (dies gilt nur, falls als Startvektor $\overrightarrow{x_0}$ ein Nullvektor verwendet wird). Die Anzahl der beteiligten Rechner war auf zwei beschränkt. Der Fehler ε wurde mittels der Formel 8.3 berechnet.

$$\varepsilon = \frac{\rho_{s_{numerisch}} - \rho_{s_{analytisch}}}{\rho_{s_{analytisch}}} * 100 \tag{8.3}$$

Die Zahl der benötigten Iterationsschritte stieg von $rtol = 10^{-1}$ bis $rtol = 10^{-5}$ über 2, 7, 50, 82, auf 94, wobei als Gleichungslöser das CG-Verfahren und als Vorkonditionierer Block-Jacobi mit SSOR zum Einsatz kamen. Wie aus Tabelle 8.3 zu entnehmen ist, verbessert sich der Fehler für dieses Dikemodell zwischen 10^{-3} und 10^{-5} nur in geringem Maße. Für die weiteren Tests, bei denen ausschließlich dieses Modell zum Einsatz kam, wurde das Abbruchkriterium auf $rtol = 10^{-3}$ gesetzt. Weiterführende Genauigkeitsuntersuchungen für dieses Programm sind in Günther (1998) enthalten. Im originalen *GEO4D*-Programmcode wurden lediglich die Gleichungslöser und die dazugehörigen Matrix- und Vektorspeicherroutinen ausgetauscht. An der Diskretisierung wurde nichts verändert. Damit sollten die erreichten Genauigkeiten des abgeändertes Programms mit denen des Originals übereinstimmen. Für in der Praxis relevante Aufgaben ist die mit dem Wert für $rtol = 10^{-3}$ erzielte Genauigkeit ausreichend. Allerdings muss für detailliertere Untersuchungen dieser auf $rtol = 10^{-6}$ verringert werden.

[4] Package for Linear Algebra with Sparse Matrices

$\frac{AB}{2}$		**rtol** $= 10^{-1}$		**rtol** $= 10^{-2}$		**rtol** $= 10^{-3}$		**rtol** $= 10^{-4}$		**rtol** $= 10^{-5}$	
	$\rho_{s_{ana}}$	$\rho_{s_{num}}$	ε	$\rho_{s_{num}}$	ε	$\rho_{s_{num}}$	ε	$\rho_{s_{num}}$	ε	$\rho_{s_{num}}$	ε
$[m]$	$[\Omega \cdot m]$	$[\Omega \cdot m]$	[%]	$[\Omega \cdot m]$	[%]	$[\Omega \cdot m]$	[%]	$[\Omega \cdot m]$	[%]	$[\Omega \cdot m]$	[%]
30	96,6465	96,8006	0,16	96,5779	-0,07	96,6005	-0,05	96,6005	-0,05	96,6005	-0,05
40	89,7649	89,7787	0,01	89,6727	-0,10	89,6737	-0,10	89,6737	-0,10	89,6737	-0,10
50	72,8746	74,2218	1,85	72,9837	0,15	72,9794	0,14	72,9793	0,14	72,9793	0,14
60	80,1007	83,0964	3,74	80,2704	0,21	80,2683	0,21	80,2680	0,21	80,2680	0,21
70	81,8092	86,5737	5,82	81,7578	-0,06	81,7843	-0,03	81,7837	-0,03	81,7836	-0,03
80	83,3146	88,4166	6,12	83,3371	0,03	83,2158	-0,12	83,2150	-0,12	83,2150	-0,12
90	84,6484	90,1439	6,49	84,6713	0,02	84,5257	-0,14	84,5250	-0,15	84,5250	-0,15
100	85,8358	91,7334	6,87	85,8018	0,04	85,7106	-0,15	85,7101	-0,15	85,7102	-0,15
200	92,8855	99,6492	7,28	93,4055	0,24	92,8675	-0,02	92,8584	-0,03	92,8596	-0,03
300	95,8961	99,9993	4,28	97,5667	1,74	96,1636	0,28	96,1463	0,26	96,1497	0,26

Tabelle 8.3: Schlumberger-Sondierung — Bestimmung von *rtol*

8.2 Bestimmung des optimalen iterativen Gleichungslösers

Um den geeignetsten iterativen Gleichungslöser zu finden, mussten für alle Löser gleiche Ausgangsbedingungen geschaffen werden. Es wurde die in Tabelle 8.4 aufgeführte Modellgeometrie für alle iterativen Verfahren genutzt. Neben dem Konvergenzkriterium 8.1 mit

Modellgröße	**x-, y-Richtung** [m]	**z-Richtung** [m]	**Δ** [m]	**Unbekannte**
$91 \times 91 \times 60$	-420 bis 420	0 bis 280	10 in x, y; 5 in z	496 860

Tabelle 8.4: Modellgröße — Vergleich der Krylov-Unterraummethoden

$rtol = 10^{-3}$ wurde zusätzlich die Zahl der Iterationsschritte auf 500 beschränkt. Für alle Verfahren wurde als Vorkonditionierer das Jacobi-Verfahren eingesetzt. Zur Ermittlung statistisch gesicherter Ergebnisse wurden die einzelnen Tests jeweils 100-mal durchgeführt. Die Ergebnisse des rund $32,5\,h$ andauernden Tests sind in Tabelle 8.5 aufgeführt, sowie in Abbildung 8.2 veranschaulicht. Die Ergebnisse zeigen, dass die Verfahren BiCGSTAB, CR und GMRES das lineare Gleichungssystem sehr schnell lösen können, weil sie nur wenige Iterationsschritte benötigen. Betrachtet man aber die Zeit pro Iteration, so zeigt die Methode der konjugierten Gradienten (CG) die größte Konvergenzgeschwindigkeit. Es fällt auch auf, dass viele Verfahren (Richardson, Chebychev, LSQR, TCQMR) mit dem verwendeten Vorkonditionierer in 500 Iterationsschritten keine Konvergenz erreichen. Damit müssen diese für die weiteren Untersuchungen nicht mehr betrachtet werden. Da die Matrix A symmetrisch und positiv definit ist sowie die Ergebnisse des Tests nicht gegen die Verwendung des CG-Verfahrens als iterativen Gleichungslöser sprechen, ist für alle folgenden Untersuchungen die Methode der konjugierten Gradienten angewandt worden. Die Ergebnisse lassen vermuten, dass möglicherweise andere iterative Verfahren schneller sind. Dazu muss bemerkt werden, dass das CG-Verfahren stark von einer effektiven Vorkonditionierung abhängt und deshalb

Verfahren	**Iterationen**	$\bar{\mathbf{t}}$ [s]	$\mathbf{s}_{\bar{\mathbf{t}}}$ [s]	**Konvergenz**
BiCGSTAB	89	28,7	0,1	ja
CR	155	32,0	6,9	ja
GMRES	155	36,2	0,1	ja
CG	347	56,2	1,0	ja
Richardson	500	75,1	0,4	nein
Chebychev	500	78,9	0,2	nein
BiCG	347	106,0	0,6	ja
CGS	435	139,8	0,3	ja
TFQMR	422	145,6	0,8	ja
LSQR	500	148,2	0,6	nein
TCQMR	500	265,4	0,5	nein

Tabelle 8.5: Rechenzeit und Konvergenzverhalten verschiedener Krylov-Unterraummethoden

mit einem besseren Vorkonditionierer die Zeit zum Lösen des Gleichungssystems wesentlich verringert werden kann. Im Rahmen dieser Studienarbeit ist es nicht mehr möglich gewesen, diesen Aspekt näher zu beleuchten. In einer späteren Arbeit sollte dies nachgeholt werden.

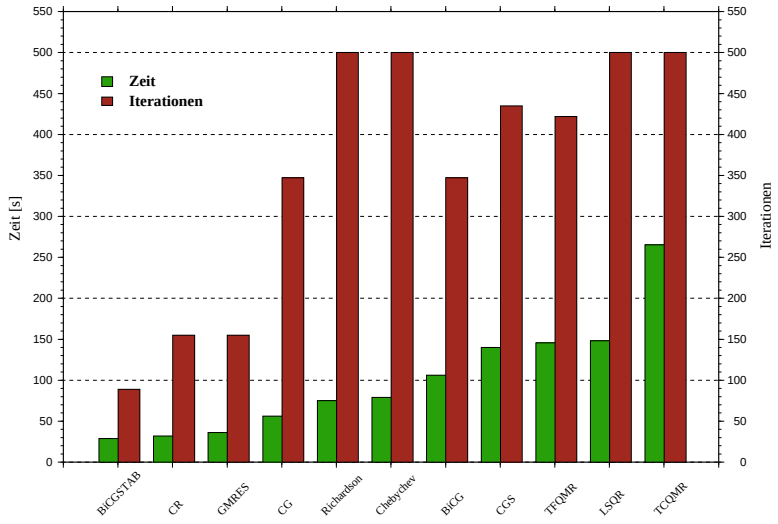

Abbildung 8.2: Vergleich von Laufzeit und Iterationszahl verschiedener Krylov-Unterraummethoden mit Jacobi-Vorkonditionierer

8.3 Vergleich verschiedener Vorkonditionierer

Wie bereits in Abschnitt 8.2 erwähnt, hängt die Konvergenz des CG-Verfahrens stark von einer effektiven Vorkonditionierung ab. Darum soll in diesem Abschnitt ein möglichst guter und einfach anzuwendender Vorkonditionierer gefunden werden. Dazu wurde das in Tabelle 8.6 aufgeführte Dikemodell benutzt. Auf Grund der fast 33 h Laufzeit des letzten Tests und

Modellgröße	**x-, y-Richtung [m]**	**z-Richtung [m]**	**Δ [m]**	**Unbekannte**
$91 \times 91 \times 44$	-420 bis 420	0 bis 400	10	364 364

Tabelle 8.6: Modellgröße — Vergleich der Vorkonditionierer

der Erkenntnis, dass bereits stabile Mittelwerte nach weitaus weniger Durchläufen erreicht werden, wurde die Zahl der zu absolvierenden Testläufe auf 20 reduziert. Zum Einsatz kamen

		Zahl der Prozessoren P						
		1	**2**	**3**	**4**	**5**	**6**	**7**
BJacobi mit SSOR	n	43	48	48	47	49	51	54
	$\overline{t}_n(P)$ $[s]$	0,84	0,51	0,36	0,31	0,27	0,24	0,22
	$s_{\overline{t}_n}(P)$ $[s]$	0,005	0,001	0,001	0,001	0,001	0,001	0,002
	$S_{n_{par}}(P)$	1,00	1,65	2,33	2,71	3,11	3,50	3,82
	$E_{n_{par}}(P)$	1,00	0,83	0,78	0,68	0,62	0,58	0,55
BJacobi mit ILU	n	32	65	76	89	88	95	98
	$\overline{t}_n(P)$ $[s]$	0,62	0,37	0,27	0,23	0,21	0,18	0,18
	$s_{\overline{t}_n}(P)$ $[s]$	0,01	0,001	0,001	0,001	0,001	0,001	0,001
	$S_{n_{par}}(P)$	1,00	1,68	2,26	2,65	2,90	3,39	3,39
	$E_{n_{par}}(P)$	1,00	0,84	0,75	0,66	0,58	0,56	0,48
BJacobi mit Jacobi	n	-	-	-	184	-	-	-
	$\overline{t}_n(P)$ $[s]$	-	-	-	0,14	-	-	-
	$s_{\overline{t}_n}(P)$ $[s]$	-	-	-	0,001	-	-	-
ASM mit SSOR	n	43	58	53	53	53	53	54
	$\overline{t}_n(P)$ $[s]$	0,96	0,61	0,49	0,42	0,41	0,38	0,37
	$s_{\overline{t}_n}(P)$ $[s]$	0,003	0,001	0,002	0,002	0,001	0,001	0,001
	$S_{n_{par}}(P)$	1,00	1,57	1,96	2,29	2,34	2,53	2,59
	$E_{n_{par}}(P)$	1,00	0,79	0,65	0,57	0,47	0,42	0,37
ASM mit ILU	n	32	40	41	41	43	43	43
	$\overline{t}_n(P)$ $[s]$	0,78	0,50	0,41	0,36	0,36	0,32	0,32
	$s_{\overline{t}_n}(P)$ $[s]$	0,001	0,004	0,003	0,001	0,002	0,002	0,002
	$S_{n_{par}}(P)$	1,00	1,44	1,90	2,17	2,17	2,44	2,44
	$E_{n_{par}}(P)$	1,00	0,72	0,63	0,54	0,43	0,41	0,35
ASM mit Jacobi	n	-	-	-	222	-	-	-
	$\overline{t}_n(P)$ $[s]$	-	-	-	0,22	-	-	-
	$s_{\overline{t}_n}(P)$ $[s]$	-	-	-	0,001	-	-	-

Tabelle 8.7: Vergleich der unterschiedlichen Vorkonditionierer

diesmal alle zur Verfügung stehenden Rechner/Prozessoren. In Tabelle 8.7 sind die Resultate zusammengestellt. Die Ergebnisse werden in den Abbildungen 8.3 und 8.4 veranschaulicht. Bei den Testläufen zeigte sich nicht die zu erwartende Unabhängigkeit der Iterationszahl von der Prozessoranzahl. Dies ist auf die unterschiedlich gute Vorkonditionierung der einzelnen Blöcke (1 Block je Prozessor) zurückzuführen. Mit schwankenden Iterationszahlen lassen sich jedoch keine zuverlässigen Vergleiche durchführen, so dass für die Auswertung die Zeit pro Iteration t_n herangezogen wird. Daher müssen die Formeln 4.1 und 4.4 modifiziert werden. Mit den folgenden beiden Gleichungen wurden der parallele Speedup und die parallele Effizienz berechnet:

$$S_{n_{par}}(P) = \frac{t_n(1)}{t_n(P)} \tag{8.4}$$

$$E_{n_{par}}(P) = \frac{t_n(1)}{P \cdot t_n(P)} = \frac{S_n(P)}{P}. \tag{8.5}$$

In Abbildung 8.3 zeigt sich, dass der Block-Jacobi-Vorkonditionierer schneller als der ASM-Vorkonditionierer ist. Es wird ebenfalls ersichtlich, dass zwischen den Vorkonditionierern für jeden Block (sprich SSOR und ILU) nur geringe Unterschiede bestehen, diese sich aber

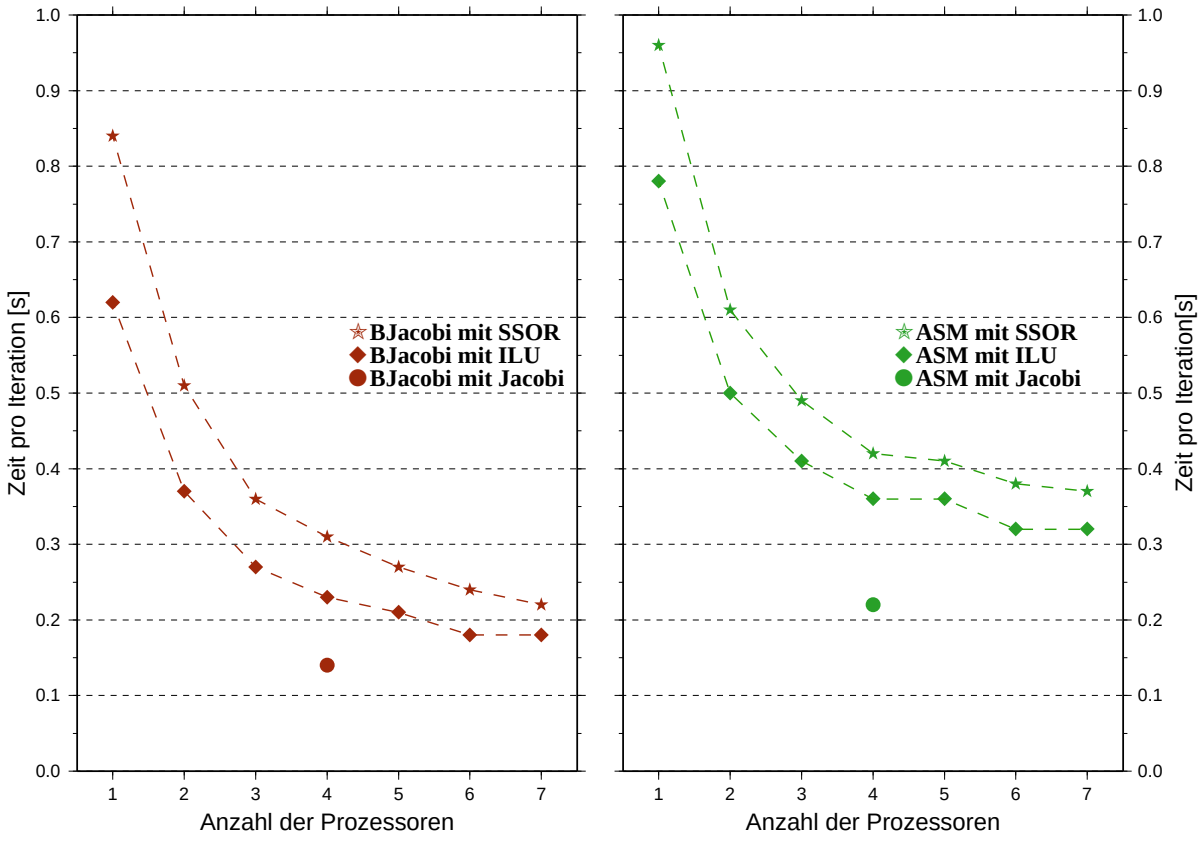

Abbildung 8.3: Vergleich der unterschiedlichen Vorkonditionierer anhand der Zeit pro Iteration

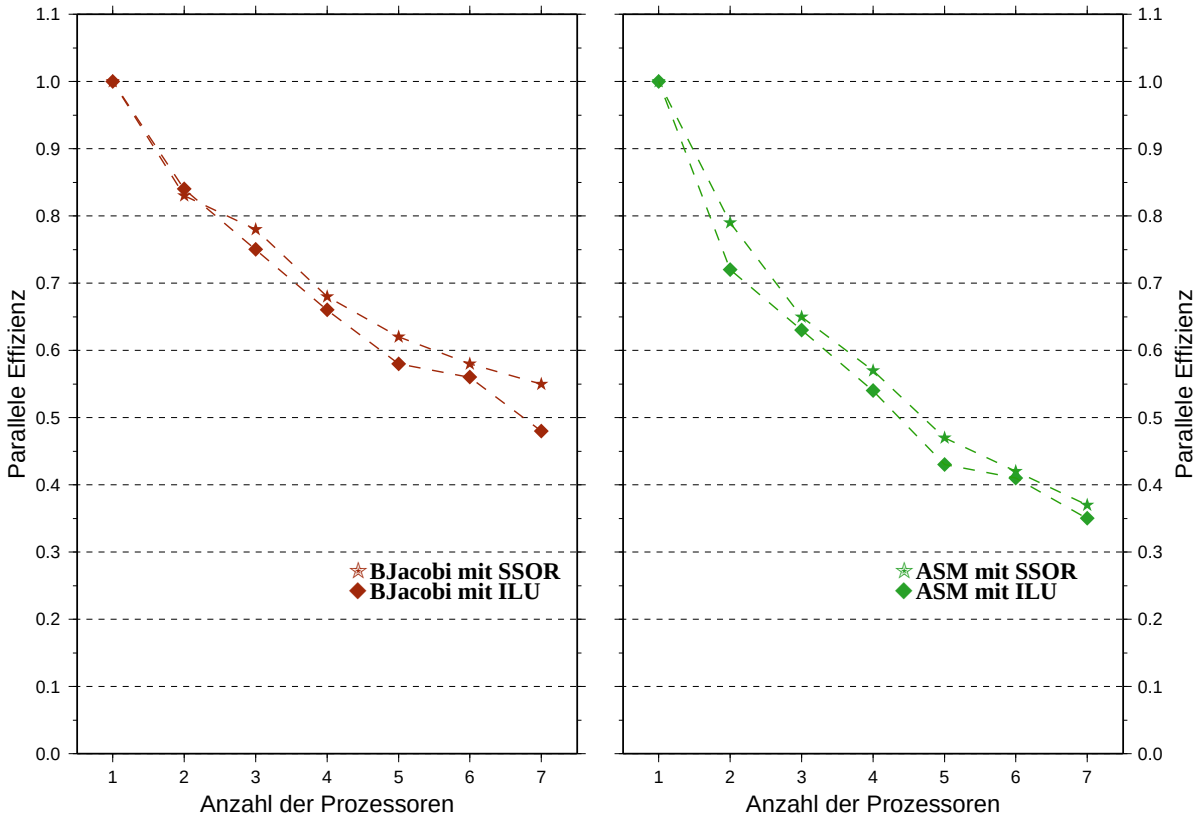

Abbildung 8.4: Vergleich der unterschiedlichen Vorkonditionierer anhand der Effizienz

bereits auf die Effizienz auswirken (siehe Abbildung 8.4). Es ist ebenso erkennbar, dass der Jacobi-Vorkonditionierer die höchste Konvergenzgeschwindigkeit herbeiführt. Auf Grund der großen Iterationszahl ($n = 184$ und $n = 222$) ist er für die weitere Verwendung nicht geeignet. In Abbildung 8.4 erkennt man die höhere Effizienz mit guter Vorkonditionierung (BJacobi im Vergleich zu ASM). Mit steigender Prozessorzahl sinkt diese wieder. Dieses Verhalten kann auf die Modellgröße zurückgeführt werden, denn mit steigender Rechnerzahl nimmt das Verhältnis zwischen Kommunikationszeit und Rechenzeit zu, so dass die Gesamtlaufzeit des Algorithmus zunehmend durch die Kommunikationszeit bestimmt wird. Mit steigender Gesamtlaufzeit sinkt wieder der Speedup und damit die Effizienz. Im Vergleich zeigt sich, dass die Vorkonditionierung mit Block-Jacobi und SSOR als Blockvorkonditionierer effektiv ist. Diese Art der Vorkonditionierung ist leicht anzuwenden und wird im weiteren Verlauf der Skalierungstests genutzt.

8.4 Einfluss der Modellgröße

Nachdem in den vorangegangenen Abschnitten die günstigste Kombination aus Vorkonditionierer und Gleichungslöser gefunden worden ist, soll in diesem Abschnitt der Einfluss der Modellgröße auf den Speedup und damit die Effizienz untersucht werden. Dazu kam wieder das bekannte Dikemodell zum Einsatz. Während der Tests wurde das innere äquidistante Gitter verändert. In Tabelle 8.8 werden die verwendeten Modelle mit ihren Gittereigenschaften und der Zahl der Unbekannten aufgelistet. Mit diesen Modellen und allen verfügbaren

Modellgröße	**x-, y-Richtung [m]**	**z-Richtung [m]**	**Δ [m]**	**Unbekannte**
$51 \times 51 \times 24$	-220 bis 220	0 bis 200	10	62 424
$71 \times 71 \times 34$	-320 bis 320	0 bis 300	10	171 394
$91 \times 91 \times 44$	-420 bis 420	0 bis 400	10	364 364
$111 \times 111 \times 54$	-520 bis 520	0 bis 500	10	665 334
$131 \times 131 \times 64$	-620 bis 620	0 bis 600	10	1 098 304
$151 \times 151 \times 74$	-720 bis 720	0 bis 700	10	1 687 274

Tabelle 8.8: Modellgrößen — Einfluss der Modellgröße

Rechnern wurden 20 Durchläufe für jede Modellgröße gerechnet. Die Ergebnisse werden in Tabelle 8.9 aufgeführt. Die Werte für den parallelen Speedup $S_{n_{par}}$ und die parallele Effizienz $E_{n_{par}}$ wurden nach den Gleichungen 8.4 und 8.5 berechnet. Es zeigt sich wieder der bekannte Effekt der Änderung der Iterationszahl mit der Zahl der Prozessoren. Dieses Verhalten lässt sich durch die Funktionsweise des Vorkonditionierers erklären. Da auf jeden Block der Matrix (wobei sich ein Block auf einem Prozessor befindet) der SSOR-Vorkonditionierer angewendet wird und nicht jeder Block auf Grund der Diskretisierung zwingend die gleiche Matrixstruktur besitzt, ergeben sich unterschiedlich gut vorkonditionierte Blöcke, die dann im unterschiedlichem Lösungsverhalten des linearen Gleichungssystems resultieren. In Abbildung 8.5 links wird der Einfluss der Modellgröße auf die Zeit pro Iteration graphisch dargestellt. Es ist zu erkennen, dass mit zunehmender Modellgröße die Zeit pro Iteration ansteigt, da die Größe der Matrix zunimmt und damit der numerische Aufwand wächst. Der Zeitgewinn pro Iteration bei großen Modellen ist am deutlichsten. Dieser Umstand kann wieder anhand des Verhältnisses zwischen Kommunikationszeit und Rechenzeit aufgeklärt werden, denn bei großen Modellen müssen pro Iteration mehr Operationen durchgeführt werden, was zu einer Erhöhung der Rechenzeit führt. Da sich aber die Kommunikationszeit nur gering verändert, wird das Verhältnis günstiger für große Modelle. Das resultiert in einem bedeutenden Zeitgewinn beim Einsatz mehrerer Rechner. Im rechten Teil der Abbildung 8.5 wird die parallele Effizienz über der Rechneranzahl aufgetragen. Man erkennt, dass die Effizienz für kleine Modelle scheinbar gut ist. Für diese Modelle schwanken die Iterationszeiten teilweise erheblich, was sich in einer hohen Standardabweichung ausdrückt. Damit reicht die Genauigkeit der Zeitmessung nicht mehr aus, um die Änderung der Rechenzeit für steigende Prozessoranzahl

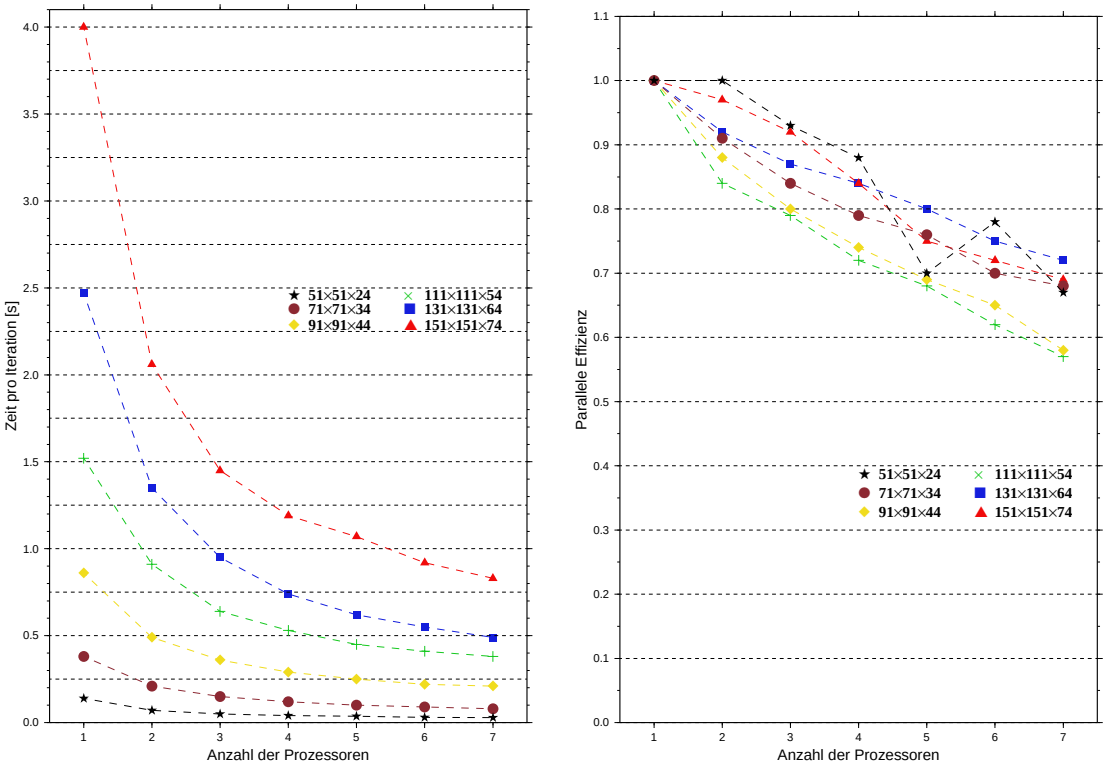

Abbildung 8.5: Vergleich unterschiedlicher Modellgrößen anhand der Zeit pro Iteration und der Effizienz

korrekt wiederzugeben. Unter Berücksichtigung dieses Sachverhaltes ist es nicht möglich, die Effizienz exakt zu berechnen. Für kleine Modelle ist die Interpretation der Effizienzwerte nicht sinnvoll. An den Modellgrößen $111\times111\times54$ bis $151\times151\times74$ kann abgelesen werden, dass die parallele Effizienz mit der Modellgröße wächst. Dies spiegelt den bereits erwähnten Sachverhalt, Verhältnis zwischen Kommunikationszeit und Rechenzeit, wider. Modellgrößen über $151\times151\times74$ konnten auf Grund der unterschiedlichen Arbeitsspeicherausstattung nicht auf allen beteiligten Rechnern berechnet werden. Damit musste auf die Einbeziehung dieser Ergebnisse in die Auswertung verzichtet werden.

		Zahl der Prozessoren P						
		1	**2**	**3**	**4**	**5**	**6**	**7**
51×51×24	n	37	50	63	71	76	77	80
	$\overline{t}_n(P)$ $[s]$	0,14	0,07	0,05	0,04	0,04	0,03	0,03
	$s_{\overline{t}_n}(P)$ $[s]$	0,0002	0,0002	0,0004	0,0002	0,0010	0,0004	0,0010
	$S_{n_{par}}(P)$	1,00	2,00	2,80	3,50	3,50	4,67	4,67
	$E_{n_{par}}(P)$	1,00	1,00	0,93	0,88	0,70	0,78	0,67
71×71×34	n	41	53	58	63	58	61	81
	$\overline{t}_n(P)$ $[s]$	0,38	0,21	0,15	0,12	0,10	0,09	0,08
	$s_{\overline{t}_n}(P)$ $[s]$	0,01	0,0004	0,001	0,001	0,001	0,001	0,001
	$S_{n_{par}}(P)$	1,00	1,81	2,53	3,17	3,80	4,22	4,75
	$E_{n_{par}}(P)$	1,00	0,91	0,84	0,79	0,76	0,70	0,68
91×91×44	n	43	48	48	47	49	51	54
	$\overline{t}_n(P)$ $[s]$	0,86	0,49	0,36	0,29	0,25	0,22	0,21
	$s_{\overline{t}_n}(P)$ $[s]$	0,024	0,001	0,001	0,002	0,003	0,001	0,01
	$S_{n_{par}}(P)$	1,00	1,76	2,39	2,97	3,44	3,91	4,10
	$E_{n_{par}}(P)$	1,00	0,88	0,80	0,74	0,69	0,65	0,58
111×111×54	n	38	49	38	40	51	53	55
	$\overline{t}_n(P)$ $[s]$	1,52	0,91	0,64	0,53	0,45	0,41	0,38
	$s_{\overline{t}_n}(P)$ $[s]$	0,02	0,001	0,001	0,003	0,005	0,02	0,007
	$S_{n_{par}}(P)$	1,00	1,67	2,38	2,87	3,38	3,71	4,00
	$E_{n_{par}}(P)$	1,00	0,84	0,79	0,72	0,68	0,62	0,57
131×131×64	n	31	39	40	22	22	22	24
	$\overline{t}_n(P)$ $[s]$	2,47	1,35	0,95	0,74	0,62	0,55	0,49
	$s_{\overline{t}_n}(P)$ $[s]$	0,001	0,004	0,002	0,01	0,01	0,01	0,02
	$S_{n_{par}}(P)$	1,00	1,83	2,60	3,34	3,98	4,49	5,04
	$E_{n_{par}}(P)$	1,00	0,92	0,87	0,84	0,80	0,75	0,72
151×151×74	n	16	18	17	17	18	19	19
	$\overline{t}_n(P)$ $[s]$	4,00	2,06	1,45	1,19	1,07	0,92	0,83
	$s_{\overline{t}_n}(P)$ $[s]$	0,11	0,004	0,006	0,01	0,03	0,02	0,02
	$S_{n_{par}}(P)$	1,00	1,94	2,76	3,36	3,74	4,35	4,81
	$E_{n_{par}}(P)$	1,00	0,97	0,92	0,84	0,75	0,72	0,69

Tabelle 8.9: Einfluss der Modellgröße auf die Skalierung

8.5 Einfluss des Diskretisierungabstandes

Um dem Einfluss des Diskretisierungsabstandes näher zu untersuchen, mussten Modelle gleicher Modellgröße und unterschiedlicher Gitterabstände erstellt werden. Als Grundlage diente wieder das Dikemodell 8.1. Die verwendeten Modelle sind in Tabelle 8.10 aufgeführt. Die Er-

Modellgröße	**x-, y-Richtung [m]**	**z-Richtung [m]**	**Δ [m]**	**Unbekannte**
$131 \times 131 \times 64$	-620 bis 620	0 bis 600	10	1 098 304
$131 \times 131 \times 64$	-310 bis 310	0 bis 300	5	1 098 304
$131 \times 131 \times 64$	-124 bis 124	0 bis 120	2	1 098 304

Tabelle 8.10: Modellgrößen — Einfluss des Diskretisierungsabstandes

gebnisse der 20 Durchläufe pro Modellgröße sind in Tabelle 8.11 aufgeführt. Aus diesen Ergebnissen sind die Abbildungen 8.6 bis 8.8 erstellt worden. In Tabelle 8.11 erkennt man das unterschiedliche Verhalten der Iterationszahl mit der Anzahl der Rechner. Bei der Diskretisierung $\Delta = 10\,m$ steigt die Iterationszahl bis 3 Prozessoren konstant an, springt dann auf nahezu die Hälfte herab und bleibt dann relativ konstant. Bei den Diskretisierungen mit $\Delta = 5\,m$ und $\Delta = 2\,m$ steigt die Iterationszahl mit der Zahl der Rechner konstant an. Dieses unterschiedliche Verhalten für die verschiedenen Gitterabstände ist nicht erklärbar. Bei allen Modellen sinkt die Gesamtrechenzeit mit der Prozessorenanzahl, trotz Anstei-

		Zahl der Prozessoren P						
		1	**2**	**3**	**4**	**5**	**6**	**7**
$\Delta = 10$m	n	31	39	40	22	22	22	24
	$\overline{t}(P)$ $[s]$	76,4	52,8	38,0	16,2	13,6	12,0	11,8
	$\overline{t}_n(P)$ $[s]$	2,47	1,35	0,95	0,74	0,62	0,55	0,49
	$s_{\overline{t}_n}(P)$ $[s]$	0,001	0,004	0,002	0,01	0,01	0,01	0,02
	$S_{n_{par}}(P)$	1,00	1,83	2,60	3,34	3,98	4,49	5,04
	$E_{n_{par}}(P)$	1,00	0,92	0,87	0,84	0,80	0,75	0,72
$\Delta = 5$m	n	60	82	90	95	101	106	113
	$\overline{t}(P)$ $[s]$	152,8	110,2	85,1	69,3	61,0	55,4	52,8
	$\overline{t}_n(P)$ $[s]$	2,53	1,34	0,95	0,73	0,60	0,52	0,47
	$s_{\overline{t}_n}(P)$ $[s]$	0,01	0,003	0,001	0,006	0,003	0,004	0,01
	$S_{n_{par}}(P)$	1,00	1,89	2,66	3,47	4,22	4,87	5,38
	$E_{n_{par}}(P)$	1,00	0,94	0,89	0,87	0,84	0,81	0,77
$\Delta = 2$m	n	66	97	108	118	127	132	140
	$\overline{t}(P)$ $[s]$	167,0	130,6	101,8	86,4	76,6	68,9	64,9
	$\overline{t}_n(P)$ $[s]$	2,53	1,35	0,94	0,73	0,60	0,52	0,46
	$s_{\overline{t}_n}(P)$ $[s]$	0,006	0,01	0,002	0,01	0,003	0,01	0,003
	$S_{n_{par}}(P)$	1,00	1,87	2,69	3,47	4,22	4,87	5,50
	$E_{n_{par}}(P)$	1,00	0,94	0,90	0,87	0,84	0,81	0,79

Tabelle 8.11: Einfluss des Diskretisierungsabstandes auf die Skalierung

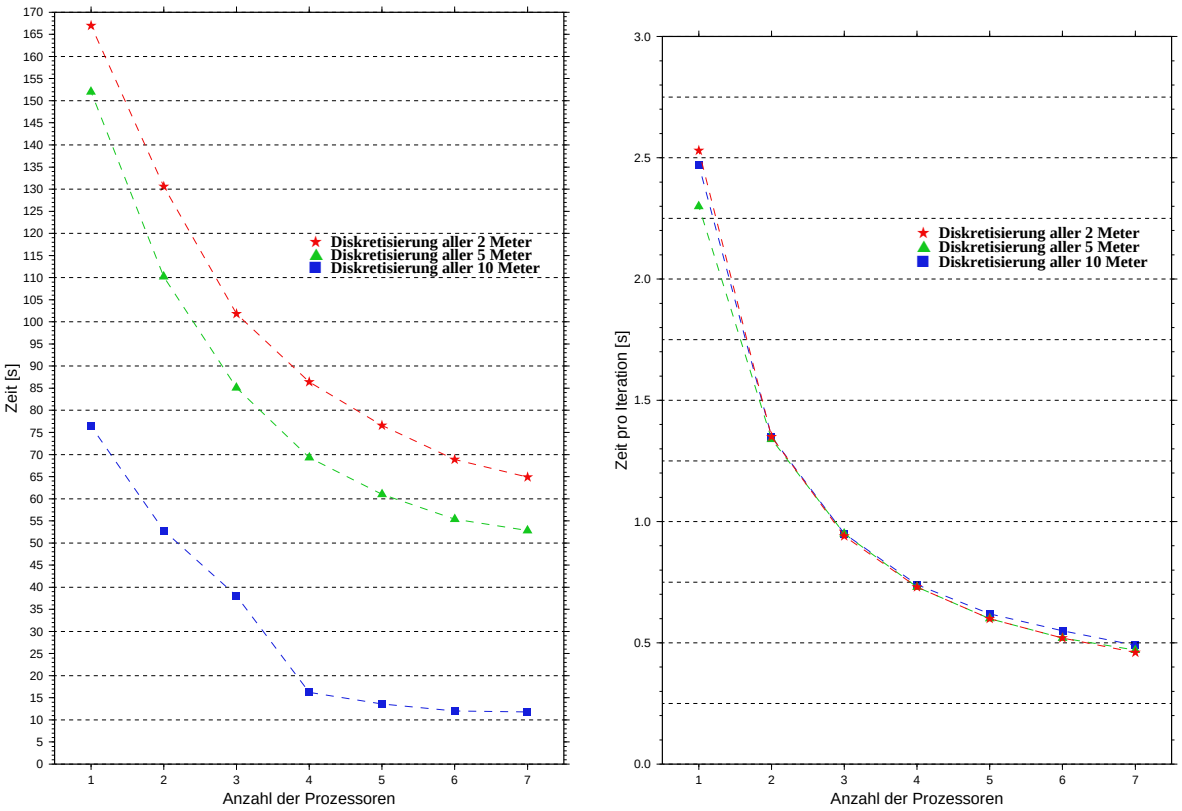

Abbildung 8.6: Vergleich von Gesamtlaufzeit und Zeit pro Iteration für verschiedene Diskretisierungsabstände

gens der Iterationszahl. Dieses Verhalten ist im linken Teil der Abbildung 8.6 dargestellt. Darin ist deutlich der Sprung zwischen 3 und 4 Prozessoren bei der Diskretisierung mit $\Delta = 10\,m$ zu erkennen. Der numerische Aufwand zur Lösung des Gleichungssytems steigt mit der Verringerung des Diskretisierungsabstandes. Damit steigt auch die Gesamtlaufzeit, was in dieser Abbildung ebenfalls zu sehen ist. Da sich die Modellgröße nicht verändert hat, ist die Zeit pro Iteration für alle Modelle gleich, weil die Anzahl der zu lösenden Gleichungen konstant geblieben ist. Dies ist klar in dem rechten Teil der Abbildung 8.6 zu erkennen. Die höhere Gesamtlaufzeit erklärt sich nur aus der höheren Iterationszahl. Da sich die Zeiten pro Iteration nur geringfügig unterscheiden, sind die errechneten Werte für den parallelen Speedup und die parallele Effizienz nur sehr geringen Schwankungen unterlegen. Dies ist in den Abbildungen 8.7 und 8.8 dargestellt.

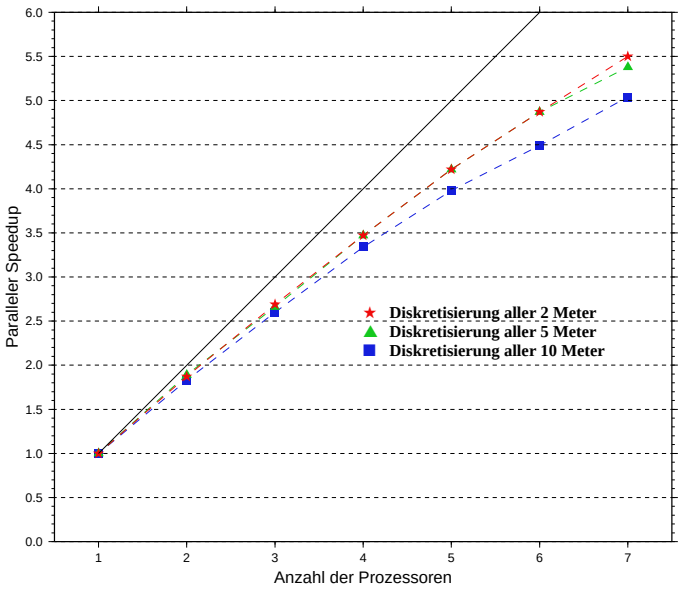

Abbildung 8.7: Vergleich des erreichten Speedup

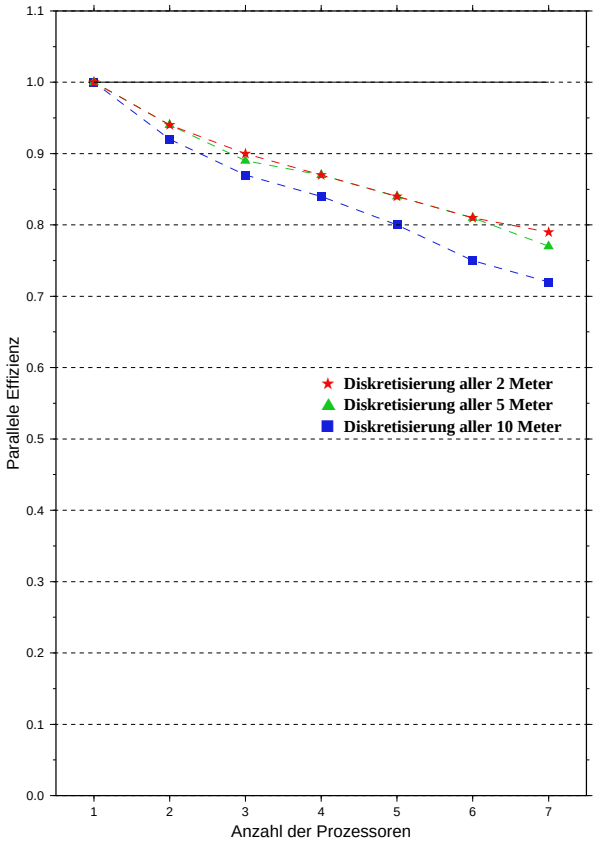

Abbildung 8.8: Vergleich der erreichten Effizienz

8.6 Vergleich zwischen LASPack und PETSc

In diesem Abschnitt soll kurz die Effektivität der Bibliothek PETSc im Vergleich zu der in *GEO4D* verwendeten Bibliothek LASPack untersucht werden. Da bei LASPack nur das sequentielle Lösen eines linearen Gleichungssystems möglich ist, wurde für den Vergleich ein Rechner genutzt. Die Größe des verwendeten Dikemodells ist in Tabelle 8.12 aufgeführt. Aus

Modellgröße	**x-, y-Richtung [m]**	**z-Richtung [m]**	**Δ [m]**	**Unbekannte**
$91 \times 91 \times 44$	-420 bis 420	0 bis 400	10	364 364

Tabelle 8.12: Modellgröße — Vergleich der Bibliotheken

beiden Bibliotheken wurde das CG-Verfahren mit SSOR als Vorkonditionierer verwendet. Der Relaxationskoeffizient wurde für beide Bibliotheken auf $\omega = 1,75$ gesetzt. Die Ergebnisse der 20 Durchläufe auf *matlab* sind in Tabelle 8.13 zusammengefasst. Ob die selben Ausgang-

	n	$\overline{t}\ [s]$	$s_{\overline{t}}\ [s]$	$\overline{t}_n\ [s]$	$s_{\overline{t}_n}\ [s]$
PETSc	80	39,6	0,2	0,495	0,002
Laspack	80	47,4	0,1	0,593	0,001

Tabelle 8.13: Vergleich der Bibliotheken

bedingungen für diese Untersuchung vorgelegen haben, lässt sich anhand der Iterationszahl überprüfen. Diese sollte, wie es hier der Fall ist, gleich sein. Die gemessenen Gesamtlaufzeiten zeigen einen Geschwindigkeitsvorteil für PETSc. Dieser beläuft sich auf gute $7s$ bei der Gesamtlaufzeit und auf $0,1s$ bei einem Iterationsschritt. Die 16% Zeitvorteil bei der sequentiellen Abarbeitung sind nicht zu unterschätzen. Bei Simulation einer flächenhaften Messung sind noch weitaus größere Geschwindigkeitsvorteile zu erwarten. Es ist gezeigt, dass diese Bibliothek im Vergleich zu LASPack effizient arbeitet. Was verbleibt, ist ein Vergleich zwischen anderen Implementierungen und PETSc.

8.7 Fehlerabschätzung

Zu jeder gemessenen Größe wurde bereits in den entsprechenden Tabellen die Standardabweichung für diese mit angegeben. Über das Fehlerfortpflanzungsgesetz kann die Standardabweichung für den parallelen Speedup und die parallele Effizienz berechnet werden. Für eine Ergebnisgröße $y = f(x_1, x_2)$ wird die Standardabweichung $\overline{s}_y$ nach folgender Formel berechnet:

$$\overline{s}_y = \sqrt{\left(\frac{\partial y}{\partial x_1}\,\overline{s}_{x_1}\right)^2 + \left(\frac{\partial y}{\partial x_2}\,\overline{s}_{x_2}\right)^2}. \tag{8.6}$$

Nach den Formeln 8.4, 8.5 und 8.6 berechnen sich die Standardabweichungen für den parallelen Speedup $\overline{s}_{S_{n_{par}}}$ und die parallele Effizienz $\overline{s}_{E_{n_{par}}}$ nach den Gleichungen:

$$\overline{s}_{S_{n_{par}}} = \sqrt{\left(\frac{1}{t_n(P)}\,\overline{s}_{t_n(1)}\right)^2 + \left(\frac{-t_n(1)}{(t_n(P))^2}\,\overline{s}_{t_n(P)}\right)^2} \tag{8.7}$$

und

$$\overline{s}_{E_{n_{par}}} = \sqrt{\left(\frac{1}{P}\,\overline{s}_{S_{n_{par}}}\right)^2} = \frac{\overline{s}_{S_{n_{par}}}}{P}. \tag{8.8}$$

Mit diesen beiden Gleichungen werden beispielhaft die Standardabweichungen $\overline{s}_{S_{n_{par}}}$ und $\overline{s}_{E_{n_{par}}}$ berechnet. Dazu werden die Zeilen für $\Delta = 10m$ aus der Tabelle 8.11 verwendet. Die berechneten Standardabweichungen sind in der Tabelle 8.14 eingefügt. Es wird deutlich, dass

		Zahl der Prozessoren P						
		1	**2**	**3**	**4**	**5**	**6**	**7**
$\Delta = 10$m	n	31	39	40	22	22	22	24
	$\overline{t}(P)$ $[s]$	76,4	52,8	38,0	16,2	13,6	12,0	11,8
	$\overline{t}_n(P)$ $[s]$	2,47	1,35	0,95	0,74	0,62	0,55	0,49
	$s_{\overline{t}_n}(P)$ $[s]$	0,001	0,004	0,002	0,01	0,01	0,01	0,02
	$S_{n_{par}}(P)$	1,00	1,83	2,60	3,34	3,98	4,49	5,04
	$\overline{s}_{S_{n_{par}}}(P)$	0,0006	0,005	0,006	0,05	0,06	0,08	0,10
	$E_{n_{par}}(P)$	1,00	0,92	0,87	0,84	0,80	0,75	0,72
	$\overline{s}_{E_{n_{par}}}(P)$	0,0006	0,003	0,002	0,01	0,01	0,01	0,02

Tabelle 8.14: Fehlerabschätzung

die Unsicherheit bei der Angabe des parallelen Speedup und der parallelen Effizienz maximal in der zweiten Nachkommastelle liegt. Damit rechtfertigt sich auch die Angabe bis zu dieser Genauigkeit. Die Fehler für die anderen berechneten Werte der einzelnen Tabellen sind in der selben Größenordnung zu erwarten, da sich die Standardabweichungen der gemessenen Zeiten nur unwesentlich unterscheiden.

9 Zusammenfassung

Die Lösung großer Gleichungssysteme in technischen und naturwissenschaftlichen Bereichen erfordert zunehmend den Einsatz paralleler Rechentechnik. Anhand der verschiedenen Untersuchungen in dieser Studienarbeit konnte nachgewiesen werden, dass sich der Einsatz eines Parallelrechners und speziell eines Linux-Clusters lohnen kann. Dafür spricht vor allem die erreichbare Zeitersparnis. Daneben zeigt sich auch, dass mit dem Einsatz von Parallelrechnern die Modellgröße der Simulationsrechnung steigen kann und damit die erreichte Genauigkeit zunimmt. Innerhalb der einzelnen Testläufe konnten Erfahrungen über die Parallelisierung von Programmen gesammelt werden. Diese sollen kurz zusammgefasst werden.

- Ein Cluster lässt sich mittels einer Linux-Distribution einfach einrichten.
- Über die verschiedenen Kriterien zur Bewertung der Leistungsfähigkeit paralleler numerischer Algorithmen ließ sich die günstigste Kombination aus Gleichungslöser und Vorkonditionierer als CG-Verfahren mit Block-Jacobi und SSOR bestimmen.
- Über das in PETSc zur Verfügung stehende Abbruchkriterium `rtol` lässt sich die erreichte Genauigkeit steuern. Dabei hat sich für praktische Aufgaben ein Wert von 10^{-3} als ausreichend erwiesen. In Diskussionen hat sich gezeigt, dass für spezielle Untersuchungen mit diesem Wert nicht die notwendige Genauigkeit erreicht wird, so dass mit $rtol = 10^{-6}$ gerechnet werden sollte.
- Mit zunehmender Problemgröße und feiner werdender Diskretisierung steigt der erreichte Speedup und damit die parallele Effizienz. Beim Einsatz von 7 Rechnern berechnet der Gleichungslöser etwa fünfmal so schnell das Ergebnis (im Vergleich zur Einzelrechnervariante).
- Im Vergleich zu anderen Implementierungen iterativer Gleichungslöser ist die verwendete Bibliothek PETSc effizient.

Neben den guten Ergebnissen enstanden während der Studienarbeit auch einige Fragen. Diese sollten in einer späteren Arbeit geklärt werden.

- Wie sind die einzelnen Algorithmen implementiert?
- Warum ist das Verfahren der BiCGSTAB schneller als das CG-Verfahren, obwohl mehr numerische Arbeit geleistet werden muss?
- Gibt es schnellere Kombinationen aus Gleichungslöser und Vorkonditionierer?
- Wieso variiert die Iterationszahl mit der Prozessorzahl teilweise erheblich?
- Inwieweit ist der Einsatz von PETSc-Funktionen zur Visualisierung möglich?

In der Studienarbeit wurde gezeigt, dass der Einsatz von Parallelrechnern innerhalb der Geophysik und speziell der Geoelektrik von Vorteil ist. In Zukunft sollte die Parallelisierung nicht bei den Gleichungslösern aufhören, sondern das gesamte Programm in die Parallelisierung einbezogen werden.

A Glossar

AMD, Athlon, Duron	eingetragene Warenzeichen der Advanced Micro Devices Corporation
Intel, PentiumIII	eingetragene Warenzeichen der Intel Corporation
Linux	eingetragenes Warenzeichen von Linus Torvalds
MATLAB	eingetragenes Warenzeichen der MathWorks Incorporation
SuSE	eingetragenes Warenzeichen der SuSE GmbH

BiCG	BiConjugate Gradient
BiCGSTAB	BiConjugate Gradient Stabilized
CG	Conjugate Gradient
Chebychev	Chebychev
CR	Conjugate Residual
CGS	Conjugate Gradient Squared
GMRES	Generalized Minimal Residual
LSQR	Least Squares Method
Richardson	Richardson
TCQMR	Transpose-Free Quasi-Minimal Residual (2)
TFQMR	Transpose-Free Quasi-Minimal Residual (1)

ASM	Additive-Schwarz-Methode
BJacobi	Block-Jacobi
ILU	Incomplete LU
Jacobi	Jacobi
SSOR	Symmetric Successive OverRelaxation

B Verwendete Formelzeichen

Zeichen	Erklärung
$\overline{\mathbf{t}}$	Mittelwert der Laufzeit
$\mathbf{s}_{\overline{\mathbf{t}}}$	Standardabweichung vom Mittelwert der Laufzeit
$\mathbf{P}$	Prozessoranzahl
$\mathbf{t(1)}$	Laufzeit für 1 Prozessor
$\mathbf{t(P)}$	Laufzeit für P Prozessoren
$\mathbf{S_{par}(P)}$	paralleler Speedup für P Prozessoren
$\mathbf{S_{ges}(P)}$	gesamter Speedup für P Prozessoren
$\mathbf{S_{skal}(P)}$	skalierbarer Speedup für P Prozessoren
$\mathbf{E_{par}(P)}$	parallele Effizienz für P Prozessoren
$\mathbf{E_{ges}(P)}$	gesamte Effizienz für P Prozessoren
$\mathbf{E_{skal}(P)}$	skalierbare Effizienz für P Prozessoren
$\mathbf{S}$	Speedup
$\mathbf{A_s}$	sequentieller Anteil
$\mathbf{A_p}$	paralleler Anteil
`rtol`	relative Abnahme des Residuums
`atol`	absolute Größe des Residuums
`dtol`	relative Zunahme des Residuums
$\overrightarrow{r_k}$	Residuum des k-ten Iterationsschrittes
$\overrightarrow{r_0}$	Residuum des 0-ten Iterationsschrittes
$\overrightarrow{b}$	Quellvektor
$\overrightarrow{x_k}$	Lösung des k-ten Iterationsschrittes
$\overrightarrow{x_0}$	Lösung des 0-ten Iterationsschrittes
$\mathbf{A}$	Koeffizientenmatrix
$\rho_\mathbf{s}$	spezifischer elektrischer Widerstand
ε	relativer Fehler
$\rho_{\mathbf{s_{numersich}}}$, $\rho_{\mathbf{s_{num}}}$	ρ_s numerisch berechnet
$\rho_{\mathbf{s_{analytisch}}}$, $\rho_{\mathbf{s_{ana}}}$	ρ_s analytisch berechnet
$\mathbf{t_n(1)}$	Laufzeit für 1 Prozessor und eine Iteration
$\mathbf{t_n(P)}$	Laufzeit für P Prozessoren und eine Iteration
$\overline{\mathbf{t}}\mathbf{(P)}$	Mittelwert der Laufzeit für P Prozessoren
$\overline{\mathbf{t}}_\mathbf{n}\mathbf{(P)}$	Mittelwert der Laufzeit pro Iteration für P Prozessoren
$\mathbf{s}_{\overline{\mathbf{t}}_\mathbf{n}}\mathbf{(P)}$	Standardabweichung vom Mittelwert der Laufzeit pro Iteration für P Prozessoren
$\mathbf{n}$	Iterationszahl
$\mathbf{S_{n_{par}}(P)}$	paralleler Speedup für P Prozessoren und eine Iteration
$\mathbf{E_{n_{par}}(P)}$	parallele Effizienz für P Prozessoren und eine Iteration
ω	Überrelaxationsfaktor
$\overline{s}_{S_{n_{par}}}(P)$	Standardabweichung des parallelen Speedup für P Prozessoren und eine Iteration
$\overline{s}_{E_{n_{par}}}(P)$	Standardabweichung der parallelen Effizienz für P Prozessoren und eine Iteration

Literatur

Ashton, D., Chan, A., Gropp, B., Lusk, R., Ross, R., Thakur, R. und Toonen, B. (2000). *MPI - Message Passing Interface.* http://www-unix.mcs.anl.gov/mpi/mpich.

Balay, S., Gropp, W. D., McInnes, L. C. und Smith, B. F. (1998). *PETSc home page.* (http://www-fp.mcs.anl.gov/petsc/)

Balay, S., Gropp, W. D., McInnes, L. C. und Smith, B. F. (2001). *PETSc Users Manual* (Tech. Rep. No. ANL-95/11 - Revision 2.1.0). Argonne National Laboratory.

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. und van der Vorst, H. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* Philadelphia, PA: SIAM.

Börner, R.-U., Günther, T. und Käppler, R. (1998). 3D-FD-Modellierung zur Berechnung des Tensors des scheinbaren spezifischen Widerstandes. In K. Bahr und A. Junge (Hrsg.), *Protokoll 17. Kolloquium "Elektromagnetische Tiefenforschung"* (S. 245-251). Neustadt an der Weinstraße.

Demmel, J., Heath, M. und van der Vorst, H. (1993). Parallel Numerical Linear Algebra. In *Acta Numerica 1993* (S. 111–198). Cambridge, UK: Cambridge University Press.

Günther, T. (1998). *Modellstudien zur Tensorgeoelektrik.* (Unveröffentlichte Studienarbeit, Institut für Geophysik, TU Bergakademie Freiberg)

Heath, M. T., Ng, E. und Peyton, B. W. (1991). Parallel Algorithms for Sparse Linear Systems. *SIAM Review*, *33*(3), 420–460.

Huber, W. (Hrsg.). (1997). *Paralleles Rechnen: Eine Einführung von Walter Huber.* Rosenheimer Straße 145, D-81671 München: R. Oldenbourg Verlag.

Meuer, H., Strohmaier, E., Dongarra, J. und Simon, H. D. (2001). *TOP500 Supercomputer Sites.* (http://www.top500.org)

Mönch, W. (WS 00/01). *Informatik IV: Paralleles Rechnen.* (Vorlesungsmitschrift)

Pacheco, P. S. (Hrsg.). (1997). *Parallel Programming with MPI.* 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA: Morgan Kaufmann Publishers, Inc.

Snell, Q., Helmer, G. und andere. (1998). *NetPIPE - Network Protocol Independent Performance Evaluator.* http://www.scl.ameslab.gov/netpipe/.