Institut für Parallele und Verteilte Systeme

Abteilung Simulation grosser Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Studienarbeit Nr. 2045

Performanceanalyse und -optimierung der Gleichungslösung eines bestehenden Strukturmechanik-FEM-Programms für den Tunnelbau

Kiril Dichev

Studiengang:	Informatik
Prüfer:	Prof. Dr. Hans-Joachim Bungartz
Betreuer:	Dr.-Ing. Martin Bernreuther
begonnen am:	5.12.2005
beendet am:	6.06.2006
CR-Klassifikation:	C.4, G.1.6, G.1.8 , J.2

Vorwort

Der Anlass für diese Arbeit war die schlechte Performanz des Finite-Elemente-Programms tochnog. Dieses sollte näher untersucht und optimiert werden. Die Untersuchung verlief in zwei Phasen: In den ersten drei Monaten wurde ein Praktikum bei der Firma Züblin AG abgeschlossen. Danach wurden weitere Untersuchungen in der Fakultät für Informatik durchgeführt. Schließlich wurden die Ergebnisse schriftlich erfasst.

Inhaltsverzeichnis

1 Finite Elemente **1**

1.1 FE-Methode in der Strukturmechanik 1

1.1.1 Nichtlinearität und Iterationen 3

2 Einleitung zum FE-Programm tochnog **5**

2.1 FE-Programme 5

2.2 Einfaches tochnog-Beispiel 5

2.3 Zwei Projekte als Hintergrund der Testmatrizen 8

2.3.1 CS6282 8

2.3.2 BMBF-Projekt 10

2.4 Matrizen 10

2.4.1 CS6282 10

2.4.2 BMBF 12

2.4.3 Details 12

3 Hardware und Utilities **15**

3.1 Ziel-Architektur 15

3.1.1 SGI-Rechner 15

3.1.2 Amadeus-Rechner 15

3.2 Nützliche Programme 16

3.2.1 Monitoring 16

3.2.2 Profiling 16

3.2.3 Batch System 17

4 UMFPack-Optimierungen **19**

4.1 Solver- Beschreibung 19

4.2 BLAS 20

4.2.1 Aufwendigste BLAS 3 Operationen 21

4.3 Compiler und Compileroptimierungen 21

4.3.1 Intel Compiler 24

4.3.2 GNU Compiler 24

4.3.3 Ergebnisse mit Compilerflags 25

4.4 BLAS Parallelisierung 25

5 Alternative Solver im Vergleich **29**
5.1 Pardiso 29
5.2 WSMP 29
5.3 Ergebnisse und Vergleiche 30
5.3.1 Korrektheit der Ergebnisse 30
5.4 WSMP auf dem Amadeus 31

6 tochnog-Durchläufe mit UMFPack **33**
6.1 Atlas BLAS 33
6.2 Wie man ein proprietäres Programm profiliert 33

7 Zusammenfassung **37**

1 Finite Elemente

Der Finite-Elemente-Ansatz hat einen sehr umfangreichen theoretischen und praktischen Hintergrund. Dabei gibt es für jedes Anwendungsgebiet teilweise unterschiedlich hergeleitete Gleichungssysteme. Hier betrachten wir die FE-Methode im Rahmen der Strukturmechanik.

1.1 FE-Methode in der Strukturmechanik

Als ein Modellproblem der Strukturmechanik kann man folgendes Problem beschreiben: auf ein horizontal liegendes Rohr, das in mehreren Stützstellen gelagert ist, wird Druck ausgeübt. Die Frage ist: Wie verformt sich dieses , wenn es bestimmte physikalische Eigenschaften hat.

Von jeder Stützstelle gehen z.B. im zweidimensionalen Fall zwei zueinander senkrechte Vektoren aus. Diese beschreiben die Verformung die bei jeder Stützstelle stattfindet.Das sind die Freiheitsgrade: $\delta_i, i = 1..n$ Die äußeren Kräfte, die auf das Rohr einwirken, werden durch $f_i, i = 1..n$ beschrieben. Abb. (1.1) zeigt eine einfache Modellierung mit drei Stützstellen und sechs Freiheitsgraden.

Wir gehen nun davon aus, dass das System linear-elastisch ist - das bedeutet, dass einerseits jede Verformung bei Entfernen der Last komplett wieder zurückgeht, und dass die Verformungen proportional zur Last sind.

Dieses einfache Modell lässt sich dann so darstellen:

$$\begin{aligned} \delta_1 &= \delta_{11} + \delta_{12} + \delta_{13} + ... + \delta_{1n} \\ \delta_2 &= \delta_{21} + \delta_{22} + \delta_{23} + ... + \delta_{2n} \\ &... \\ \delta_n &= \delta_{n1} + \delta_{n2} + \delta_{n3} + ... + \delta_{nn} \end{aligned} \tag{1.1}$$

Es existiert dabei für jedes δ_{ij} eine Konstante c_{ij} , die entscheidet, was für ein Anteil die Last f_j auf die Verschiebung δ_i hat:

$$\delta_{ij} = c_{ij} f_j \tag{1.2}$$

Bei n Freiheitsgraden gibt es n^2 solche Koeffizienten. Nach Einsetzen von 1.2 in 1.1 bekommt man:

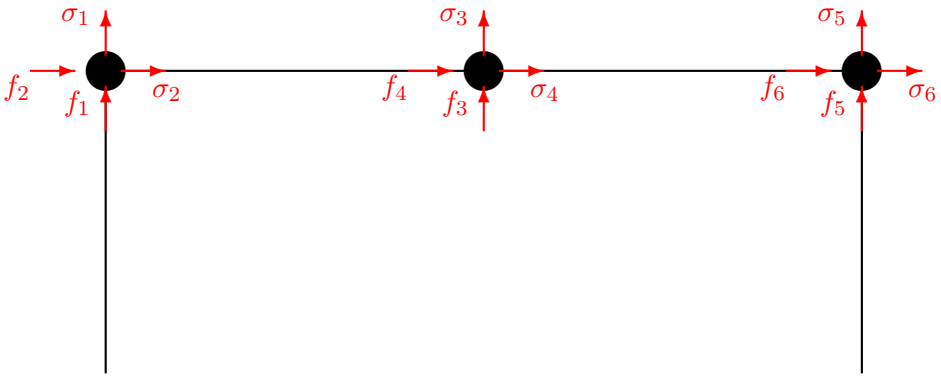

Abbildung 1.1: Ein einfaches Modell: ein Rohr wird an 2 Stellen gestützt. In jedem der drei Punkte werden die äusseren Kräfte f_i und die Verschiebungen σ_i angegeben

$$\begin{aligned} \delta_1 &= c_{11}f_1 + c_{12}f_2 + ... + c_{1n}f_n \\ \delta_2 &= c_{21}f_1 + c_{22}f_2 + ... + c_{2n}f_n \\ &... \\ \delta_n &= c_{n1}f_1 + c_{n2}f_2 + ... + c_{nn}f_n \end{aligned} \tag{1.3}$$

Anders ausgedrückt hat man:

$$d = Cf \tag{1.4}$$

mit

$$d = \begin{pmatrix} \delta_1 \\ \delta_2 \\ \dots \\ \delta_n \end{pmatrix} \qquad C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{n1} & \dots & \dots & c_{nn} \end{pmatrix} \qquad f = \begin{pmatrix} f_1 \\ f_2 \\ \dots \\ f_n \end{pmatrix} \tag{1.5}$$

In der Praxis wird aber nicht die Matrix C sondern $K = C^{-1}$ bei einem sogenannten Assemblierungs-Prozess berechnet. K heisst Steifigkeitsmatrix und ihre Koeffizienten werden dann gemäß der Struktur berechnet. Damit liegt folgende Gleichung vor:

$$Kd = f \tag{1.6}$$

d liefert dann einen Verschiebungsvektor. Jede Komponente liefert die Verschiebung bzgl. einem Freiheitsgrad.

Im Mittelpunkt dieser Studienarbeit steht das Lösen dieses Systems. Für diese Phase werden verschiedene Solver eingesetzt und untersucht.

1.1.1 Nichtlinearität und Iterationen

Man könnte prinzipiell in Abhängigkeit des betrachteten Systems folgende drei Typen von Nichtlinearität unterscheiden:

- linear-elastische Systeme

  Das ist die einfachste Art und 1.6 beschreibt so ein System vollständig.

- material-nichtlineare Systeme

  Hier ist das Spannungs-Dehnungs-Verhältnis nichtlinear. Zusätzlich wird der Initialzustand oft nicht wieder erreicht, nachdem man die Last entfernt.

- geometrisch-nichtlineare Systeme

  Bei diesen Systemen hängen die internen Kräfte vom Deformationszustand ab.

Der zuerst betrachtete Ansatz ist linear-elastisch. Für den Tunnelbau ist aber so ein Ansatz in der Regel nicht ausreichend. Es treten material-nichtlineare und geometrisch-nichtlineare Effekte auf. Z.B. ist die Verschiebung an Kontaktstellen nicht vernachlässigbar, der Boden ist nichtlinear,die Randbedingungen ändern sich während man Lasten aufbringt usw.

Es wird hier eine kurze Einleitung in die nichtlineare Analyse versucht.

Wir betrachten jetzt wie die Zeit eine entscheidende Rolle bei unseren Modellen spielt. Man kann sich z.B. vorstellen, dass bei unserem Rohr zwischen 2 Stützstellen 1 cm unter dem Rohr eine Feder steht. Es wird jetzt eine Last aufs Rohr gesetzt, und zwar linear mit der Zeit. Offensichtlich wird sich das Rohr in der ersten Phase (vor dem Kontakt mit der Feder) leichter verbiegen als in der zweiten Phase (die Feder stützt zusätzlich das Rohr). Damit haben wir ein nichtlineares Spannungs-Dehnungs-Verhältnis $(\sigma - \epsilon)$. Damit kann man bei vorgegebener Funktion $\sigma(t)$ für die Spannung auch die Dehnung $\epsilon(\sigma(t))$ als Funktion der Zeit modellieren. Also sind die Steifigkeit der Matrix und auch die einwirkenden Kräfte zeitabhängig. D.h, wenn das System im einfachsten Fall so aussieht

$$KU = R \tag{1.7}$$

haben wir jetzt:

$$K_t U_t = R_t \tag{1.8}$$

Die Gleichgewichtsbedingungen des Finite-Elemente-Systems sehen dann so aus:

$$R_t - F_t = 0 \tag{1.9}$$

Der Vektor R_t beschreibt die äußeren Kräfte, die zum Zeitpunkt t auf die Knoten einwirken. F_t entspricht $K_t U_t$ und steht für die inneren Kräfte, die in diesem Zeitpunkt auf die Knoten wirken. Das obige Gleichgewicht muss in allen Zeitpunkten gelten. Wir haben zwei Arten von Iterationen:

- 1. zeitliches Iterieren - F_t,K_t,R_t vorgegeben oder aus vorigen Zeitschritten zu berechnen

- 2. Newton-Raphson Iterationen. Diese sind innerhalb eines Zeitschrittes zu berechnen. Gesucht wird ein U_t so dass $K_t U_t - R_t$ sehr klein ist. Dabei werden F_t und K_t berechnet: F_t aus U_t; K_t im Prinzip so, dass $K_t = \frac{\partial^t F}{\partial^t U}$

$$K_{t+\Delta t}^{(i-1)} \Delta U^{(i)} = R_{t+\Delta t} - F_{t+\Delta t}^{(i-1)} \tag{1.10}$$

$$U_{t+\Delta t}^{(i)} = U_{t+\Delta t}^{(i-1)} + \Delta U^{(i)} \tag{1.11}$$

$$mit Anfangsbedingungen$$

$$U_{t+\Delta t}^{(0)} = U_t; K_{t+\Delta t}^{(0)} = K_t; F_{t+\Delta t}^{(0)} = F_t \tag{1.12}$$

Bei (1.10) wird der Solver aufgerufen.

2 Einleitung zum FE-Programm tochnog

2.1 FE-Programme

Zuerst verschaffen wir uns eine Übersicht und stellen die Frage: wie sind FE-Programme in der Arbeit eines Ingenieurs einzuordnen? Welche Rolle spielen dabei die Resultate eines solchen Programms?

Eine Skizze wird in Abb. 2.1 angegeben[1] : Grob kann man den Zyklus so beschreiben: in einer ersten Phase wird das physikalische Problem mathematisch modelliert. Das Modell wird dem FE-Programm übergeben und ein Ergebnis erzielt. Dieses Ergebnis wird dann interpretiert. Wenn das Ergebnis schlecht ist, wird der iterative Zyklus fortgesetzt: man kann entweder an der Genauigkeit des Modells zweifeln und dieses verbessern, oder man kann das physikalische Problem ändern (Das untersuchte Bauwerk ist nicht gut konzipiert).

2.2 Einfaches tochnog-Beispiel

Im Folgenden wird ein einfaches Beispiel der Arbeit mit tochnog beschrieben: Es handelt sich um die Berechnung der Temperaturverteilung anhand der Diffusions-Konvektions-Gleichung. Die Gleichung hat die allgemeine Form:

$$\rho C(\dot{T} + \beta_i \frac{\partial T}{\partial x_i}) = k(\frac{\partial^2 T}{\partial x_1^2} + \frac{\partial^2 T}{\partial x_2^2} + \frac{\partial^2 T}{\partial x_3^2}) - aT + f \tag{2.1}$$

Für die Parameter gilt: T ist die Unbekannte (Temperatur), und die restlichen Parameter werden initialisiert.

Die Eingabe für tochnog sieht so aus:

[1] Von [2] übernommen

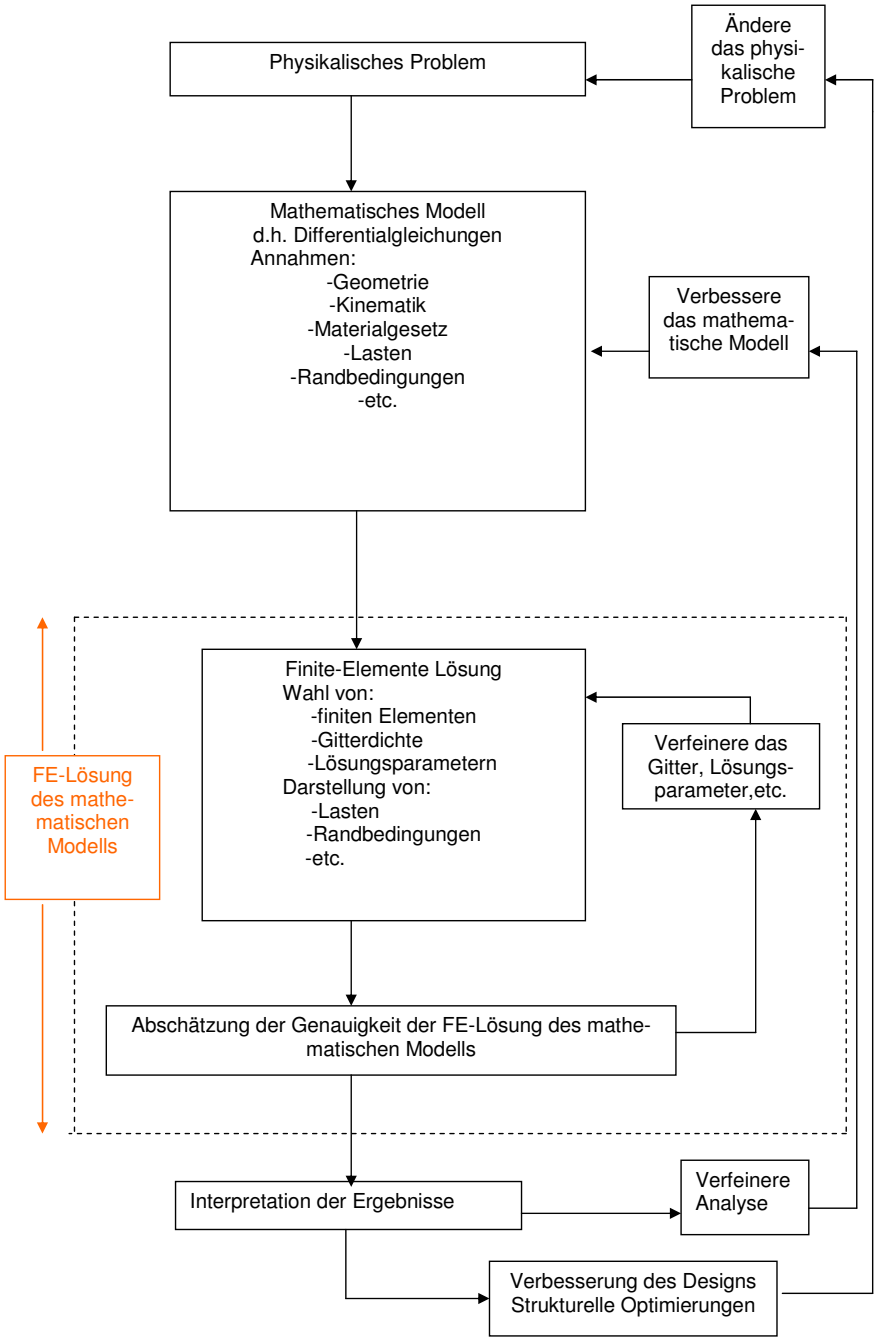

Abbildung 2.1: Rolle der FE-Programme

```
( Heat transfer: diffusion + convection.
  One-dimensional.
  Stationar.
  Left node prescribed temperature 1.
  Right node prescribed temperature 0.
  Middle node should get temperature 1.0 )

echo -yes
number_of_space_dimensions 1
condif_temperature
end_initia

node 1  0
node 2  1
node 3  2
element 1  -bar2 1 2
element 2  -bar2 2 3

geometry_point 1  0. 1.e-4
geometry_point 2  2. 1.e-4

bounda_dof 0  -geometry_point 1 -temp
bounda_time    0  0.0 1. 100. 1.
bounda_dof 1  -geometry_point 2 -temp
bounda_time    1  0.0 0.0 100.0 0.

group_type 0  -condif
group_condif_density 0  1.0
group_condif_capacity 0  1.0
group_condif_conductivity 0  0.1
group_condif_flow 0  10.

control_timestep   0  1. 2.0
control_print      0  -time_current -node_dof

target_item  0   -node_dof 2 -temp
target_value 0   1.0 2.e-2
end_data
```

Die Interpretation der Eingabedatei ist einfach: es wird ein eindimensionales Wärmeleitungsproblem definiert: auf der x-Achse seien die Punkte bei 0, 1 und 2 angegeben. Über **-bar2** werden lineare Elemente im 1D-Fall verwendet. **group_type** legt den Differentialgleichungstyp fest: in diesem Fall eine Konvektion-Diffusion-Gleichung vom oben angegebenen Typ. Die restlichen **group**-Angaben definieren alle in der Gleichung auftre-

tenden Parameter. Durch **bounda_time** werden Dirichlet-Randbedingungen angegeben: bei 0 und 2 soll die Temperatur zwischen Zeit 0.0 und 100.0 jeweils 1.0 und 0.0 sein. Anfangsbedingungen werden nicht gegeben. Es sei bemerkt, dass es sich hier um eine stationäre Gleichung handelt, d.h. die Zeit spielt keine Rolle. Der target-Teil steht für eine Kontrolle der Berechnungsergebnisse: kontrolliere, ob der mittlere Punkt sich höchstens um $2 * 10^{-2}$ vom Wert 1.0 unterscheidet. Wenn das nicht der Fall ist, erfolgt eine Fehlermeldung in der Log-Datei.

Hier werden zwei Zeitschritte berechnet: Zeitpunkt 1.0 und Zeitpunkt 2.0. Dies wird über die Angabe control_timestep festgelegt. tochnog führt aber für jeden Zeitschritt per Default zwei Iterationsschritte des Newton-Raphson-Iterationsverfahrens aus. Man kann auch explizit die Anzahl der Iterationsschritte beeinflussen, so dass unter Umständen viele Schritte innerhalb eines Zeitschritts möglich sind. Jeder einzelne Iterationsschritt ruft den Solver auf. Im obigen Beispiel wurde der Solver viermal aufgerufen. Es ist schon an diesem kleinen Beispiel klar, dass der Solver eine zentrale Rolle spielt.

tochnog ist unter anderem eng gebunden an das Programm GiD. GiD ist eine interaktive GUI für die Definition, Vorbereitung und Visualisierung von Daten, die mit numerischer Simulation zu tun haben. Unter anderem kann GiD aber auch Meshes für Finite-Elemente-Programme generieren und auch Ergebnisse anzeigen. Die zwei letzten Eigenschaften sind in Kombination mit tochnog nützlich. In einem Präprozessor-Modus kann man zuerst in GiD ein Gitternetz modellieren und verschiedene Eigenschaften definieren (z.B. wie dicht soll das Netz sein, was für Finite-Elemente sollen verwendet werden usw.). Das so generierte Gitter kann man als DAT-Datei exportieren. Diese kann dann in tochnogs Eingabedatei aufgerufen werden.

Weiterhin wird wie im obigen Beispiel in der Eingabedatei (Default-Name: tochnog.dat) die komplette Berechnung konfiguriert. Man kann Vorgänge beschreiben, die geplant wurden: z.B. das Ausgraben von Boden, das Eingießen von Beton usw. Während oder nach der Berechnung kann man Ausgabe-Dateien generieren, die z.B. für GiD oder gnuplot geeignet sind. In Abb. 2.2 z.B. sieht man die graphische Darstellung mit GiD vom Ergebnis der Berechnung, diesmal im Postprozessor-Modus. Bei den Punkten 0 und 2 sind die vorgegebenen fixen Werte jeweils 1.0 und 0.0. Im Punkt 1 hat sich der Wert 1.0 als stationäre Lösung eingestellt.

2.3 Zwei Projekte als Hintergrund der Testmatrizen

Wir schauen uns zwei Projekte der Firma Züblin an, die mit den später betrachteten Matrizen zu tun haben.

2.3.1 CS6282

Im Rahmen einer Ausschreibung für den Bau einer Metrolinie in Budapest war die Firma Züblin einer der Kandidaten. Das Projekt sah vor, dass der Bewerber zwei eingleisige U-Bahn-Röhren, einen Startschacht, eine Station und fünfzehn Querschläge zwischen den zwei Röhren baut (siehe Abb.2.3).

Abbildung 2.2: Visualisierung der Ergebnisse mit GiD: die Färbung zeigt, dass im Intervall [0,1] die Temperatur 1.0 Grad beträgt. Im Intervall [1,2] fällt sie ab auf 0.0 Grad.

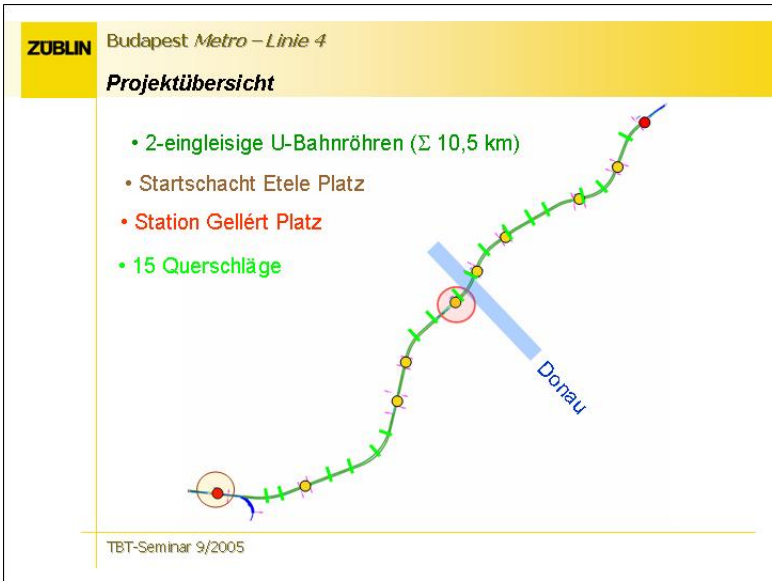

Abbildung 2.3: Übersicht des Budapest-Metro-Projekts

Der Verlauf der Röhren war so vorgesehen, dass sie unter anderem unter der Donau und unterhalb von hohen Gebäuden verlaufen sollen. Für diese kritischen Schnitte waren Simulationen unerlässlich. In einem Abschnitt sollen die U-Bahn-Röhren ein 15-stöckiges Hochhaus unterfahren (siehe Abb. 2.4). Dieser Abschnitt trägt den Namen CS6282.

2.3.2 BMBF-Projekt

In einem weiteren aktuell laufenden Projekt wird die Station Blijdorp als Teil eines U-Bahnprojekts in Rotterdam geplant (Abb. 2.5).

Es werden zwei Schlitze ausgegraben, die zuerst mit einer Stützflüssigkeit und dann mit Beton gefüllt werden. Nach dem Erstarren des Betons bildet sich eine sogenannte Schlitzwand, die dann stabil gegenüber dem Bodendruck von der Seite ist. Zwischen den gergestellten Schlitzwänden kann anschließend der Boden ausgehoben und die Station hergestellt werden.

Ziel der Anwendung von tochnog bei diesem Projekt ist ein Abgleich der Modellierung mit den tatsächlich auftretenden Verformungen der Schlitzwände in der Bauphase. Wichtige Fragen sind, wie sich der Boden verformt und wie sich Druck, Dichte und Temperatur im kompletten Bauverlauf ändern. Im Laufe des Projekts werden Messungen auf der Baustelle und Simulationen mit tochnog durchgeführt.

Bisherige Ergebnisse zeigen, dass die Simulationen vergleichsweise ungenaue Ergebnisse liefern, was vor allem an die ungenügenden Annahmen im Modell liegt.

2.4 Matrizen

Beim Ausführen von tochnog wurden die Steifigkeitsmatrix und der Lastvektor bei jedem Solveraufruf auf die Festplatte ausgegeben. Das Format zum Abspeichern von dünnbesetzten Matrizen ist Harwell-Boeing. Es ist weit verbreitet und konnte leicht in Matlab zur Veranschaulichung der Struktur importiert werden. Außerdem konnten die Matrizen auf verschiedenen Solvern ohne Modifikation angewendet werden. Im Folgenden werden die Matrizen näher untersucht.

Eine sehr wichtige Einschränkung bei den untersuchten Matrizen war, dass diese der Anfangsphase der tochnog-Ausführung entnommen wurden. Es ist in der Regel so, dass die Matrix sich in den nächsten Phasen verändert : innerhalb der Newton-Raphson Iterationen oder von der einen zur nächsten Bauphase (durch die virtuelle Zeit spezifiziert).

2.4.1 CS6282

Die Matrix (siehe Abb. 2.6) wurde generiert über eine sehr feine Gitterstruktur, die mit GiD (Preprocessor Mode) erstellt wurde. Obwohl die Matrix groß ist (n=238559) und viele Nichtnullen hat (7.6 Mio.), hat sich diese Matrix als relativ einfach für die Solver erwiesen. Die Anzahl der Nichtnullen bei einer LU- Zerlegung (mit dem Solver UMFPack) wächst auf etwa 45 Mio. und das einmalige Lösen des Systems auf einer SGI-Maschine mit UMFPack hat ohne BLAS (Abschnitt 4.2) 36 Sekunden gebraucht. Die Matrix ist bis auf 4 Einträge strukturell symmetrisch.

5.2.4 Cross section at CH 62+82

Fig 5.3 depicts the finite element mesh used for CH 62+82 (Pest side), for further information see cross section at CH 15+00.

Dead load of building:

g = (15 floors + 2 basement + roof) * 20 kPa = 360 kPa

Additional load on model due to building (dead load of building subtracted by weight of soil):

p = 360 – 2.3 * 17.5 – 2.0 * 20.0 – 1.0 * 19.5 - 0.4 * 11.5 = 255.65 kPa

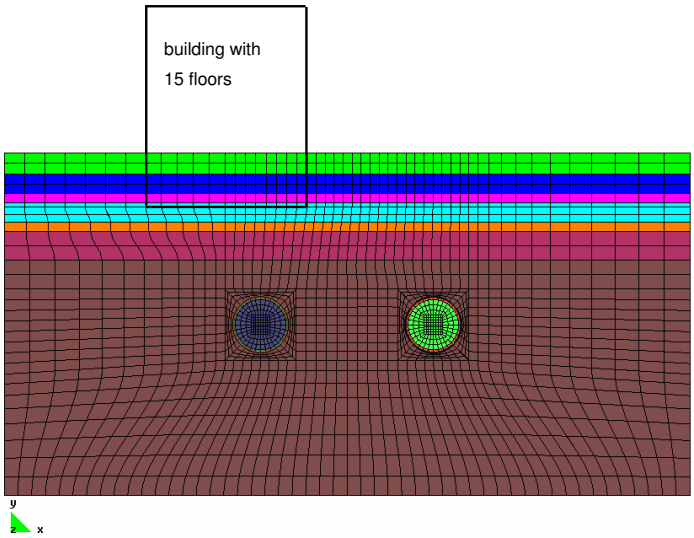

Fig 5.3 Finite element mesh at CH 62+82, different colours mark different element groups

Abbildung 2.4: Kritischer Tunnelabschnitt unter einem 15-stöckigen Hochhaus

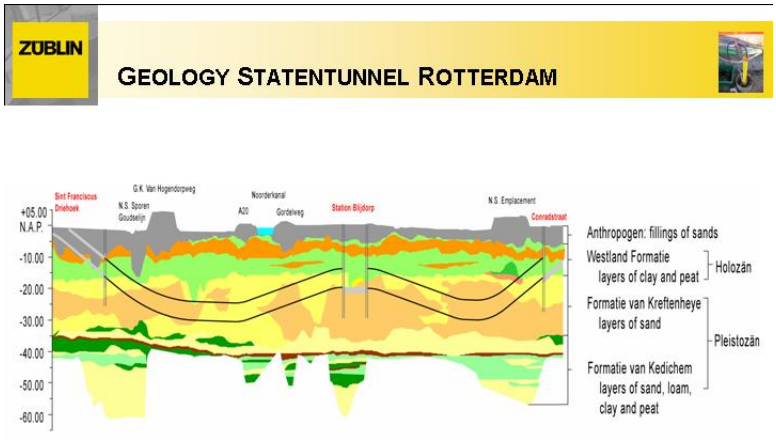

Abbildung 2.5: Station Blijdorp im Rotterdam-Tunnel

2.4.2 BMBF

Diese Matrix (siehe Abb. 2.7) ist etwas kleiner als die CS6282 (n=66420). Jedoch ist diese mit ihren etwa 5 Mio. Nichtnullen etwa 10 Mal dichter besetzt als die vorige Matrix. Bei der Zerlegung zeigt sich diese Matrix als extrem zeit- und speicheraufwendig: bei der LU-Zerlegung mit UMFPack explodieren die Nichtnullen auf 174 Mio. und ohne BLAS braucht dieser Solver für ein einziges Lösen des LGS ca. 1 Stunde. Die generierte Matrix ist strukturell symmetrisch.

2.4.3 Details

Tabelle 2.1 fasst die wichtigsten Eigenschaften der zwei Matrizen tabellarisch zusammen:

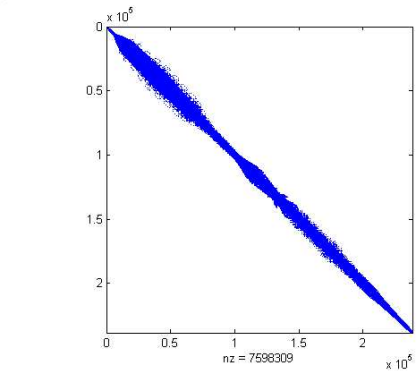

Abbildung 2.6: CS6282-Matrix: Nichtnullen-Struktur

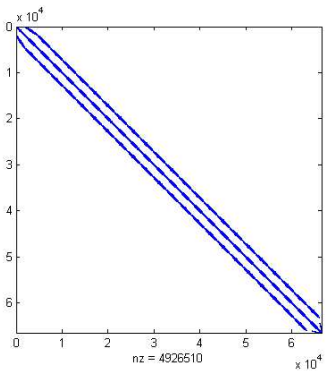

Abbildung 2.7: BMBF-Matrix: Nichtnullen-Struktur

Eigenschaft	*CS6282*	*BMBF*
n	238559	66420
Nichtnullen am Anfang	ca. 7.6 Mio.(0.01 %)	5 Mio.(0.1 %)
Nach LU-Zerlegung (mit UMFPack)	45.5 Mio.(0.08 %)	174 Mio.(4 %)
Rechenaufwand	gering	hoch
Umkehrwert der Kondition (abgeschätzt)	6.85e-03	1.71e-03
Anzahl der dichten Matrizen	29792	3514
Maximale Größe einer dichten Matrix	1044484	36990724 (!!!)
strukturell symmetrisch ($\forall i, j$ a_{ij} und a_{ji} beide ungleich Null oder beide gleich Null)	fast	ja
Grad an Asymmetrie ($\#a_{ij}$ mit $a_{ij} \neq a_{ji}$)	443584	738174

Tabelle 2.1: Details zu den Matrizen

3 Hardware und Utilities

3.1 Ziel-Architektur

3.1.1 SGI-Rechner

Die Zielarchitektur ist ein SGI Altix 350 System. Die Altix 350 ist eine Shared-Memory-Architektur. Diese kann bis zu 32 Prozessoren und 384 GB Speicher haben. Dabei ist besonders die “expand on demand ” Eigenschaft attraktiv. Diese erlaubt zu beliebigen Zeitpunkten das System zu upgraden, indem weitere Prozessoren und/oder Speicher gekauft wird. Charakteristisch für dieses System ist die NUMAflex-Architektur. Diese wird auch bei der Altix 3000 Serie eingesetzt. Die NUMAflex-Architektur bietet sowohl Vorteile, die für SMPs (Symmetric Multiprocessors) typisch sind, als auch solche, die die MPPs (Massively Parallel Processors) haben. Der Vorteil bei SMP ist die einfache Parallelisierung aus Programmiersicht - die OpenMP API ist ein Beispiel dafür. Der Vorteil bei MPP ist die hohe Skalierbarkeit - und die NUMAflex-Architektur skaliert gut. Bei NUMAflex werden Module gebildet - sogenannte Bricks. Sie enthalten Hauptspeicher, bis zu 2 CPUs, I/O. Verschiedene Bricks werden dann über NUMAlink Kabeln verbunden. Der Zugriff auf den Speicher eines anderen Bricks ist dann mit höherer Latenz und leicht niedrigerem Durchsatz verbunden.

Das von uns betrachtete System besitzt 2 sogenannte C-Bricks. Auf jedem Brick sind 2 Itanium2-Prozessoren untergebracht: auf dem ersten Brick 2 CPUs mit jeweils L1,L2,L3-Cache von 32 kB, 256 kB und 1.5 MB und Prozessorfrequenz von 1.5 MHz, auf dem 2. Brick 2 CPUs mit jeweils L1,L2,L3-Cache von 32 kB, 256 kB und 6 MB und Prozessorfrequenz von 1.6 MHz. Brick 1 hat einen Speicher von 11824 MB, Brick 2 hat 5902 MB, insgesamt also knapp 18 GB. Die 2 Bricks sind über 2 Links miteinander verbunden (formal eine 2-Module-Ring Topologie). Die Bandbreite in dem Fall ist 3.2 GB/s pro Prozessor. Das Betriebssystem ist SGI Advanced Linux Environment (basierend auf Red Hat) mit SGI ProPack. SGI Propack ist ein umfassendes Paket von Software für das SGI-Altix-System. Unter anderem enthält es die SCSL: die Scientific Computing Software Library von SGI mit Implementierungen von verschiedenen numerischen Algorithmen (unter anderem sind BLAS und LAPACK Bibliotheken die solche Algorithmen implementieren und in der SCSL vorhanden sind).

3.1.2 Amadeus-Rechner

Der Amadeus ist ein HP Integrity RX4640 Server mit 4 Itanium2-Prozessoren. Die Taktfrequenz ist 1.3 GHz, der L3-Cache hat jeweils 3 MB. Der Hauptspeicher beträgt 32 GB. Die verfügbaren Netzwerke sind GB-Ethernet und InfiniBand. Einige Tests werden auf diesem Rechner im Anschluss durchgeführt.

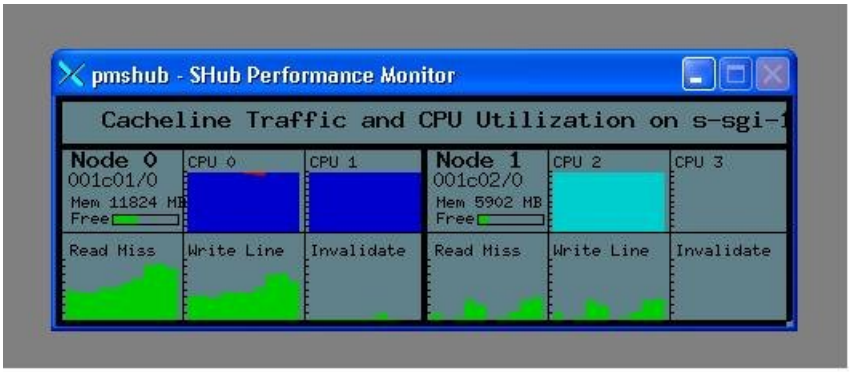

Abbildung 3.1: pmshub - Monitoring-Programm für SGI-Altix-Systeme

3.2 Nützliche Programme

3.2.1 Monitoring

Ein nützliches Monitoring-Tool ist pmshub (siehe 3.1). Es wird die Auslastung von jedem Prozessor angegeben sowie der Speicherverbrauch pro Brick. Es sind auch weitere Details sichtbar (z.B. Anteil Read Misses bei Speicherverbrauch, welche Auslastung der Prozessoren von Prozessen des eingeloggten Benutzers und welche von Prozessen anderer Benutzer verursacht wird).

3.2.2 Profiling

Profiling ist bei Optimierung von Software ein zentrales Thema. Der allgemeine Ansatz ist, ein Zeitprofil des zu optimierenden Programms zu erstellen und sich auf die Abschnitte zu konzentrieren, die die meiste Zeit brauchen.

Hier wurden zwei Profiling-Möglichkeiten betrachtet: der GNU Profiler *gprof* und das für den Itanium spezifische Perl-Skript *profile.pl*.

In diesem Fall hat sich *profile.pl* als besser geeignet erwiesen.

Die entscheidende Schwäche von *gprof* ist, dass für ein vollständiges Profil alle Quelldateien neu kompiliert und eingebunden werden müssen. Unter anderem war es nicht möglich, den Anteil z.B. von BLAS-Aufrufen zu bestimmen, da die BLAS-Bibliotheken meistens vorkompiliert sind. *gprof* stellt dann im flachen Profil nur die Zeitanteile dar, von denen der Quellcode vorliegt, und ignoriert die in vorkompilierten Bibliotheken verbrauchte Zeit. Dieses Profil ist oft zu ungenau. Andere Nachteile von *gprof* sind das relativ begrenzte Profiling (nur Zeit) und das häufige Modifizieren von Makefiles extra für profilierte Programme.

Das Skript *profile.pl* basiert auf dem Tool *pfmon*. Dieses Programm ist speziell für Itanium- Prozessoren geschrieben und kann fast 500 Hardware-Events monitoren - unter anderem CPU-Zyklen, L1-Cache, L2-Cache Leseversuche usw. Außerdem können die betrachteten Prozessoren spezifiziert werden. Im Gegensatz zu *gprof* benötigt *profile.pl*

nur die Symboltabelle des Compilers zum Profilieren. Damit konnten auch Bibliotheksaufrufe profiliert werden.

Da tochnog nur als kommerzielle Software vorliegt (stripped C/C++ Binaries ohne Symboltabellen) konnte diese nicht direkt profiliert werden. Hier betrachten wir das Profilieren von dem Solver. Im letzten Kapitel schauen wir uns ein indirektes Zeitprofil von tochnog an.

3.2.3 Batch System

Das verwendete Batch System ist *LSF* der Firma *Platform*. LSF ist eine Software zur Verwaltung und Ausführung von komplizierten Rechenaufgaben auf verteilten Systemen wie Clusters. Durch LSF kann man bei einem Job unter anderem Prioritäten setzen, Reports erstellen, und in dem Fall, dass ein einzelner Supercomputer im Cluster den Job abarbeiten soll, kann dieser genauer spezifiziert werden.

4 UMFPack-Optimierungen

4.1 Solver- Beschreibung

UMFPack ist ein direkter Solver für unsymmetrische dünnbesetzte Matrizen. Er löst das LGS

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

durch die sogenannte **U**nsymmetric-pattern **M**ulti**F**rontal Method. Es wurde UMFPack Version 4.4 verwendet.

UMFPack durchläuft drei wichtige Phasen zur Lösung eines LGS:

1. Spaltenvertauschungen zum Reduzieren von fill-in[1], Bestimmen der Nichtnullen-Struktur, Abschätzung des benötigten Speicherplatzes, Aufbau der Datenstruktur

   - `umfpack_*_symbolic`

2. Manchmal weitere Spaltenvertauschungen, Zeilenvertauschungen (Pivotieren) für numerische Stabilität und weitere fill-in Reduktion, LU- Zerlegung

   - `umfpack_*_numeric`

3. Lösen der Gleichung aufgrund der Zerlegung

   - `umfpack_*_solve`

Anstelle des *-Zeichens werden dann Bezeichnungen für die entsprechenden Typen von Systemen angegeben. Die Routinen für Freigabe von Ressourcen werden hier nicht betrachtet.

Im Folgenden wird der Quellcode für die allgemeine Lösung eines linearen Gleichungssystems mit UMFPack angegeben. Es wird ein reales Gleichungssystem mit Zahlen vom Typ double (doppelte float-Genauigkeit) mit Integer-Zahlen von Standardgrösse für die Indizierung gelöst.

[1] Auffüllen mit Nichtnullen

```
// die Dimension von A, die Matrix A und den Vektor b
//von der Festplatte in die Variablen n, Ap,Ai,Ax,b einlesen
readInMemo();
//Zeitmessung beginnt
umfpack_tic(stats);
// Wende den Solver an
double *null = (double *) NULL ;
void *Symbolic, *Numeric ;
(void)
umfpack_di_symbolic (n, n, Ap, Ai, Ax, &Symbolic, null, null) ;
(void)
umfpack_di_numeric (Ap, Ai, Ax, Symbolic, &Numeric, null, null) ;
umfpack_di_free_symbolic (&Symbolic) ;
x = (double *) malloc(n*sizeof(double));
(void)
umfpack_di_solve (UMFPACK_A, Ap, Ai, Ax, x, b, Numeric, null, null) ;
umfpack_di_free_numeric (&Numeric) ;
//Zeitmessung endet
umfpack_toc(stats);
//Eventuell x zur Kontrolle abspeichern...
```

4.2 BLAS

BLAS steht für **B**asic **L**inear **A**lgebra **S**ubroutines. Das sind Routinen für grundlegende Vektor- und Matrixoperationen. Die BLAS sind genau spezifiziert und lassen sich in 3 Teile gliedern: Level 1 BLAS sind skalare, Skalar-Vektor- oder Vektor-Vektor- Operationen, Level 2 BLAS sind Matrix-Vektor-Operationen, und Level 3 BLAS sind Matrix-Matrix Operationen.

Es gibt Referenz-Implementierungen der BLAS (z.B. in Fortran77), aber viele Hardware-Hersteller bieten eigene optimierte BLAS-Bibliotheken an.

Der Solver UMFPack verwendet per Default keine BLAS-Bibliotheken. Falls der Benutzer über BLAS Bibliotheken verfügt, können diese (durch Anpassen des Makefile) eingebunden und UMFPack neu kompiliert werden. Wir betrachten hier den Effekt, den die Benutzung von der SGI BLAS und Goto BLAS hat. Die SGI BLAS ist eine für SGI Serverplatformen optimierte BLAS-Bibliothek, die Goto BLAS liegt für verschiedene Architekturen (inklusiv Itanium2) vor. Beide Bibliotheken sind parallelisiert.

Zuerst untersuchen wir die Ausführung auf einem Prozessor.

Bei der BMBF Matrix ist der Effekt gigantisch: die Laufzeit ohne BLAS beträgt 3516 Sekunden. Ein Profil ergibt ein Anteil von 97.69 % in der Routine `umfdi_blas3_update`. Diese wird aufgerufen für die Level 3 BLAS Operationen falls kein BLAS benutzt wird. Das Kompilieren mit der SGI BLAS reduziert die Laufzeit auf 221 Sekunden. Dabei werden 71.88 % in `__scsl_dgemm_hoistc_` verbraucht- die Gauss-Elimination-Matrix-Matrix Multiplikation der SGI BLAS. Bei Goto BLAS sind es dann 156 Sekunden, an erster Stelle mit 55 % ist `dgemm`.

Weniger beeidruckend ist das Ergebnis bei CS6282 - die Laufzeit fällt ab von etwa 36

Sekunden auf etwa 14 Sekunden. Der `umfdi_blas3_update` Anteil von etwa 77 % beim Solver fällt ab auf etwa 13 % bei `__scsl_dgemm_hoistc_`. Bei Goto BLAS sind es etwa 22 Sekunden, an erster Stelle mit 11 % ist `dgemm`.

Damit ist SGI BLAS für CS6282 etwas besser (61 % schneller), Goto BLAS ist etwas besser bei BMBF (96 % schneller).

Abb. 4.1 und 4.2 zeigen graphisch die erreichte Beschleunigung durch BLAS jeweils bei CS6282 und BMBF.

4.2.1 Aufwendigste BLAS 3 Operationen

Die aufwendigsten Level 3 BLAS Operationen finden statt bei dem Update von U und bei der Berechnung des Schur-Komplements.

```
//Update von U
BLAS_TRSM_RIGHT (n, k, LU, nb, U, dc) ;
//Schur Komplement (C=C-L*U)
BLAS_GEMM (m, n, k, L, U, dc, C, d) ;
```

Diese Updates finden aber nur bezüglich speziellen Teilmatrizen statt, die aus bestimmten Zeilen und Spalten der Hauptmatrix zusammengesetzt werden. In der Terminologie von UMFPack sind das *frontal matrices* - kleine dichte Matrizen, die selber sogenannte Ketten bilden. Diese Ketten werden dann sequentiell durchlaufen und die Haupmatrix updatet. Ein kleines Beispiel für die Zerlegung in Frontalmatrizen wird in Abb.(4.3) angegeben[2].

Offensichtlich ist der entscheidende Faktor bei der Laufzeit die Grösse der Frontalmatrizen. Es ist besser viele kleine Matrizen zu multiplizieren als eine grosse Matrix. Durch ein report von UMFPack lässt sich die Anzahl der Frontalmatrizen und die maximale Grösse einer solchen Matrix ausgeben. Die Angaben wurden in Tabelle 2.1 gemacht. Es ist zu erwarten, dass die BMBF-Matrix als dichter besetzt eine grössere maximale Frontalmatrix hat. Das fällt sehr eindeutig aus: die BMBF-Matrix besitzt 3514 dichte Matrizen, und die Matrix mit maximaler Grösse hat dabei knapp 37 Mio. Einträge. Bei der CS6282-Matrix gibt es dabei 29792 dichte Matrizen, die grösste von denen hat aber ca. 1 Mio. Felder.

4.3 Compiler und Compileroptimierungen

Der nächste Schritt nach dem Binden von sequentiellem BLAS ist die Untersuchung, wie sich unterschiedliche Compiler und Compilerflags auf die Laufzeit auswirken. Dabei binden wir für beide Matrizen die SGI BLAS in den Solver und versuchen, weitere Verbesserungen zu erzielen.

[2] übernommen von [7]

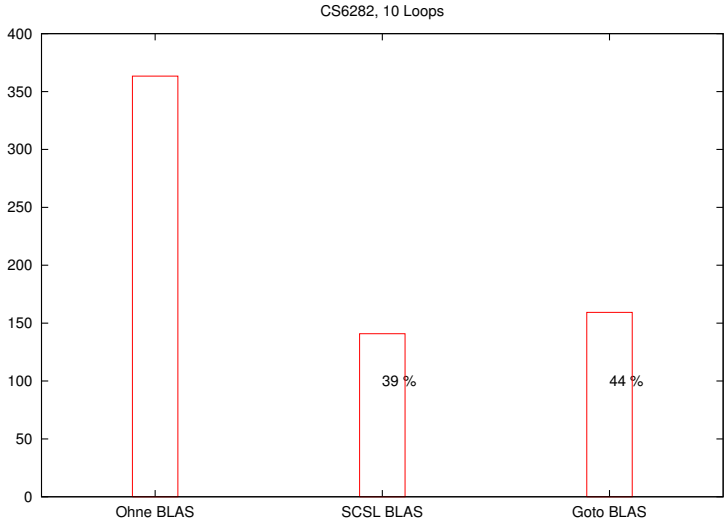

Abbildung 4.1: BLAS-Effekt bei CS6282-Matrix

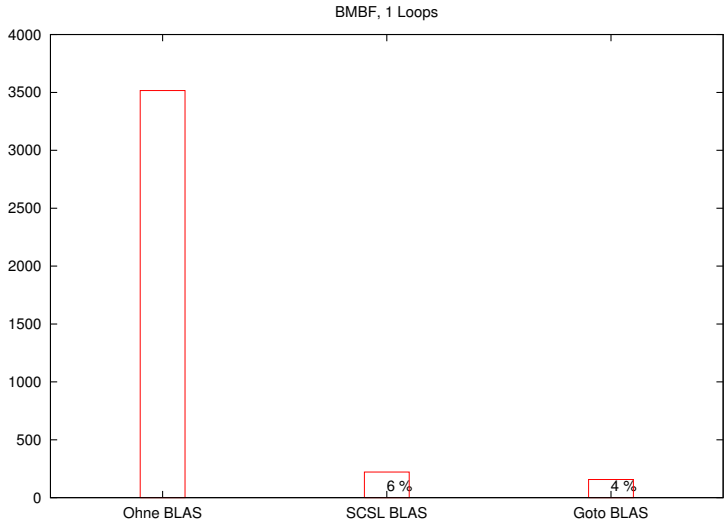

Abbildung 4.2: BLAS-Effekt bei BMBF-Matrix

Figure 2: Example numerical factorization: matrices and tree

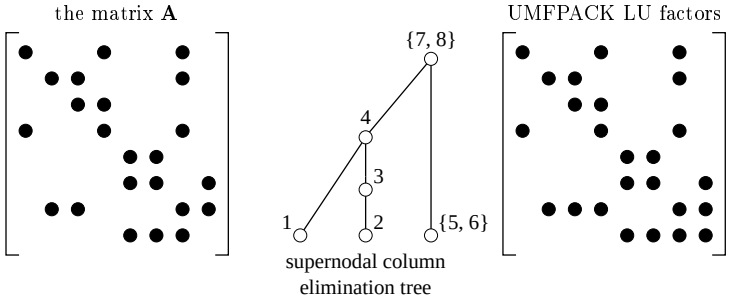

Figure 3: Example numerical factorization: details of frontal matrices

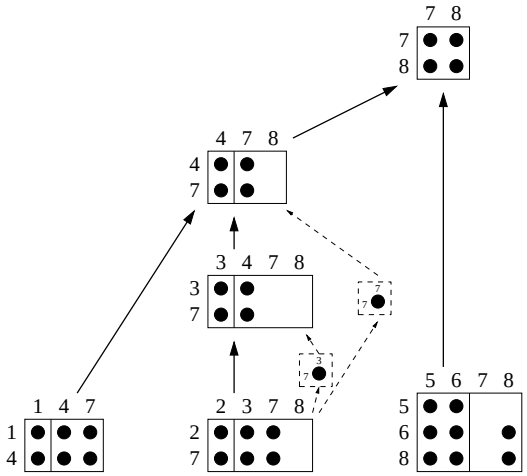

Abbildung 4.3: Beispiel einer Zerlegung in Frontalmatrizen

4.3.1 Intel Compiler

Es wurde Intel Compiler Version 8.1 benutzt.
Vier generelle Optimierungsrichtungen wurden getestet:

- allgemeine Optimierungen über -Ox (x=0..3) Angaben. O0 schaltet jede Optimierung aus, -O1 zielt auf Codegrössenoptimierung und Code-Lokalität und ermöglicht einige Speed-Optimierungen. O2 ist die empfohlene Option für Speed-Optimierungen und die Default-Optimierungsstufe. -O3 schaltet agressivere Optimierungsmethoden ein, die aber nicht immer eine Verbesserung der Laufzeit bringen.

- intra-/interprozedurale Optimierungen -ip und -ipo. Diese Optimierungen haben im Mittelpunkt Code-Inlining. Bei -ipo (interprozedurale Optimierungen) müssen alle Quelldateien vorliegen, die aufgerufen werden.

- profilgesteuerte Optimierungen -prof_gen und -prof_use. Diese Drei-Stufen-Optimierung erfolgt so: zuerst wird das Programm mit -prof_gen kompiliert, dann werden häufig auftretende Durchläufe ausgeführt und automatisch Profile generiert. Im 3. Schritt wird dann das Programm nochmal anhand dieser Profile mit -prof_use kompiliert und so speziell für diese Runs optimiert.

- Schleifen- und OpenMP-Parallelisierung mit -parallel und -openmp Optionen. -parallel aktiviert Multithreading für Schleifen, die parallel ausgeführt werden können, -openmp parallelisiert bei Anwesenheit von OpenMP Direktiven.

Für weitere Details siehe [4].

4.3.2 GNU Compiler

Es wurde GCC Version 3.2.3 benutzt. Der GNU Compiler hat eine sehr grosse Auswahl an Compiler-Direktiven. Es wurden nur einige wenige Kombinationen probiert:

- -O3: Die maximale Speed-Optimierungsstufe. Aktiviert viele Flags, die nicht weiterhin beschrieben werden.

- -funroll-loops: Loop Unrolling, insb. nützlich da das ineffiziente Loop Branching bei weniger und grösseren Loops seltener stattfinden kann.

- -fprefetch-loop-arrays: Prefetch Memory für Loops, die grosse Datenmengen laden.

- -fomit-frame-pointer: z.B. bei Aufrufen von einzeiligen Subroutinen nützlich (Auslassen von Rahmen).

Die ausführlichste Quelle der Optimierungsoptionen sind sicherlich die Manual Pages von gcc. Für eine etwas strukturiertere Vorgehensweise wird auf Fachzeitschriften oder Foren verwiesen.

4.3.3 Ergebnisse mit Compilerflags

Die etwas überraschende Erkenntnis war, dass insgesamt das Kompilieren mit dem Intel Compiler ohne Flags am schnellsten war. Keine Flags brachten bedeutende Verbesserungen (Abb. 4.4) . Im Prinzip kann man sich das schon erklären - immerhin sind die aufwendigsten Operation vorkompiliert. Wichtige Beobachtung war, dass ecc -O3 entweder Segmentation Fault oder falsche Ergebnisse lieferte.

Insgesamt bekam der Autor den Eindruck, dass Intel deutlich weniger Optimierungsoptionen als der GNU Compiler hat. Dafür ist die Vorgehensweise zum Optimieren besser organisiert. Dafür hat man bei gcc eine enorme Menge an Flags, die aber kaum einzeln getestet werden können.

4.4 BLAS Parallelisierung

Im Prinzip bietet SGI sowohl eine sequentielle als auch eine parallele BLAS Version an. Die Parallelisierung basiert auf OpenMP. Im Gegensatz ist Goto BLAS parallelisiert und versucht per Default, alle Prozessoren auszunutzen. Es wurden bisher die sequentielle SGI BLAS und Goto BLAS mit "unterdrückter" Parallelisierung benutzt. Jetzt untersuchen wir die parallele SGI BLAS und Goto BLAS mit entsprechend angegebenen Umgebungsvariablen.

Es sei erwähnt, dass wir auf der primitivsten Ebene parallelisieren - bei den arithmetischen Operationen. Unser UMFPack- Algorithmus ist immer noch sequentiell.

Die Ergebnisse lassen sich so zusammenfassen: parallele BLAS Versionen lohnen sich da, wo auf grosse Frontalmatrizen BLAS Operationen ausgeführt werden. Bei kleinen Frontalmatrizen ist der Trade-Off für Multithreading so hoch, dass die Parallelisierung eindeutig zu einer Verlangsamung führt.

Im Hinblick auf unsere Testmatrizen: für die CS6282 lohnt sich gar keine BLAS Parallelisierung, bei der BMBF bringt diese einen deutlichen Zeitgewinn. Dass Multithreading im Fall CS6282 einfach zu aufwendig ist, zeigen auch folgende 2 Profilausschnitte. Es wurde hier die parallele SGI BLAS benutzt. Die aufgelisteten 5 Routinen stehen ganz oben auf der CPU-Takten-Liste. Zuerst haben wir 1 Thread. Wie erwartet ist hier *dgemm* die aufwendigste Routine.

```
      Ticks     Percent  Cumulative   Routine
                          Percent
-----------------------------------------------------------------
       4357       19.74    19.74      __scsl_dgemm_hoistc_
       2113        9.57    29.32      _IO_vfscanf_internal
       1448        6.56    35.88      umfdi_assemble_fixq
       1430        6.48    42.36      umf_i_analyze
       1287        5.83    48.19      shift_pivot_row
```

Dann betrachten wir die aufwendigsten 5 Routinen bei 4 Threads auf 4 Prozessoren. Die Routinen *dgemm* belegt nur noch Platz 5, alle Routinen davor hängen mit OpenMP und Multithreading zusammen.

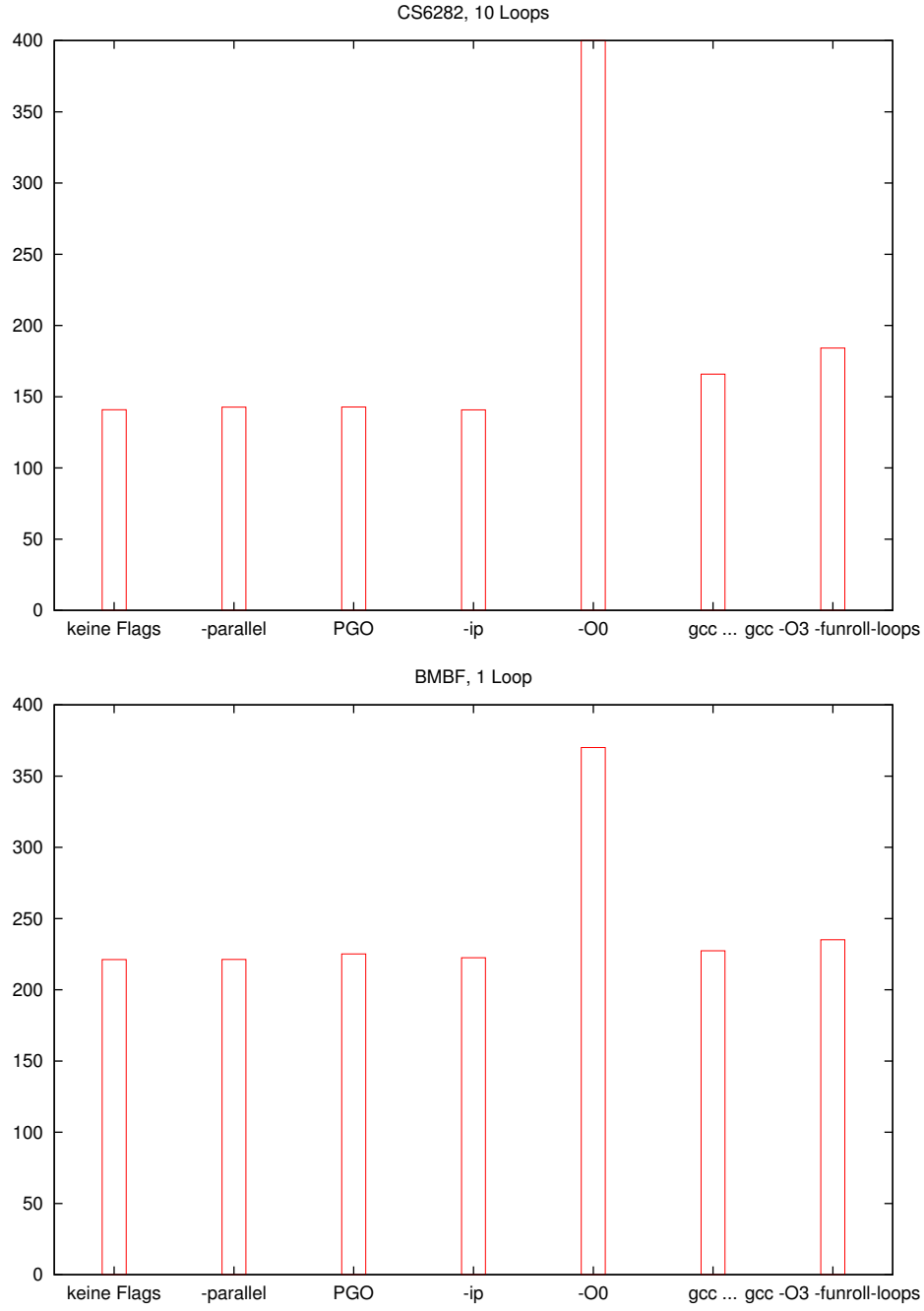

Abbildung 4.4: Compiler-Optimierungen bei CS6282 und BMBF

```
27238       30.37    30.37      __kmp_wait_sleep
12116       13.51    43.88      __kmp_yield
 9725       10.84    54.72      LIBRARY:/opt/intel_cc_80/lib/libguide.so
 8816        9.83    64.55      __kmp_ia64_pause
 5940        6.62    71.18      __scsl_dgemm_hoistc_
```

Jetzt zur BMBF Matrix. Mit 1 Thread haben wir hier.

```
18770       68.37    68.37      __scsl_dgemm_hoistc_
 1700        6.19    74.56      shift_pivot_row
 1281        4.67    79.22      _IO_vfscanf_internal
  545        1.99    81.21      __strtod_internal
  524        1.91    83.12      __scsl_dgemm_hoistb_
```

Bei 4 Threads auf 4 Prozessoren haben wir in der Tat einen hohen Multithreading-Aufwand, jedoch nicht höher wie der Aufwand für die Matrix-Matrix-Multiplikation:

```
22486       37.70    37.70      __scsl_dgemm_hoistc_
12074       20.24    57.94      __kmp_wait_sleep
 5236        8.78    66.71      __kmp_yield
 4164        6.98    73.69      LIBRARY:/opt/intel_cc_80/lib/libguide.so
 3795        6.36    80.06      __kmp_ia64_pause
```

Dabei zeigt Goto BLAS bei BMBF das bessere Verhalten (4 CPUs brauchen 51 % der Laufzeit auf 1 CPU), dafür ist es deutlich schlechter bei CS6282 (4 CPUs brauchen 277 % der Laufzeit auf 1 CPU). Abb. 4.5 zeigt die vollständigen Ergebnisse der BLAS-Parallelisierung mit den 2 Matrizen auf 1,2 und 4 CPUs.

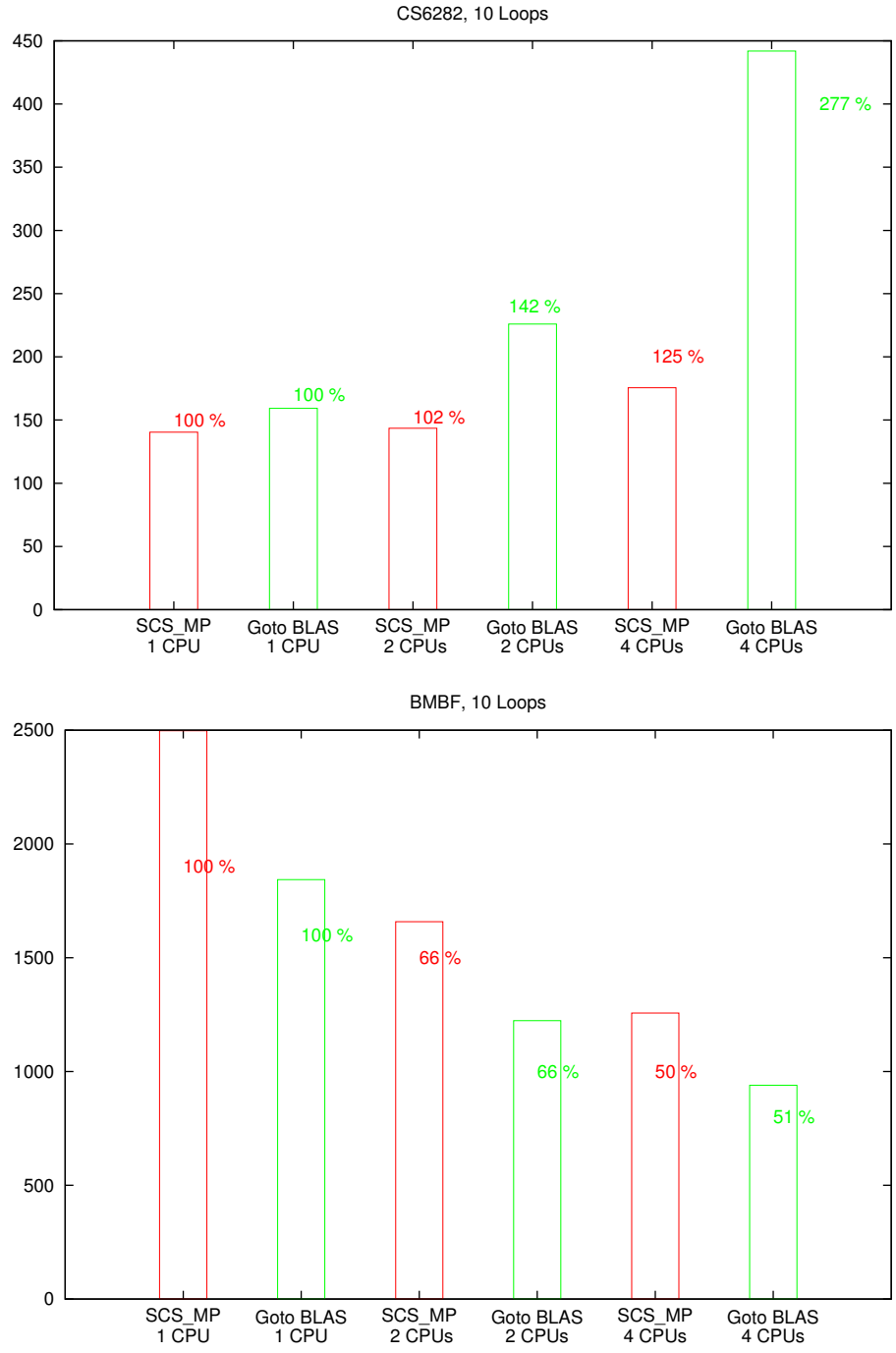

Abbildung 4.5: BLAS Parallelisierung - CS6282 und BMBF

5 Alternative Solver im Vergleich

Eine entscheidende Schwäche von UMFPack ist die Tatsache, dass der Open-Source-Algorithmus sequentiell ist. Es existiert eine parallele Version von UMFPack von Steve Hadfield, die nicht frei verfügbar ist.

Im Gegensatz dazu gibt es eine Menge von alternativen direkten Solvern, die parallelisiert sind. Im Folgenden betrachten wir zwei state-of-the-art direkte Löser: Pardiso und WSMP. Beide haben den Anspruch, Thread-Safety, Speichereffizienz und Robustheit zu besitzen. Die Bibliotheken beider Solver bestehen sowohl aus Fortran- als auch aus C-kompiliertem Code. Die Routinen sind aufrufbar über C und Fortran. Die Solver sind für akademische Zwecke kostenlos, ansonsten kann jeweils eine kommerzielle Lizenz gekauft werden.

5.1 Pardiso

Pardiso wurde an der Universität Basel entwickelt. Er ist für symmetrische und unsymmetrische dünnbesetzte Gleichungssysteme geeignet und speziell für shared-memory Multiprozessoren parallelisiert. Details zum Algorithmus finden sich hier ([8]). Eine frühe Pardiso-Version wurde in das Math Kernel Library von Intel integriert. Seitdem wurde der Solver deutlich geändert und wird weiterentwickelt. Die hier getestete Version ist 3.0.

Pardiso braucht die Level-3-BLAS Operationen. Dazu ist es beim Kompilieren notwendig, dass zusätzlich zur Pardiso-Bibliothek obligatorisch eine BLAS-Bibliothek verlinkt wird. Weitere Hinweise zum Kompilieren für jede Architektur, für die Pardiso vorliegt, finden sich im Manual.

5.2 WSMP

WSMP wurde von der IBM alphaWorks-Forschungsruppe, vor allem von Anshul Gupta ([12]) entwickelt. Der Solver ist für symmetrische und allgemeine Systeme geeignet. Zusätzlich zu einer SMP-Parallelisierung mit Multithreading bietet IBM auch eine distributed-memory Variante mit MPI, die selber auf jedem Multiprozessor Multithreading ausnutzen kann.

WSMP benutzt die ATLAS BLAS per Default, jedoch kann eine beliebige alternative BLAS Bibliothek verlinkt werden.

Die verwendete WSMP-Version ist 6.0.4.

5.3 Ergebnisse und Vergleiche

Notiz: Es stellte sich heraus, dass Goto BLAS zusammen mit beiden Solvern zum Absturz führte, falls die Solver mit mehr als einem Thread gestartet wurden. Unter Umständen wurde die Berechnung auch mit falschen Ergebnissen abgeschlossen, sogar im Fall, dass der Solver und Goto jeweils 1 Thread hatten (bei beiden Solvern). Insgesamt empfiehlt sich Goto BLAS nicht bei den parallelen Solvern.

Bei dem WSMP- Solver wurde die Kompilierung mit der sequentiellen SGI BLAS gewählt. Ein wichtiger Hinweis im Manual ist, dass eine sequentielle BLAS Version vom Solver erwartet wird, da WSMP selber eine Parallelisierung durchführt. Ansonsten ist mit viel längeren Rechenzeiten zu rechnen.

Die Anzahl von Threads wird bei WSMP im Quellcode gesetzt (über die Bibliotheks-Routine **wsetmaxthrds**). Die Parallelisierung hat bei beiden Matrizen zu Zeitreduzierung geführt. Abb. 5.1 zeigt die Ergebnisse für die zwei Matrizen. Mit 4 CPUs hat man für CS6282 etwa 15 Sekunden für die Bestimmung von x, für BMBF etwa 29 Sekunden.

WSMP bietet viele Parameter an, von denen aber wenige vom Benutzer angegeben werden müssen. Die Parameter sind per Default sehr Fortran-orientiert, und bei der C-Schnittstelle müssen extra Indizierung und Matrix-Format berücksichtigt werden.

Der Pardiso-Solver wurde auch mit der SGI BLAS kompiliert. Ähnlich wie bei WSMP, wird darauf hingewiesen,dass BLAS und zusätzlich LAPACK bei Pardiso seriell und thread-safe sein sollen. Die Kontrolle der Prozesse erfolgt über die Umgebungsvariable `OMP_NUM_THREADS`, die dann bei Programmstart eins der Solver-Parameter setzt. Ein Vorteil des Solvers bei der C-Programmierung ist die gute Dokumentation der C-Schnittstelle.

In Abb. 5.1 sind die Ergebnisse für die 2 Matrizen dargestellt: Pardiso schafft es mit 4 CPUs in 17.7(18.7) Sekunden, die BMBF-Gleichung zu lösen. In diesem Fall hat man noch die Auswahl, ob man BMBF als eine unsymmetrische Matrix betrachtet oder ob man die strukturelle Symmetrie von BMBF ausnutzen möchte. Unter Ausnutzung der strukturellen Symmetrie hat man hier die beste Zeit. Für die CS6282 muss man zwangsläufig eine unsymmetrische Matrixstruktur angeben, die Rechnung braucht dann etwa 10.5 Sekunden.

Damit ist Pardiso eindeutig der schnellste Löser unter den bisher getesteten Lösern.

Damit ist auch diese zweite Phase besonders für die BMBF-Matrix mit deutlichen Zeitgewinnen abgeschlossen. Man beachte, dass die in Abb. 5.1 skizzierten Ergebnisse mit der Ausgangssituation (links) anfangen, dann die beste Zeit nach der Einbindung von sequentiellem/parallelem BLAS mit UMFPack, und dann die Wirkung der alternativen Solver untersuchen. Bei BMBF hat sich die Laufzeit grob noch sechsfach verringert. Damit beträgt sie nur noch etwa 0.5 % der Anfangslaufzeit.

5.3.1 Korrektheit der Ergebnisse

Da jeder Solver viele Optionen anbietet, war die Vorgehensweise ziemlich praktisch: mit möglichst wenigen nicht-Default-Optionen ein genaues Ergebnis für den Vektor x zu erzielen. Als eine Möglichkeit wurden die Ergebnisvektoren zwischen verschiedenen

# CPUs	SGI (seq.SGI BLAS)	Amadeus (F77 BLAS)
1	24.797852	29.947266
2	17.442383	17.363281
4	15.035156	15.942383
1	53.218750	65.882812
2	41.672852	39.293945
4	29.226562	26.687500

Tabelle 5.1: Bemessene Zeit auf der SGI-Maschine und auf Amadeus

Lösern über die Norm $\max_i |x_i - y_i|$ verglichen. Diese Abweichung war höchstens 10^{-13}. Die Ergebnisvektoren sind übrigens bei beiden Matrizen nah am Nullvektor: $\max_i |x_i|$ ist bei CS6282 0.160105, bei BMBF 0.530821. Zum Glück waren diese weit genug vom Nullvektor entfernt, um ein genaues Ergebnis vom Nullvektor zu unterscheiden.

5.4 WSMP auf dem Amadeus

Auf Amadeus wurde WSMP (die neuere Version 6.2.28) unter Verwendung von Fortran Bibliotheken (u.a. Fortran77 BLAS) kompiliert und getestet. Damit konnte man einen direkten Vergleich zur bisher verwendeten SGI Architektur machen.

Der Ein-Prozessor-Lauf ist generell etwas schlechter bei Amadeus, was aber wegen den verschiedenen eingesetzten BLAS Bibliotheken keine grosse Aussagekraft hat. Parallelisierung hat aber bei Amadeus eine höhere Auswirkung gezeigt als bei der verwendeten SGI-Maschine. Die mit vier CPUs parallelisierte Variante ist bei Amadeus insgesamt leicht besser- etwa gleich schnell bei CS6282 und überlegen bei BMBF.

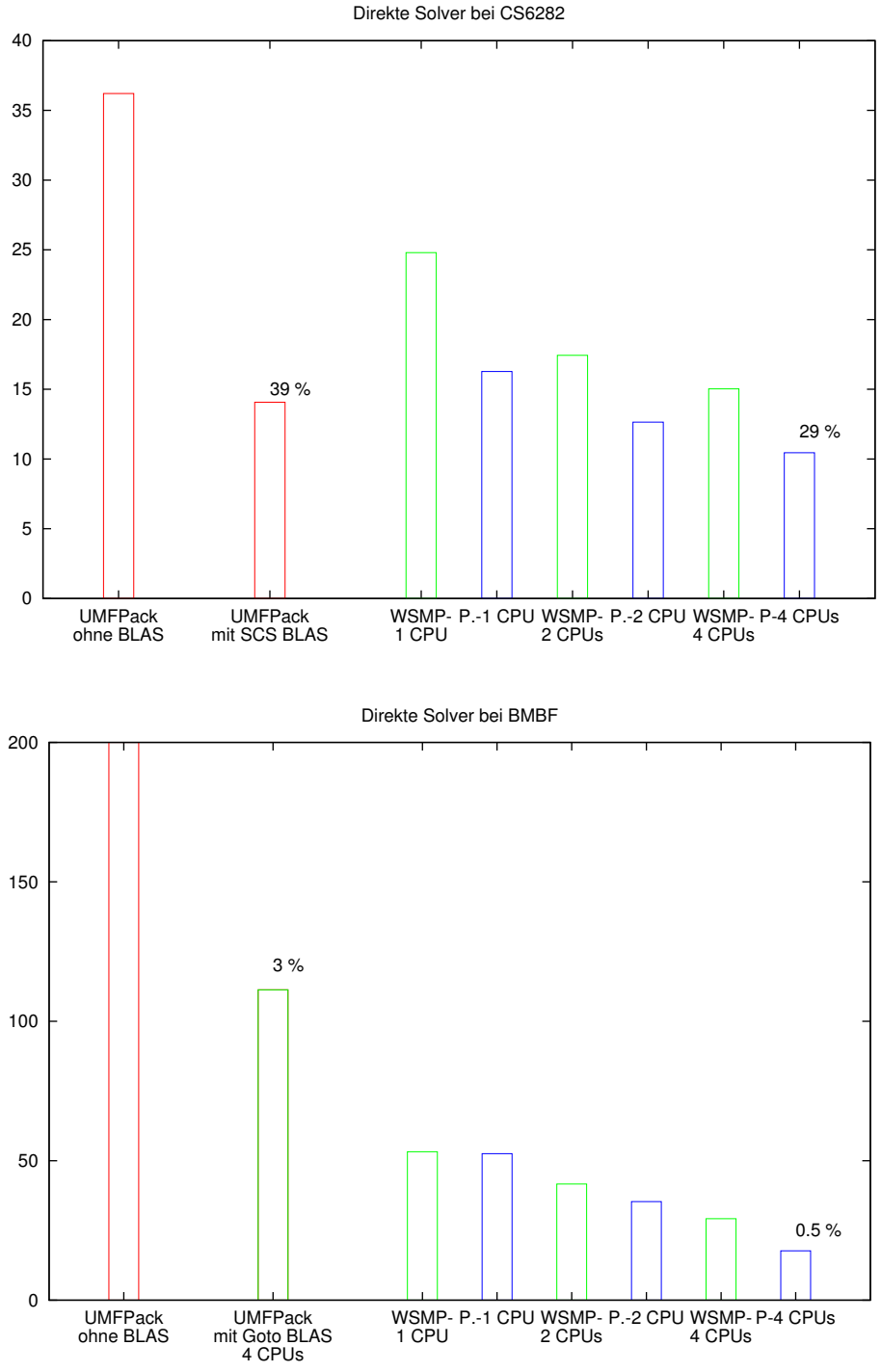

Abbildung 5.1: Vergleich der Solver Pardiso und WSMP mit den bisherigen Ergebnissen: CS6282(oben) und BMBF(unten)

6 tochnog-Durchläufe mit UMFPack

6.1 Atlas BLAS

ATLAS BLAS wurde als eine interessante Alternative der bisherigen BLAS Versionen in Betracht gezogen. Der Vorteil von ATLAS ist, dass es als Source-Code zur Verfügung steht und auf der Zielmaschine kompiliert werden kann (also völlig portabel). Insbesondere kann man dann nach Wunsch eine statische Bibliothek erstellen, und jede benötigte Routine in die ausführabare Datei einbinden. ATLAS steht für **A**utomatically **T**uned **L**inear **A**lgebra **S**oftware. Bei der Kompilierung versucht ATLAS BLAS selbst iterativ festzustellen, welche Compile-Optionen die Effizienz am meisten steigern. Unter anderem werden teilweise bei manchen Architekturen bekannte Vorteile ausgenutzt. Als Ergebnis ist ATLAS BLAS konkurrenzfähig zu den maschinenoptimierten BLAS- Versionen.

Tests mit sequentiellem ATLAS BLAS haben beim einmaligen Lösen mit UMFPack bei BMBF mit 239 Sekunden etwa 35 % schlechteres Verhalten als Goto BLAS und bei CS6282 mit 20.9 Sekunden etwa 33% schlechteres Verhalten als SGI BLAS gezeigt. Umso überraschender fallen die nachfolgenden Ergebnisse mit tochnog aus.

6.2 Wie man ein proprietäres Programm profiliert

Im Anschluss der Arbeit bei Züblin wurden Tests mit tochnog durchgeführt. Obwohl man nicht über den Quellcode verfügt, ermöglicht tochnog über die Option -print_where bestimmte Outputs, die Auskunft darüber geben, in welcher Phase tochnog gerade eintritt. Ein Perl-Skript wurde mit dem Ziel entwickelt, die Outputs für Messung der Dauer der einzelnen Phasen auszunutzen. Damit hat man sich einen Überblick verschafft, wie lange der Solver als Bestandteil des Programms benötigt und ob der optimierte Solver beim Einsetzen im Program tatsächlich die gewünschte Effizienzsteigerung zeigt.

Die letzte Frage ist bis jetzt völlig offen: schließlich basieren alle Tests mit den Solvern nur auf zwei Matrizen, die erstens nur bestimmte Probleme wiederspiegeln, und zweitens nur die Anfangsphase der Berechnung.

Es wurden drei tochnog-Versionen getestet: die originelle tochnog3.1 Version mit UMFPack ohne BLAS, eine Version die ATLAS BLAS verwendet und eine die Goto BLAS verwendet. Für den Entwickler war besonders ATLAS BLAS von Interesse, da nur diese als statische Bibliothek kompiliert werden konnte und anschließend beim Linking in die ausführbare Datei eingebunden werden konnte.

Goto BLAS wurde hier nicht eingeschränkt, d.h. es wurden alle 4 Prozessoren ausgenutzt.

Berechnung	Solverzeit (Sek.)/ Anteil	Gesamtzeit (Sek.)	Zeitgewinn (gesamt)
kbt - tochnog3.1	47103 (35 %)	136507	0 %
kbt - tochnog3.1 mit ATLAS	6599 (6.7 %)	99090	27.5 %
kbt - tochnog3.1 mit Goto	7854 (8 %)	97205	28.6 %
rotterdam - tochnog3.1	20125 (44 %)	46060	0 %
rotterdam - tochnog3.1 mit ATLAS	4538 (21 %)	21703	53 %
rotterdam - tochnog3.1 mit Goto	4964 (32 %)	15385	66.7 %
stahl - tochnog3.1	14952 (51 %)	29289	0 %
stahl - tochnog3.1 mit ATLAS	4336 (23 %)	18920	35 %
stahl - tochnog3.1 mit Goto	8803 (37 %)	23642	19 %

Tabelle 6.1: Solver- und Gesamtzeit von drei Langzeitberechnungen mit verschiedenen tochnog-Versionen

Drei aktuelle Probleme aus dem Tiefbau wurden eingesetzt: der Bau eines Tunnelabschnitts (RandStadRail-Statenwegtrace in Rotterdam), Einbau eines Stahl-Tübbing-Ringes (auch Rotterdam) und die Berechnung eines Querschlages (Katzenbergtunnel).

Die Ergebnisse zu den 3 Langzeitrechnungen haben eindeutig die Verbesserung des Solveranteils gezeigt (Tabelle 6.1)

Die Interpretation der Ergebnisse sorgt für Schwierigkeiten: obwohl der Solveranteil in allen 3 Fällen am meisten durch die Verwendung von ATLAS BLAS verringert wird, haben wir in 2 Fällen betragsmässig die beste Zeit bei Goto BLAS. Dieser Fakt wir dadurch interpretiert, dass parallel zu den Langzeitberechnungen andere Rechnungen liefen. Mit der Annahme, dass parallel laufende Berechnungen den relativen Anteil des Solvers an der Berechnung nicht beeinflussen, sondern nur die absolute Zeit, wird ATLAS BLAS für die bessere Variante erklärt.

Abb. (6.1) zeigt zwei der generierten Zeitprofile für eine der Langzeitberechnungen (kbt). Alle aufgelisteten Phasen (x-Achse) wurden tatsächlich ausgeführt. Die Zeitmessung zu jedem Zeitpunkt wurde auf ganze Sekunden abgerundet. Damit lief für ein Ereignis keine Zeit, solange dieses nicht das Inkrementieren einer Systemsekunde überschritt. Die y-Achse stellt den Anteil an der Gesamtberechnung dar.

Bei diesem Beispiel braucht die komplette Berechnung mit der alten tochnog3.1 fast 38 Stunden. Etwa 50 % der Laufzeit entfällt für die Phase calculate, an zweiter Stelle ist dann der Solver mit 35 %. Rechts sieht man ganz deutlich den Abfall des Solveranteils (der dritte Balken von links) auf 6.7 %. Die Gesamtberechnung benötigt dann noch 27.5 Stunden.

Die Ergebnisse der Berechnungen der nichtoptimierten und der ATLAS-BLAS-optimierten Version von tochnog wurden z.B. bei der kbt-Berechnung graphisch anhand von GiD verglichen. Einige Zahlenwerte wie Maximum und Minimum der Verschiebung in jeder Dimension wurden auch verglichen und haben übereingestimmt. Ob die BLAS-Anwendung zu Genauigkeitsverlusten bei tochnog führt (und in welchem Ausmass) wurde hier nicht genauer untersucht.

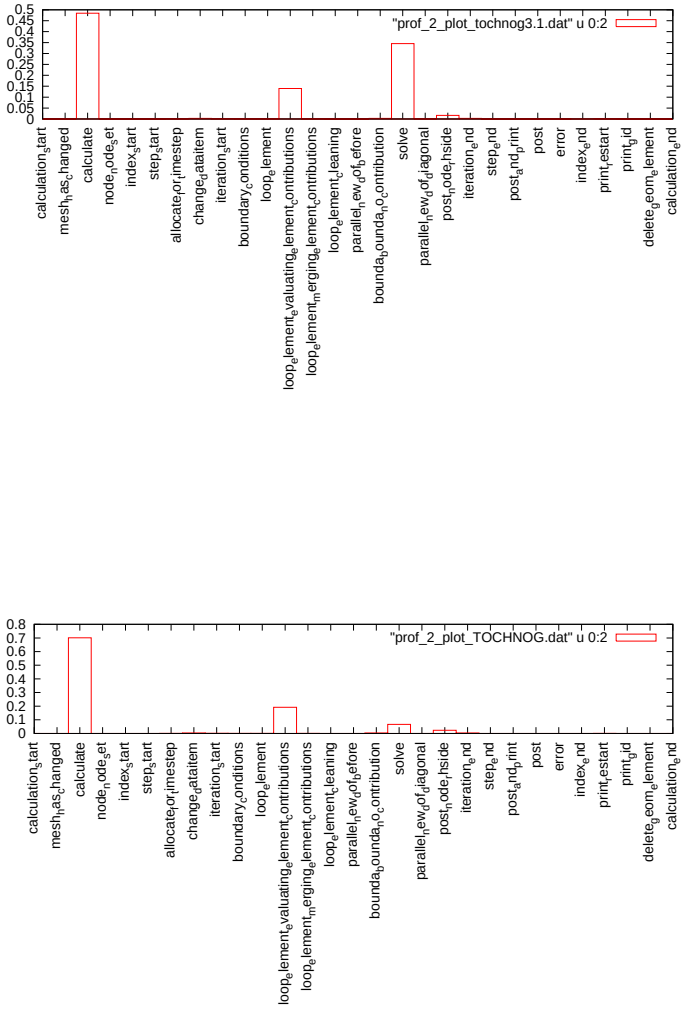

Abbildung 6.1: tochnog3.1 ohne (links) und mit BLAS (rechts)

7 Zusammenfassung

Der Ausgangspunkt für die meisten Untersuchungen in dieser Arbeit waren zwei Testmatrizen, die in der Anfangsphase von verschiedenen Berechnungen generiert wurden. Im Laufe der Untersuchungen zeigte sich die BMBF-Matrix als besser geeignet für Optimierungstests. Am Anfang hat sich der deutliche Effekt von BLAS Bibliotheken gezeigt. Das dürfte eigentlich keine Überraschung sein, da diese speziell für höhere Effizienz der algebraischen Operationen implementiert sind. Die Frage, ob die BLAS Parallelisierung den Solver optimiert, kann man aber nicht eindeutig beantworten: die Antwort lautet ja für schwierige Matrizen, d.h. für Matrizen mit grossen dichten Blöcken. UMFPack hat bei der Zerlegung der BMBF-Matrix in dichten Matrizen einen sehr grossen dichten Block gebildet. Das führte bei einer Matrix-Matrix-Multiplikation ohne BLAS zu extrem langen Laufzeiten. Bei der Untersuchung von zwei alternativen Solvern hat man die positive Wirkung von einer Parallelisierung auf der Algorithmus-Ebene gesehen. Dabei ist für jede Matrixart eine Reduzierung der Laufzeit mit Erhöhung der Parallelität zu betrachten. Der Solver Pardiso zeigte hier bei beiden Matrizen eine optimale Laufzeit.

Im Anschluss wurde die Laufzeit von drei tochnog-Versionen bei verschiedenen Langzeitberechnungen gemessen. Alle Versionen verwenden den UMFPack-Solver, aber zwei verbesserte Versionen sind unter Ausnutzung von BLAS kompiliert. Damit konnte man beobachten, inwieweit der Solver die Gesamtzeit des FE-Programms reduziert. Die Ergebnisse fielen sehr eindeutig aus: am Beispiel ATLAS BLAS hat man zwischen 27.5 % und 53 % der Gesamtlaufzeit reduziet. tochnog hat offensichtlich ein sehr grosses Optimierungspotential.Die tochnog-Version mit ATLAS BLAS Nutzung von UMFPack lieferte bei einer Langzeitberechnung gleiche Ergebnisse wie die nichtoptimierte Version. Man sollte aber unbedingt auch über das Einbinden von einem Solver wie Pardiso nachdenken. Aus den bisherigen Ergebnissen (vergleiche Abb. 5.1) kann man schließen, dass das Programm dadurch weiter an Effizienz gewinnen würde.

Literaturverzeichnis

[1] R.J. Astley. Finite Elements in Solids and Structures: An Introduction, Chapman & Hall, London, 1993

[2] Klaus-Jürgen Bathe. Finite Element Procedures, Prentice-Hall of India, Private Limited, 2003

[3] Tochnog Professional User's Manual - FEAT
`http://www.feat.nl/tochnog/tnu/tnu.html`

[4] Intel Compiler
`http://www.intel.com/support/performancetools/c/linux/sb/CS-022672.htm`

[5] Dokumentation zu SGI-Optimierungen
`http://techpubs.sgi.com`

[6] UMFPack- von Timothy Davis und Iain Duff
`http://www.cise.ufl.edu/research/sparse/umfpack/`

[7] Timothy Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method, May 6, 2003

[8] Pardiso- von Olaf Schenk und Klaus Gärtner
`http://www.computational.unibas.ch/cs/scicomp/software/pardiso/`

[9] O.Schenk, K.Gärtner, and W.Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors, BIT, 40(1):158-176,2000.

[10] O.Schenk and K.Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. Journal of Future Generation Computer Systems,20(3):475-487,2004.

[11] O.Schenk and K.Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report,Department of Computer Science,University of Basel,2004. Submitted.

[12] WSMP- von Anshul Gupta
`http://www-users.cs.umn.edu/~agupta/wsmp.html`
`http://www.alphaworks.ibm.com/tech/wsmp`

[13] Goto BLAS - von K. Goto
`http://www.tacc.utexas.edu/resources/software/`

[14] Atlas BLAS
http://math-atlas.sourceforge.net/

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Stuttgart, 29.05.2006