FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Untersuchung des parallelen Gleichungslösers Paragon ProSolver-DES – slab solver – auf dem Rechner Intel Paragon XP/S-10

Guido Zavagli, Heribert Burg

KFA-ZAM-IB-9506

März 1995
(Stand 08.03.95)

Inhaltsverzeichnis

1 Einführung

Die Modellierung physikalischer Vorgänge führt in vielen Fällen auf das Problem der Lösung partieller Differentialgleichungen. Da solche Systeme meist nicht geschlossen gelöst werden können, werden Diskretisierungsverfahren angewandt. Ein Beispiel dafür ist die Methode der Finiten Elemente. Die partiellen Differentialgleichungen werden auf diese Weise durch ein lineares Gleichungssystem angenähert, dessen Lösung mittels eines Rechners erfolgen kann.

Die Rechenleistung von Rechnersystemen mit einem einzigen oder wenigen Prozessoren reicht oftmals nicht aus, um große lineare Gleichungssysteme in vertretbarem Zeitrahmen zu lösen. Der Einsatz massiv-paralleler Rechnersysteme kann hier Abhilfe schaffen, wenn die verwendeten Programme optimal auf die Rechnerstruktur zugeschnitten sind.

Die Firma Intel vertreibt das massiv-parallele Rechnersystem Intel Paragon XP/S [1,2]. Zur Lösung linearer Gleichungssysteme auf diesem Rechner wurde die Bibliothek Intel Paragon ProSolver entwickelt. Sie steht beispielsweise in Konkurrenz zur public domain Software ScaLAPACK [3]. Die ProSolver-Bibliothek besteht aus vier Teilpaketen:

- Paragon ProSolver-SES
- Paragon ProSolver-IES
- Paragon ProSolver-DES
- Paragon ProSolver-FFT

Das ProSolver-SES-Paket (Skyline Equation Solver) enthält Routinen, die lineare Gleichungssysteme mit dünnbesetzter Koeffizientenmatrix anhand einer LR-Zerlegung lösen. Dünnbesetzte Matrizen werden nach dem Skyline-Prinzip gespeichert. Hierbei wird jede Zeile und Spalte ab dem ersten Element, das ungleich Null ist, bis zur Hauptdiagonalen abgespeichert. Darüberhinaus stehen Funktionen zur Zusammenstellung der Matrizen zur Verfügung. Die Daten können wahlweise reell oder komplex mit doppelter Genauigkeit sein.

Das ProSolver-IES-Paket (Iterative Equation Solver) löst ebenfalls lineare Gleichungssysteme mit schwachbesetzter Koeffizientenmatrix. Im Gegensatz zum ProSolver-SES wird hier ein iteratives Verfahren angewandt, das CG-Verfahren (Conjugate Gradients). Optional wird eine Vorkonditionierung des Systems durch unvollständige Cholesky-Faktorisierung angeboten. Die ProSolver-IES-Routinen benutzen die DME-Bibliothek (Distributed Matrix Environment) für die Verwaltung der Matrizen auf dem Parallelrechner. Die DME-Bibliothek wird herangezogen, um Matrizen zu erzeugen, Werte und Attribute einer Matrix festzulegen und die Datenaufteilung der Matrix auf die beteiligten Knoten zu spezifizieren. Die Lösung eines Gleichungssystems erfordert also sowohl den Einsatz der DME-Bibliothek als auch des ProSolver-IES-Paketes. Mit den DME-Routinen werden sämtliche Matrizen (d.h. Koeffizientenmatrix, rechte Seite und Lösungsvektor) erzeugt, auf die Knoten verteilt und gefüllt. Die Matrizen werden anschließend in die IES-Anwendung eingebunden und eventuell vorkonditioniert. Danach kann der Lösungsvektor ausgerechnet werden. Zum Schluß werden die Matrizen wiederum mit einer DME-Funktion gelöscht.

Gleichungssysteme mit dichtbesetzter Koeffizientenmatrix werden mit dem ProSolver-DES-Paket auf der Basis der LR-Zerlegung gelöst. Dieses wird im nächsten Abschnitt ausführlich beschrieben.

Das ProSolver-FFT-Paket (Fast Fourier Transform) beinhaltet Routinen, die Daten aus dem Zeitbereich in den Frequenzbereich transformieren, bzw. umgekehrt. Es stehen Versionen für Probleme im 2- bzw. 3-dimensionalen Raum zur Verfügung. Alle Routinen bearbeiten Daten, die spalten- bzw. flächenweise auf die Knoten des Paragon-Rechners verteilt sind.

ProSolver-SES und -DES stehen in zwei Versionen zur Verfügung. Bei der *in-core*-Version wird nur der Hauptspeicher der beteiligten Knoten benutzt. Bei der *out-of-core*-Version werden die Daten auf dem parallelen File-System (PFS) abgelegt .

Vor Beginn der Untersuchungen, gegen Ende 1994, waren die Löserpakete SES und IES zur Lösung dünnbesetzter Systeme nicht mehr Bestandteil des offiziellen Lieferumfanges der Paragon ProSolver-Bibliothek. Daher beschränkten sich die Untersuchungen über parallele Gleichungslöser, die im Rahmen von Studienarbeiten durchgeführt wurden, auf den ProSolver-DES und hier auf die *out-of-core*-Routinen, da das Interesse der Behandlung sehr großer Systeme galt. Dieser Bericht befaßt sich dem *slab solver*-Teil des DES-Paketes, dem die Studienarbeit von Herrn Zavagli [6] zugrundeliegt.

2 Programmpaket ProSolver-DES

Der ProSolver-DES [5] ist ein Programmpaket, das auf der Basis der LR-Zerlegung große lineare Gleichungssysteme mit komplexen Zahlen auf dem Paragon-Parallelrechner löst. Der ProSolver-DES ist zur Zeit in der Version 1.2 erhältlich.

Die (*out-of-core*)-Routinen des Paketes zielen auf sehr große Gleichungssysteme, bei denen der Hauptspeicher des einzelnen Knoten nicht ausreicht, um das komplette System zu speichern. Daher werden die Daten auf dem PFS (Parallel File System) [2] abgelegt und blockweise in den Hauptspeicher eingelesen, um dort verarbeitet zu werden.

Bei dem ProSolver-DES können für das Gleichungssystem

$$Ax = b$$

mehrere rechte Seiten definiert werden. Dadurch entstehen mehrere Lösungsvektoren x. Die rechten Seiten und die Lösungsvektoren werden jeweils zusammengefaßt und als Matrizen betrachtet, die genau wie die Koeffizientenmatrix in Blöcke aufgeteilt werden.

Die ProSolver-Routinen arbeiten mit Blöcken fester Größe. Die optimale Größe der Blöcke wird bei der Erzeugung der Koeffizientenmatrix ermittelt. Falls sich die Koeffizientenmatrix nicht in optimale Blöcke teilen läßt, wird sie entsprechend erweitert. Die erweiterten Bereiche werden vom ProSolver-DES automatisch gefüllt und beinhalten Einsen auf der Hauptdiagonalen und sonst Nullen.

2.1 Varianten

Der ProSolver-DES verwendet wahlweise zwei Algorithmen zur Gleichungslösung, den *square-solver* und den *slab-solver*. Beide Algorithmen arbeiten mit komplexen Zahlen und benutzen die LR-Blockzerlegung mit anschließender Rücksubstitution.

2.1.1 *slab-solver*

Der *slab-solver* verwendet eine LR-Blockzerlegung mit vollständiger Spaltenpivotisierung. Die Koeffizientenmatrix wird in Spaltenblöcke (*slabs*) fester Breite geteilt. Jeder Spaltenblock wird wiederum so auf die Prozessoren verteilt, daß alle Prozessoren die gleiche Anzahl von Zeilen bekommen (Abb. 1). Da der *slab-solver* zu einem bestimmten Zeitpunkt nur mit einem Spaltenblock arbeitet, kann die Pivotelementsuche auf einer kompletten Spalte erfolgen. Falls das gefundene Pivotelement Null ist, bricht das Programm ab. Darüberhinaus besteht die Möglichkeit, eine untere Grenze für das Pivotelement anzugeben. Unterschreitet das Pivotelement diese Grenze, bricht das Programm wiederum ab.

Die Anzahl der Spalten in einem Block muß ein ganzes Vielfaches der Prozessorzahl sein. Falls es notwendig ist, kann der slab-solver die Koeffizientenmatrix erweitern (in Abb. 1 mit gestrichelter Linie dargestellt). Dabei müssen folgende Gleichungen gültig bleiben:

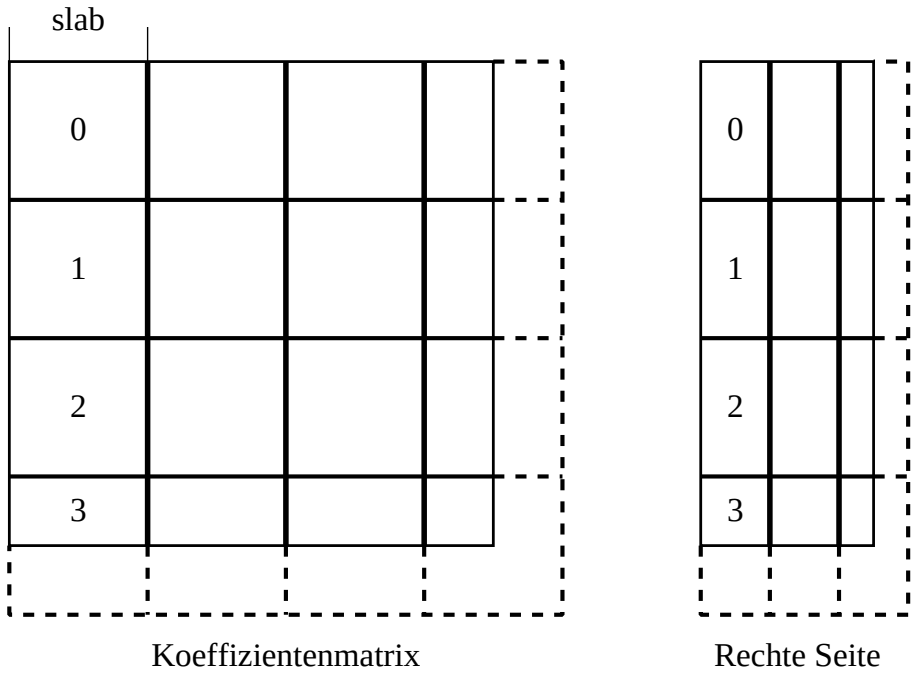

Abbildung 1: Blockzerlegung beim slab-solver

- $na = np * nrow$
- $na = nacol * naslab$
- $nacol = np * i$

In diesen Gleichungen ist *na* die Größe der Matrix (Zeilenanzahl), *np* die Prozessorzahl, *nrow* die Anzahl von Zeilen, die jeder Prozessor bekommt, *nacol* die Breite der Spaltenblöcke, *naslab* die Anzahl der Spaltenblöcke in der Matrix und i ist eine ganze Zahl.

Die Matrix der rechten Seiten wird analog zur Koeffizientenmatrix in Spaltenblöcke unterteilt. Auch hier müssen die Spaltenblöcke die gleiche Breite haben, die mit der Spaltenblockbreite der Koeffizientenmatrix nicht übereinstimmen muß. Die Matrizen im Gleichungssystem müssen alle die gleiche Anzahl von Zeilen besitzen. Notfalls wird auch hier mit der Einheitsmatrix erweitert.

2.1.2 *square-solver*

Der *square-solver* unterscheidet sich vom slab-solver in der Blockzerlegung der Matrizen. Hier wird die Koeffizientenmatrix in quadratische Blöcke unterteilt. Daraus folgt, daß der square-solver nur mit einer quadratischen Anzahl von Prozessoren arbeiten kann.

Die Matrix wird in sogenannte *disk-sections* und *node-sections* zerlegt. In Abb. 2 ist ein Beispiel zu sehen. Der Algorithmus fordert, daß es gleich viele disk-sections entlang der Zeilen als auch entlang der Spalten gibt. Falls es notwendig ist, kann auch hier mit der Einheitsmatrix erweitert werden.

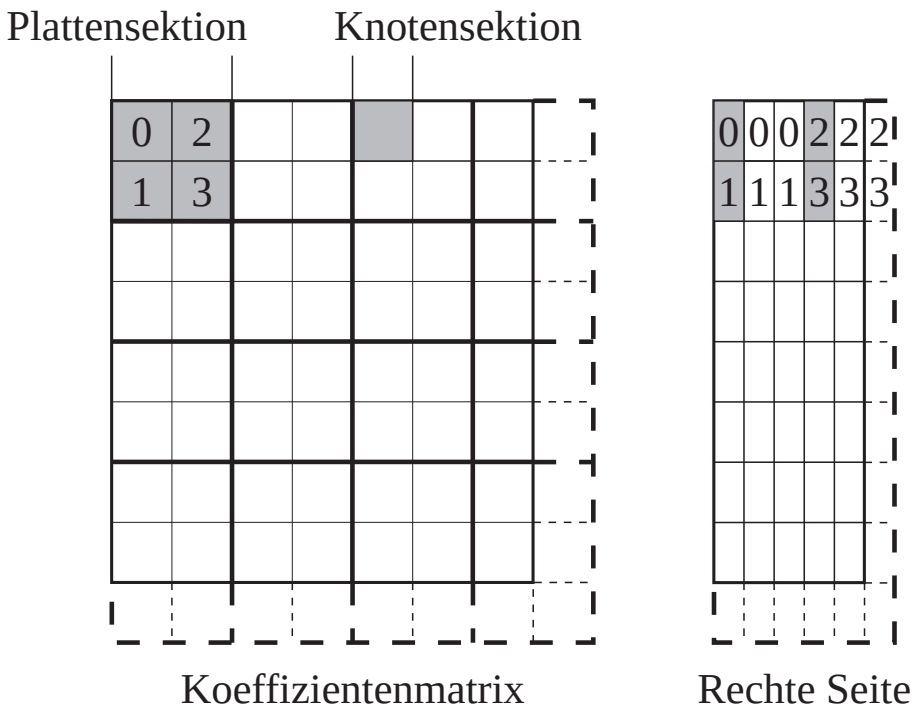

Abbildung 2: Blockzerlegung beim square-solver

Die rechtsseitige Matrix wird analog zur Koeffizientematrix zerlegt. Die Blockbreite kann kleiner sein, beide Matrizen müssen aber gleich viele Zeilen besitzen.

Der square-solver verwendet auch das Verfahren der LR-Blockzerlegung mit Spaltenpivotisierung. Allerdings kann wegen der quadratischen Blockzerlegung (s. Abb. 2) die Spaltenpivotsuche nicht auf einer gesamten Spalte der Koeffizientenmatrix erfolgen.

2.2 Anpassung des Parallel File System für den ProSolver

Die Leistung der ProSolver-DES-Routinen hängen sehr stark von den Leistungen des PFS ab, da alle Daten dort abgelegt werden. In diesem Abschnitt wird gezeigt, wie das PFS zu installieren ist, um den ProSolver-DES mit optimaler Effizienz zu betreiben.

Das PFS, wie es auf dem Paragon-Rechner der KFA Jülich installiert ist, wird in [2] beschrieben; dort wird auch darauf hingewiesen, daß zwei PFS mit verschiedenen Blockgrößen zur Verfügung stehen. Die meisten Anwendungen, so auch das im Rahmen dieser Arbeit untersuchte Testprogramm, benutzen das PFS mit der Blockgröße von 64 KB, das zugleich das größere von beiden ist. Dieses PFS erstreckt sich über insgesamt fünf RAID-Systeme, die jeweils mit einem I/O-Knoten verbunden sind. Dateien, die auf dem PFS abgelegt sind, werden blockzyklisch auf die RAID-Systeme aufgeteilt.

Bei einer Anwendung mit dem DES-slab-solver öffnet jeder Knoten eine Datei auf dem PFS. In der augenblicklichen Konfiguration wird der Dateianfang möglichst auf das RAID-System des ersten I/O-Knotens gesetzt. Eine mögliche Verteilung der Dateien ist in Abb. 3 zu sehen. In diesem Beispiel legen 100 Rechenknoten jeweils eine Datei auf dem PFS ab.

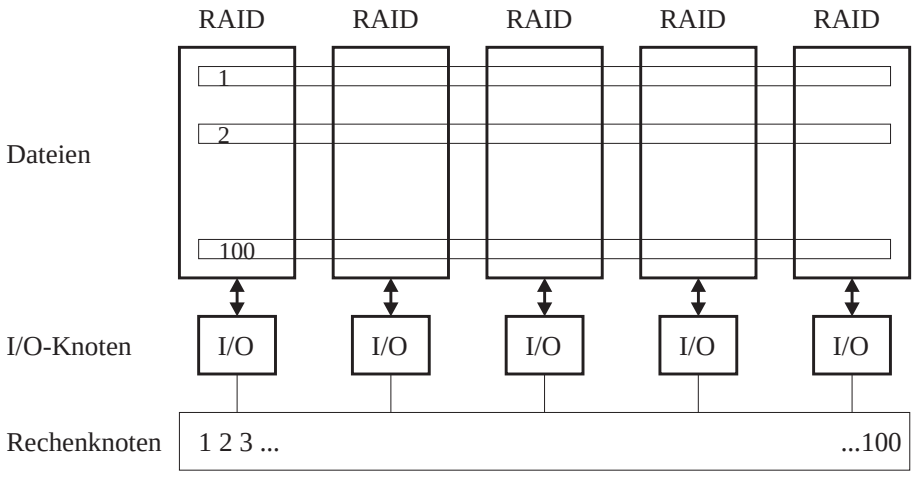

Abbildung 3: Dateiverteilung auf dem PFS

Daß diese Konfiguration für den ProSolver-DES nicht optimal ist, wird mit folgender Überlegung klar. Die Koeffizientenmatrix wird zeilenblockweise auf die Knoten verteilt. Da sich der Faktorisierungsvorgang zu einem bestimmten Zeitpunkt mit einem slab (Spaltenblock) beschäftigt, greifen alle Knoten auf Daten zu, die sich ungefähr an der gleichen Stelle in der jeweiligen Datei befinden. Das hat zur Folge, daß die Rechenknoten fast alle den gleichen I/O-Knoten in Anspruch nehmen. Der Grundvorteil des PFS, mit mehreren I/O-Knoten auf Dateien zugreifen zu können, wird somit nicht ausgenutzt.

Aus diesen Gründen wird eine besondere Konfiguration für das PFS vorgeschlagen. Die optimale Konfiguration kann für den slab-solver anders als für den square-solver sein. Wir beschränken uns auf die Konfiguration für den slab-solver, da er Gegenstand der Untersuchung ist.

Im folgenden wird stichpunktartig vorgestellt, welche Eigenschaften die optimale Konfiguration hat:

- Falls die Anzahl von Dateien größer ist als die Anzahl von I/O-Knoten, wird auf jedem I/O-Knoten ein separates PFS installiert. Dieser Fall wird in den meisten Fällen eintreten, da beim slab-solver jeder Knoten eine Datei öffnet und weil das Verhältnis von I/O-Knoten zu Rechenknoten meistens relativ klein ist. Das Verhältnis von Rechenknoten zu I/O-Knoten beträgt im Forschungszentrum Jülich 138:6, also 23:1.

- Auf jedem RAID-System werden mindestens vier, möglicherweise fünf Partitionen angelegt. Drei werden für die Datenspeicherung auf dem PFS benutzt. Das PFS wird so installiert, daß die Daten blockzyklisch auf die drei Partitionen verteilt werden. Die vierte Partition ist klein und beinhaltet den sog. *mount point* für das PFS. Die fünfte Partition ist notwendig, falls der Anwender sowohl den square-solver als auch den slab-solver benutzt und beide Konfigurationen gleichzeitig vorhanden sein sollen.

- Die Blockgröße des PFS wird auf 64 KB gesetzt.

Da jeder I/O-Knoten jetzt ein separates PFS steuert, können die Rechenknoten auf die I/O-Knoten verteilt werden. Wenn das System z.B. 100 Rechenknoten und 5 I/O-Knoten besitzt, werden jeweils 20 Rechenknoten einem I/O-Knoten, und damit einem PFS, zugeordnet (Abb. 4). Man erzwingt den Parallelismus beim Datenzugriff nicht mehr dadurch, daß das PFS sich über mehrere I/O-Knoten erstreckt, sondern dadurch, daß mehrere PFS zur Verfügung stehen.

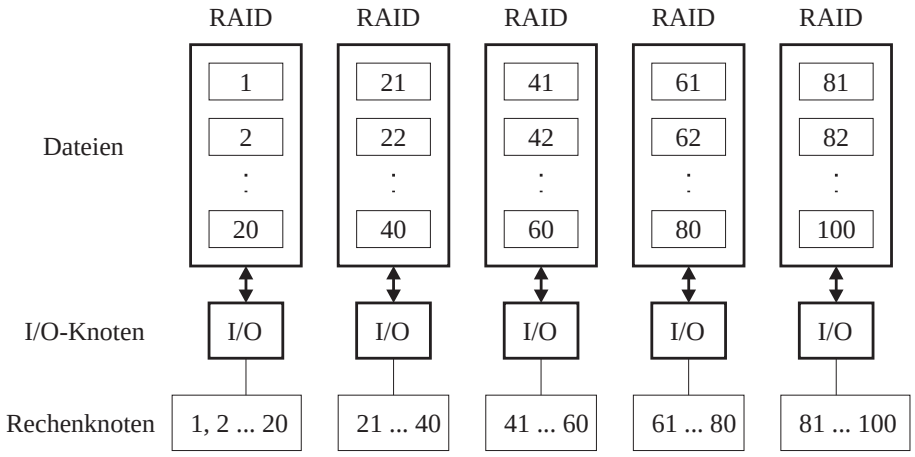

Abbildung 4: Dateiverteilung auf dem PFS mit der neuen Konfiguration

Im ProSolver-DES-Handbuch [5] werden die notwendigen Schritte zur Konfiguration des PFS im einzelnen erläutert. Die Installation setzt allerdings den Zugang zum Rechner als *host* voraus.

Obwohl die vorgeschlagene Konfiguration optimal für den ProSolver-DES ist, kann sie nicht als allgemeines Beispiel für andere Anwendungen gesehen werden. Außerdem ist es aus Platzgründen nicht möglich, mehrere PFS-Konfigurationen gleichzeitig zu installieren. Aus diesen Gründen war es nicht möglich, den ProSolver-DES mit dieser speziellen Konfiguration des PFS zu testen.

2.3 Arbeiten mit dem ProSolver-DES

Der Vorgang zur Lösung eines linearen Gleichungssystemes erfolgt in mehreren Schritten:

1. Initialisieren der Umgebung
2. Initialisieren und Füllen der Koeffizientenmatrix und der rechten Seiten
3. Faktorisieren der Koeffizientenmatrix und Lösen des Gleichungssystems
4. Auslesen des Ergebnisvektors und Schließen der Umgebung

2.3.1 Initialisieren der Umgebung

Der erste Schritt in einer Anwendung mit der ProSolver-Bibliothek besteht darin, die Prozedur **DES_create**(*path_name*) aufzurufen. Diese Prozedur erzeugt das Hauptverzeichnis

path_name auf dem PFS, das für die Datenhaltung benötigt wird. Alle weiteren Verzeichnisse werden dem Hauptverzeichnis untergeordnet.

DES_create() ist eine globale Prozedur, d.h. sie muß von allen Prozessoren aufgerufen werden. Um sicherzustellen, daß dies auch der Fall ist, wird intern eine globale Synchronisation durchgeführt.

Als nächstes können mit der Prozedur **DES_set_param**(*param, value*) die Parameter einzeln geändert werden. Dabei ist *param* der Name des Parameters, der auf *value* gesetzt werden soll. Beispielsweise erfolgt die Wahl zwischen dem slab-solver und dem square-solver durch das Setzen des entsprechenden Parameters. **DES_set_param** ist eine globale Funktion, die sicherstellt, daß alle Prozessoren mit den gleichen Parametern arbeiten.

Der Anwender kann die Speicherverwaltung der DES-Routinen steuern. Mit der Prozedur **DES_use_mem()** wird ein vom Anwender angegebener Speicherplatz zur Benutzung an Stelle des Systemsspeichers reserviert.

2.3.2 Initialisieren und Füllen der Matrizen

Jetzt können mit den Prozeduren **DES_mopen()** und **DES_vopen()** die Koeffizientenmatrix bzw. die Matrix der rechten Seiten geöffnet werden. Die Prozedur

DES_mopen(*matrix_dim, factor_dir, coef_matrix_id, nrow, ncol*)

bekommt als Eingabeparameter die Problemgröße *matrix_dim* und das Verzeichnis, wo die Daten abgelegt werden sollen (*factor_dir*). Folgende Werte werden zurückgegeben: die vom DES errechnete Blockgröße (*nrow* und *ncol*), und eine Kennummer *coef_matrix_id*, die bei jeder Operation mit der Koeffizientenmatrix benötigt wird.

Das Öffnen der rechten Seiten gestaltet sich analog mit:

DES_vopen(*coef_matrix_id, num_vectors, solve_dir, rhs_vectors_id, nrow, ncol*)

Um die Matrix der rechten Seiten eindeutig einer Koeffizientenmatrix zuzuordnen, muß die Kennummer der Koeffizientenmatrix angegeben werden.

Beide Prozeduren legen die benötigten Dateien bzw. Verzeichnisse an und initialisieren die Datenstrukturen. Sind beide Operationen erfolgreich durchgeführt worden, sieht der Verzeichnisbaum folgendermaßen aus:

path_name:
- *des_params*
- *factor_dir:*
  - *des_mparams*
  - *des_mrestart*
  - *des_pivot*
  - *des_matrix.###*
- *solve_dir:*
  - *des_vparams*
  - *des_vrestart*
  - *des_rhs.###*

Es gibt drei Parameterdateien: *des_params*, *des_mparams* und *des_vparams*. Die Datei *des_params* beinhaltet alle DES-Systemparameter. Befindet sich diese Datei beim Initialisieren der Umgebung im Verzeichnis *path_name*, so benutzt DES die Werte in dieser Datei. Ansonsten wird eine neue Datei erzeugt und mit den Standardwerten gefüllt. Die Prozeduren *DES_mopen()* und *DES_vopen()* lesen diese Datei und erzeugen wiederum eigene Parameterdateien, *des_mparams* und *des_vparams*.

Der ProSolver-DES speichert in regelmäßigen Abständen seinen augenblicklichen Stand in den Dateien *des_mrestart* und *des_vrestart* ab. Außerdem werden die Ergebnisse der Spaltenpivotsuche in der Datei *des_pivot* abgelegt. Falls diese Dateien beim Starten eines Faktorisierungsvorganges in den entsprechenden Verzeichnissen vorhanden sind, wird der ProSolver-DES versuchen, den offensichtlich abgebrochenen Programmlauf wieder aufzunehmen.

Die Matrizen werden in den Dateien *des_matrix.###* und *des_rhs.###* gespeichert. Während des Lösungsvorganges wird die Koeffizientenmatrix durch die faktorisierte Matrix und die Matrix der rechten Seiten durch die Lösungsvektoren ersetzt.

Bei dem slab-solver braucht jeder Knoten

$$(na * nrow * 16) + (nrow * nacol * 16) \quad Byte$$

freien Speicherplatz auf dem PFS. *na* ist die Spaltenanzahl der Koeffizientenmatrix, *nrow* die Anzahl von Zeilen, die jedem Knoten zugeordnet werden, *nacol* die Breite eines slab in Spalten und 16 die Größe eines Datenelementes (complex*16) in Byte. Der zweite Term in der Summe stellt einen Faktorisierungsblock dar, der bei der Wiederaufnahme einer abgebrochenen Rechnung benötigt wird.

Der Verlauf der Rechnungen wird in einer log-Datei abgelegt (s.u.).

Neben den Befehlen **des_mopen()** und **des_vopen()** gibt es **des_mopen_man** und **des_vopen_man**. Sie unterscheiden sich nur in der Tatsache, daß mit letzteren die Blockgröße vom Anwender gesetzt werden kann.

Der nächste Schritt besteht darin, die erzeugten Matrizen zu füllen. Dazu steht die Prozedur **DES_put()** zur Verfügung. Die genaue Syntax lautet:

DES_put(*maxtrix_id, num_rows, num_cols, row_index, col_index, data_buffer, l_data_buffer_b, io_id*)

Diese Prozedur ist nicht global, d.h. sie muß nicht von jedem Prozessor aufgerufen werden. **DES_put()** bearbeitet sowohl die Koeffizientenmatrix als auch die Matrix der rechten Seiten. Mit dem Parameter *matrix_id* kann man steuern, in welche Matrix die Daten geschrieben werden sollen.

DES_put() schreibt einen Block der Größe *num_rows* $\times$ *num_cols* mit der oberen linken Ecke an der Stelle *row_index*, *col_index*. Die obere linke Ecke muß sich auf der ersten Spalte oder Zeile eines Zerlegungsblockes der Matrix befinden. Andererseits darf der zu schreibende Block nicht größer sein als ein Zerlegungsblock.

Der Anwender muß sicherstellen, daß alle Elemente der Koeffizientenmatrix gefüllt werden, da der ProSolver-DES in dieser Hinsicht keine Initialisierung übernimmt. Lediglich um die erweiterten Bereiche muß sich der Anwender nicht selbst kümmern.

2.3.3 Faktorisieren und Lösen des Gleichungssystems

Mit der Prozedur **DES_factor**() wird die Koeffizientenmatrix faktorisiert. Die ursprüngliche Koeffizientenmatrix wird dann nach und nach durch die Matrizen L und R ersetzt.

Nach der Faktorisierung kann mit der Prozedur **DES_solve**() das Gleichungssystem gelöst werden. Die Lösungsvektoren werden in die Matrix der rechten Seiten geschrieben.

Die Prozeduren **DES_factor**() und **DES_solve**() sind global, müssen also von allen Prozessoren aufgerufen werden.

2.3.4 Auslesen des Ergebnisvektors und Schließen der Umgebung

Wenn der Lösungsvorgang beendet ist, können die Lösungsvektoren mit der Funktion **DES_get**() ausgelesen werden. **DES_get**() ist keine globale Funktion. Es gelten die gleichen Bedingungen für das Einlesen wie für das Schreiben von Blöcken mit der Prozedur **DES_put**().

Am Ende des Programms sollte der Anwender die Umgebung mit **DES_close**() schliessen. Die während des Laufes erzeugten Dateien müssen mit dem Befehl **DES_delete**(*path_name*) aus jedem Verzeichnis gelöscht werden, d.h. **DES_delete**() muß für jedes vom ProSolver-DES angelegte Verzeichnis aufgerufen werden. Wenn die Dateien nicht gelöscht werden, gibt der ProSolver-DES beim nächsten Lauf eine Fehlermeldung aus.

2.3.5 Log-Datei

Die log-Datei nimmt sämtliche Meldungen auf, die der ProSolver-DES während eines Programmlaufes erzeugt. Mit der Prozedur **DES_set_param**() kann die Erzeugung der log-Datei gesteuert werden. Falls sie erzeugt werden soll, wird sie *des_log* genannt und im aktuellen Verzeichnis abgelegt. Diese Standardeinstellungen können wiederum mit **DES_set_param**() geändert werden.

Die Nachrichten in der log-Datei erstrecken sich über zwei Zeilen und werden durch eine Leerzeile getrennt. Alle Meldungen haben das gleiche Muster:

message_severity: `node` *number, day, date, time*
message_text

Die erste Zeile gibt Aufschluß über die Knotennummer (*number*) und das Datum (*day, date, time*) der Nachricht. Die Meldungen werden in drei Kategorien eingestuft, die mit dem Parameter *message_severity* bezeichnet werden:

- **information**: Das Programm meldet eine erfolgreiche Operation, wie z.B. das Ändern der Systemparameter oder das Schreiben von Daten in die Matrizen.
- **warning**: Das Programm gibt eine Warnung aus, rechnet aber weiter. Die Richtigkeit der Lösung kann allerdings nicht garantiert werden.

- **error**: Das Programm hat einen Fehler gefunden und wird unmittelbar nach der Meldung abbrechen.

An einigen Beispielen soll jetzt erläutert werden, wie die Nachrichten in der log-Datei zu bewerten sind.

Beim Initialisieren der Umgebung (**DES_create**()) gibt der ProSolver-DES folgende Meldungen aus:

```
Information message from processor    0:  Thu Dec 15 16:41:54 1994
*******            ProSolver-DES  Release R1.2           *******

Information message from processor    0:  Thu Dec 15 16:41:54 1994
********     Copyright (c) 1992, 93, 94 Intel Corporation      *******

Information message from processor    0:  Thu Dec 15 16:41:57 1994
Initial restart value =        0
Information message from processor    0:  Thu Dec 15 16:42:01 1994
Initial restart value =        0
```

Der ProSolver-DES hat keine Daten von einem vorher abgebrochenen Lauf gefunden und wird also bei Null anfangen.

Wie oben erwähnt, werden die Matrizen blockweise geschrieben. Der ProSolver-DES meldet, an welcher Stelle der Block in der Matrix geschrieben wird und wie viele Zeilen bzw. Spalten er enthält.

```
Information message from processor    0:  Thu Dec 15 16:42:03 1994
Start Insert i = (     1,     1), l = (   504,   144)
Information message from processor    0:  Thu Dec 15 16:42:03 1994
Start Insert i = (     1,   145), l = (   504,   144)
...
```

Wird die Prozedur **DES_factor**() zum Faktorisieren der Koeffizientenmatrix aufgerufen, gibt der ProSolver-DES folgende Meldungen:

```
Information message from processor    0:  Thu Dec 15 16:42:36 1994
Beginning factor at column        0

Information message from processor    0:  Thu Dec 15 16:42:39 1994
Slab        1 successfully completed, rcond_est =   0.45012E-06
Information message from processor    0:  Thu Dec 15 16:42:44 1994
Slab        2 successfully completed, rcond_est =   0.12962E-04
...
Factorization successfully completed, rcond_est =   0.4501196E-06
```

Bei dem slab-solver wird ein Spaltenblock (slab) nach dem anderen gelöst, beginnend mit der linken Seite. Zum Schluß wird noch eine Meldung ausgegeben, daß der Faktorisierungsvorgang erfolgreich beendet wurde.

Der Lösungsvorgang des Gleichungssystems wird in ähnlicher Weise dokumentiert:

```
Information message from processor    0:  Thu Dec 15 16:44:48 1994
Beginning solve at column        0
Information message from processor    0:  Thu Dec 15 16:44:56 1994
Slab        1 successfully completed
...
Solve successfully completed
```

Zum Schluß wird der Lösungsvektor ausgelesen. Die Meldungen dazu sehen wie beim Schreiben der Matrizen aus:

```
Information message from processor    1:  Thu Dec 15 16:44:56 1994
Start Extract i = (   505,     1), l = (   504,     1)
...
```

2.3.6 Programmstart und -lauf

Um die ProSolver-DES-Bibliothek in ein Programm einzubinden, müssen beim Linken die Parameter **-ldes** und **-lkmath** angegeben werden. Der Aufruf des Cross-Compilers für den Paragon-Rechner sieht z.B. folgendermaßen aus:

if77 *deststf -nx -ldes -lkmath -o deststf*

Dabei ist *deststf* das Fortran-Programm und *deststf* die ausführbare Datei, die vom Linker erzeugt wird.

Mit dem Befehl **pexec** kann das Programm auf dem Paragon-Rechner gestartet werden:

pexec *deststf -sz 4 &*

Der Parameter *-sz 4* bedeutet, daß das Programm auf 4 Knoten laufen soll.

Die log-Datei des ProSolver-DES bietet die Möglichkeit, den augenblicklichen Stand des Lösungsvorganges zu erfahren. Dazu verwendet man den UNIX-Befehl *more* in folgender Weise:

more *-w +g des_log*

Der Befehl *more* druckt den Inhalt einer Datei auf dem Bildschirm aus. Mit den Parametern +g und -w wird sofort das Ende der Datei angezeigt und verhindert, daß das

Programm verlassen wird[1]. Wenn der ProSolver-DES Meldungen in die log-Datei schreibt, kann man sie mit Betätigen der Taste *d* (down) erreichen.

Wenn das Programm unerwartet abgebrochen wird, z.B. nach einer Fehlermeldung, bleiben die angelegten Dateien bestehen. Um einen erneuten Lauf zu ermöglichen, müssen sämtliche DES-Dateien auf dem PFS gelöscht werden, z.B. mit dem Befehl:

rm *-rf /pfs/group/user/solution*

Der Parameter *-rf* löscht rekursiv den gesamten Verzeichnisbaum ausgehend vom angegebenen Verzeichnis.

[1] Die Parameter +g und -w arbeiten so nur auf dem UNIX-System des Intel Paragon.

3 Messungen

3.1 Testumgebung

3.1.1 Testprogramm

Das für die Leistungsmessungen des ProSolver-DES eingesetzte Testprogramm ist im Anhang A abgedruckt. Im ersten Teil des Testprogramms werden die Variablen initialisiert. Danach werden mit **DES_create()** die Umgebung erzeugt und mit **DES_set_param()** die Systemparameter gesetzt. In diesem Fall wird der *slab-solver* aufgerufen und die maximale Blockgröße auf 1000×1000 gesetzt. Der ProSolver-DES arbeitet zu einem bestimmten Zeitpunkt mit einem kompletten Block von Daten. Da dieser Block im Hauptspeicher bearbeitet wird, ergibt sich die maximale Blockgröße aus dem verfügbaren Hauptspeicher auf jedem Rechenknoten (26 MB auf dem Paragon-Rechner des Forschungszentrums Jülich). Anschließend werden die Koeffizientenmatrix und die Matrix der rechten Seiten geöffnet, wobei im Testfall nur eine rechte Seite betrachtet wird.

Der nächste Schritt besteht darin, die erzeugten Matrizen zu füllen. Wie weiter oben schon erwähnt, widerspricht dabei das Verhalten des ProSolver-DES den Aussagen des Handbuches. Es stellte sich beim Testen heraus, daß die Daten in den Matrizen nur von dem Knoten gelesen und geschrieben werden können, dem sie bei der Zerlegung zugeteilt wurden. Es wird somit die Beteiligung aller Knoten an Lese- und Schreibvorgängen erzwungen, also eine Parallelisierung dieser Vorgänge. Für das Gleichungssystem wird eine Testmatrix von Gregory und Karney verwendet [4], deren Lösung bekannt ist. Da der ProSolver-DES mit komplexen Zahlen arbeitet, wird sowohl der Realteil als auch der Imaginärteil der Einträge in den Matrizen mit den Werten aus der Testmatrix gefüllt. Da dies auch für die rechte Seite gilt, ändert sich der Lösungsvektor, verglichen zum reellen Gleichungssystem, nicht.

Nach dem Schreiben der Daten in die Matrizen erfolgt eine Kontrolle, ob alle Werte richtig geschrieben wurden. Dabei stellte sich wiederum heraus, daß bei großen Prozessorzahlen nicht reproduzierbare Fehler auftreten. Eine Erklärung dafür könnte die Überlastung des PFS bei gleichzeitigem Zugriff einer großen Anzahl von Prozessoren sein, bzw. das im ZAM relativ ungünstige Verhältnis von I/O-Knoten zu Rechenknoten. Um diesen Fehler zu umgehen wird im Testprogramm dafür gesorgt, daß zu einem bestimmten Zeitpunkt nur ein Knoten Daten schreibt. Das Einlesen von Daten erfolgt hingegen immer fehlerfrei.

Der ProSolver-DES wird gegebenenfalls die Matrizen erweitern, um eine optimale Blockzerlegung zu ermöglichen. Erst danach werden die Zeilen blockweise auf die Knoten verteilt. Dabei kann es passieren, daß ein oder mehrere Knoten Zeilen zugeteilt bekommen, die sich ausschließlich im erweiterten Bereich befinden. Diese Bereiche werden vom ProSolver-DES selbst gefüllt. Deswegen überprüft das Testprogramm zunächst, ob sich ein Knoten nur im erweiterten Bereich befindet und überspringt gegebenenfalls den Schreibvorgang.

Das Schreiben an sich erfolgt blockweise (s.o.). Das Testprogramm errechnet zunächst die Größe des Blockes (*numrows* und *numcols*), berechnet die Daten und schreibt sie in das Feld *buff* und übergibt zum Schluß die Daten an die Prozedur **DES_put()**. Unmittelbar

nach dem Schreibvorgang wird die Koeffizientenmatrix vollständig eingelesen, um mögliche Fehler abzufangen.

Nach Beendigung des Lösungsvorganges kann der Lösungsvektor eingelesen werden – er befindet sich an Stelle der rechten Seite. Das Testprogramm ermittelt die Genauigkeit der errechneten Lösung, und schließt hinterher die Umgebung.

Zum Schluß werden die wichtigsten Daten (Laufzeit, Genauigkeit, usw.) für die Auswertung der Testläufe in eine spezielle Datei geschrieben.

3.1.2 Technische Anmerkungen

Die hier vorgestellten Untersuchungen befassen sich mit dem ProSolver-DES slab-solver. Der Einsatz des slab-solver ist mit einer beliebigen Anzahl von Knoten möglich. Aus Zeitgründen wurden bei den Testläufen für die Knotenzahlen 2er Potenzen und 136 gewählt (die größtmögliche Partition im Batch-Betrieb umfaßt 136 Knoten).

Die Messungen erfolgten mit dem Release 1.2 des ProSolver-DES. Die Testläufe wurden für eine bestimmte Knotenzahl mit verschiedenen Problemgrößen gefahren (Problemgröße bedeutet hier die Ordnung der Koeffizientenmatrix). Der gewählte Bereich liegt zwischen 500 und 10000. Prinzipiell sind größere Probleme möglich, solange genügend Platz auf dem PFS vorhanden ist. Doch aus Zeitgründen wurde auch hier auf zu lange Testläufe verzichtet. Dementsprechend wurde die obere Grenze der Problemgrößen von der Knotenzahl abhängig gemacht.

In den folgenden Abschnitten werden die Ergebnisse für die Genauigkeit und die Laufzeiten ausgewertet. Im letzten Abschnitt werden die Ergebnisse zusammengefaßt.

3.2 Fehlermessungen

Nach jedem Programmlauf wird der errechnete Lösungsvektor ausgelesen und mit der exakten Lösung verglichen, die anhand der bekannten Inversen der Koeffizientenmatrix ausgerechnet werden kann (s. Anh. B).

Der maximale Fehler ergibt sich aus

$$max(|x_i - x_i^*|), \qquad i = 1...n$$

und der durchschnittliche Fehler mit

$$\frac{\sum_{i=1}^{n} |x_i - x_i^*|}{n}$$

wobei x die errechnete und x^* die exakte Lösung ist.

In den Abbildungen 5 und 6 sind der maximale und der durchschnittliche Fehler dargestellt. Die Verläufe sehen qualitativ sehr ähnlich aus. Der durchschnittliche Fehler ist ungefähr um eine Zehnerpotenz kleiner als der maximale Fehler.

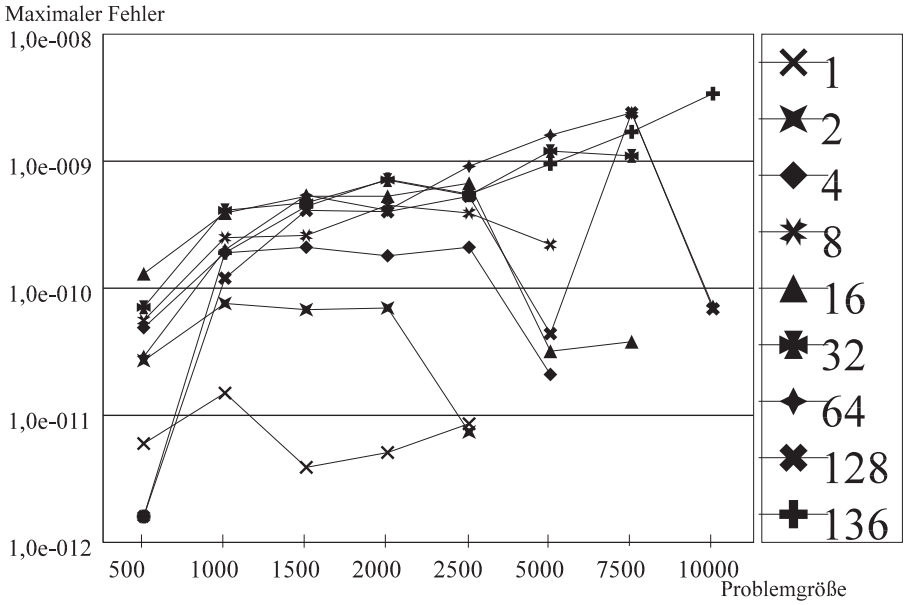

Abbildung 5: Maximaler Fehler für verschiedene Prozessorzahlen

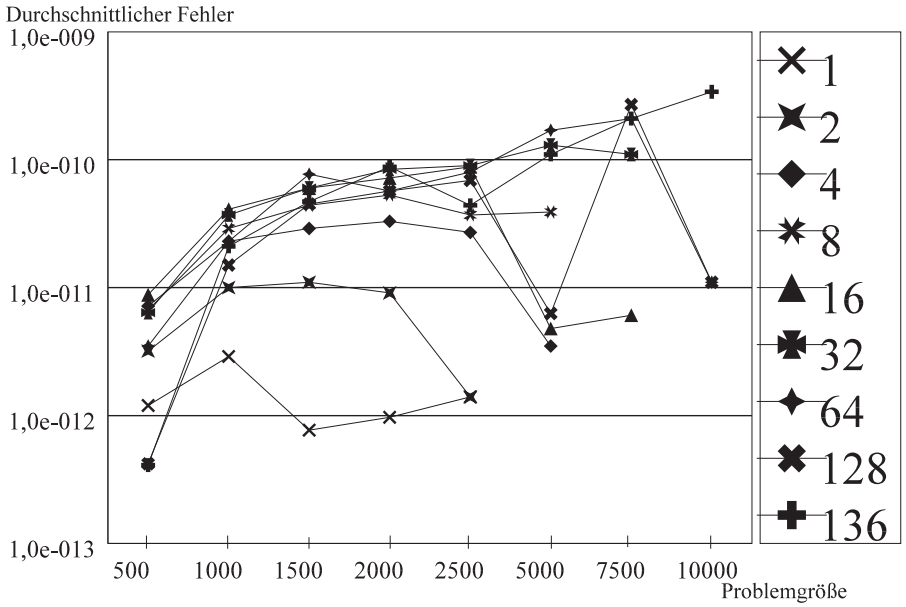

Abbildung 6: Durchschnittlicher Fehler für verschiedene Prozessorzahlen

3.3 Laufzeiten

Die Laufzeiten sind in den Abbildungen 7 und 8 in Abhängigkeit von den Problemgrößen dargestellt, wobei jede Kurve die Ergebnisse für eine Knotenzahl wiedergibt. Um eine bessere Übersicht zu bekommen, ist der gesamte Bereich der Problemgrößen in zwei kleinere unterteilt worden. Im Bereich der kleinen Problemgrößen (Abb. 7) wurden die Laufzeiten für große Prozessorzahlen weggelassen, weil sie sich von den Laufzeiten mit 64 Prozessoren kaum unterscheiden. Im Bereich der großen Problemgrößen (Abb. 8) konnten die Laufzeiten für kleine Prozessorzahlen aus Zeitgründen nicht ermittelt werden.

Anhand der Laufzeiten lassen sich der Speedup und die Effizienz ausrechnen (Abb. 9 und 10). Als Bezugswerte wurden die Laufzeiten mit einem Prozessor gewählt. Einige Werte für die Effizienz sind größer eins, was theoretisch nicht möglich ist. Dies ist darauf zurückzuführen, daß die Anwendung eines parallelen Algorithmus auf einem einzigen Prozessor merkbare Nachteile mit sich bringt.

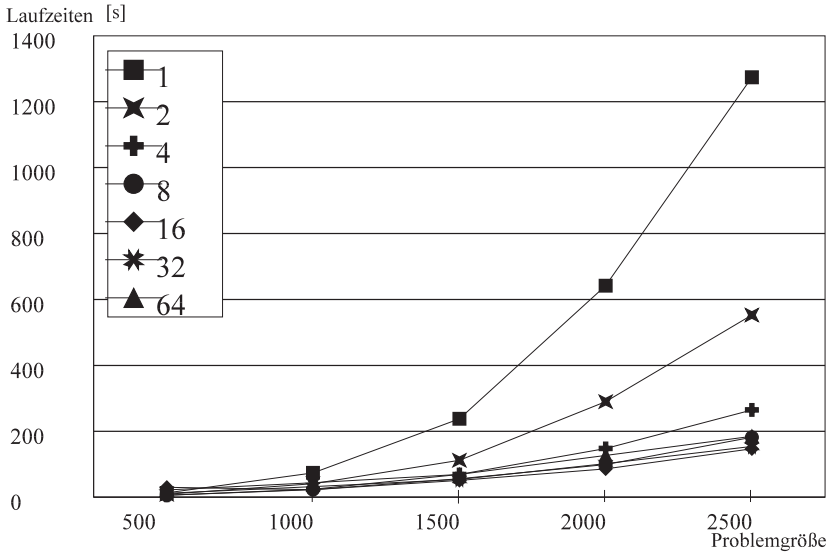

Abbildung 7: Laufzeiten für verschiedene Prozessorzahlen

Für kleine Problemgrößen verhalten sich die Laufzeiten erwartungsgemäß (Abb. 7). Die Speedup-Werte (Abb. 9) sinken relativ für steigende Knotenzahlen stark ab. Für kleine Problemgrößen wirkt sich also eine große Anzahl von Knoten als nachteilig aus. Dies wird auch verständlich, wenn man folgende Überlegung macht: Bei gleicher Problemgröße ist die Datenmenge, die jeder Knoten bearbeitet, umgekehrt proportional zur Anzahl von Knoten. Andererseits erreicht das PFS die besten Übertragungsleistungen, wenn wenige Prozesse auf ihn zugreifen und dabei möglichst große Mengen an Daten lesen bzw. schreiben. Aus diesen Gründen schneidet der ProSolver-DES bei kleinen Problemgrößen für große Prozessorzahlen relativ schlecht ab.

Im Bereich der großen Problemgrößen konnten keine Speedup-Werte ermittelt werden, weil die Läufe mit einem Knoten extrem viel Testzeit erfordert hätten. Bei Betrachtung der Abb. 8 stellt man allerdings fest, daß sich die Meßkurven für die Laufzeiten sehr stark bündeln, d.h. sie unterscheiden sich kaum bzw. überschneiden sich. Zum Beispiel liegen

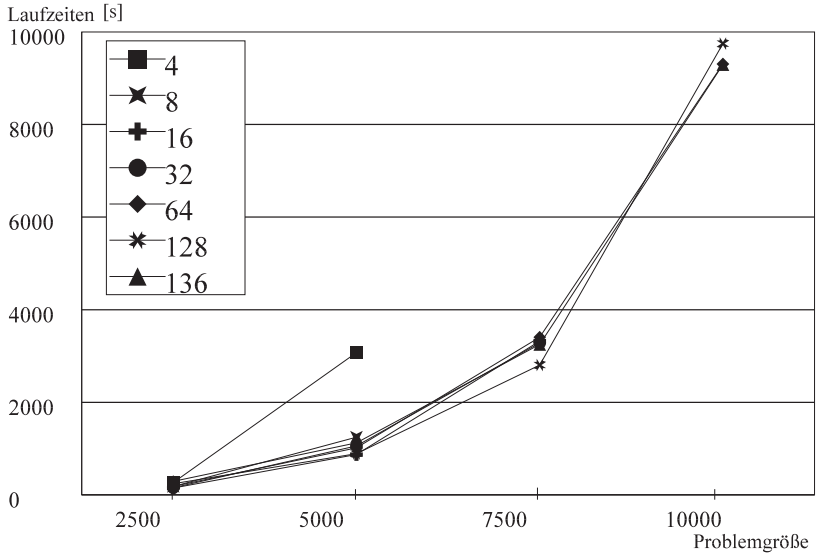

Abbildung 8: Laufzeiten für verschiedene Prozessorzahlen

die Laufzeiten für ein Gleichungssystem der Problemgröße 5000 und für Knotenzahlen von 16 bis 136 in einem Bereich von ungefähr 900 bis 1100 Sekunden. Bei einem derartigen Verhalten wird die Untersuchung der Effizienz keine weiteren Erkenntnisse bringen.

Das Verhältnis von Rechenknoten zu I/O-Knoten – wobei hier nur die I/O-Knoten zählen, die an einem RAID-System angeschlossen sind – ist für die Leistung des ProSolver-DES entscheidend. In der Beschreibung der optimalen Installation des PFS im ProSolver-DES-Handbuch wird ein Beispiel mit 100 Rechenknoten und 20 I/O-Knoten vorgestellt, Zahlen die ein Verhältnis von 5:1 ergeben. Im ZAM stehen 5 I/O-Knoten für das PFS zur Verfügung. Es entspricht daher den Erwartungen, daß die besten Ergebnisse mit Knotenzahlen von 16 bzw. 32 erreicht werden. Der Einsatz größerer Knotenzahlen verbessert die Laufzeiten nicht, weil die Leistungsgrenze des PFS erreicht ist.

Schwankungen im Verlauf der Kurven sind darauf zurückzuführen, daß nur in den seltensten Fällen eine Anwendung auf dem Parallelrechner dediziert läuft. Da die meisten Anwendungen einen, wenn auch nur geringen, Gebrauch vom PFS machen, können Konflikte beim Zugriff auf das PFS entstehen, was sich bei den ProSolver-Routinen besonders bemerkbar macht.

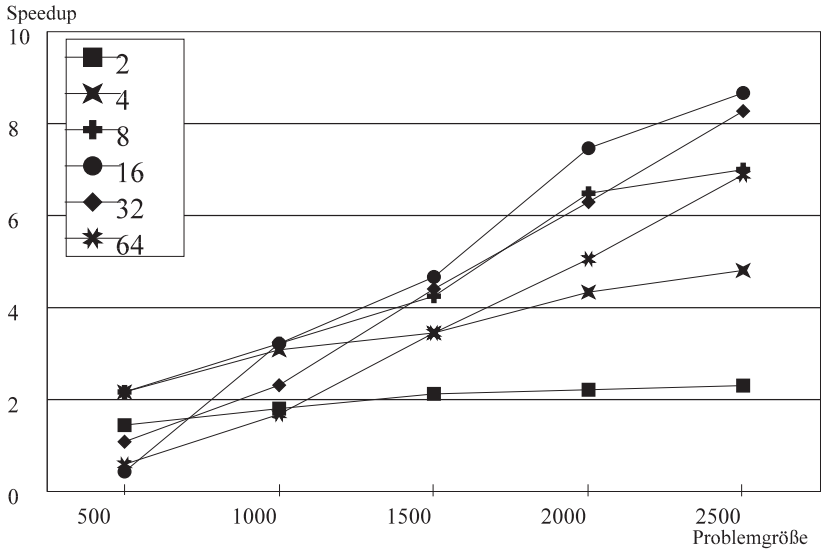

Abbildung 9: Speedup für verschiedene Prozessorzahlen

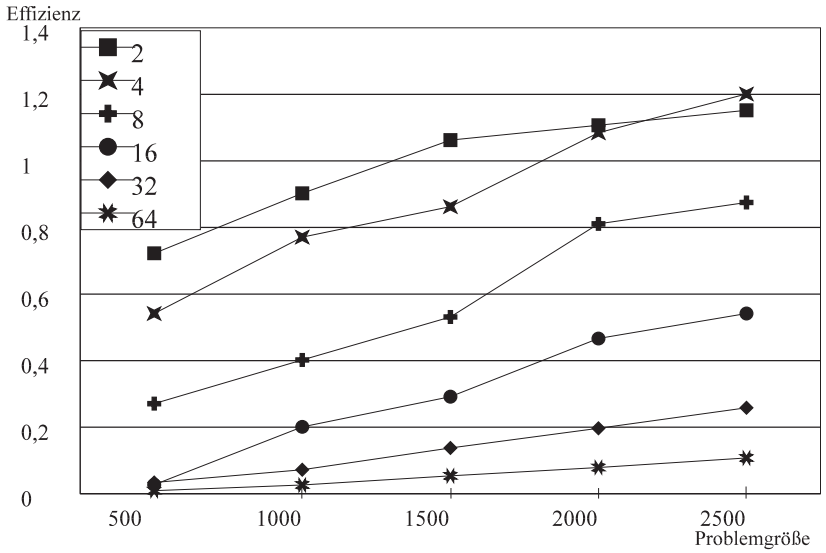

Abbildung 10: Effizienz für verschiedene Prozessorzahlen

3.4 Zusammenfassung

Es stellt sich an dieser Stelle die Frage, welche Knotenzahl optimal ist in Abhängigkeit von der Problemgröße. Dazu stehen zwei Kriterien zur Verfügung: die Effizienz und die Laufzeit. Die Ergebnisse in Abb. 10 zeigen, daß die Effizienz für größere Prozessorzahlen sehr schlecht ist. Nach dem Kriterium wäre also nur der Einsatz mit sehr kleinen Knotenzahlen sinnvoll.

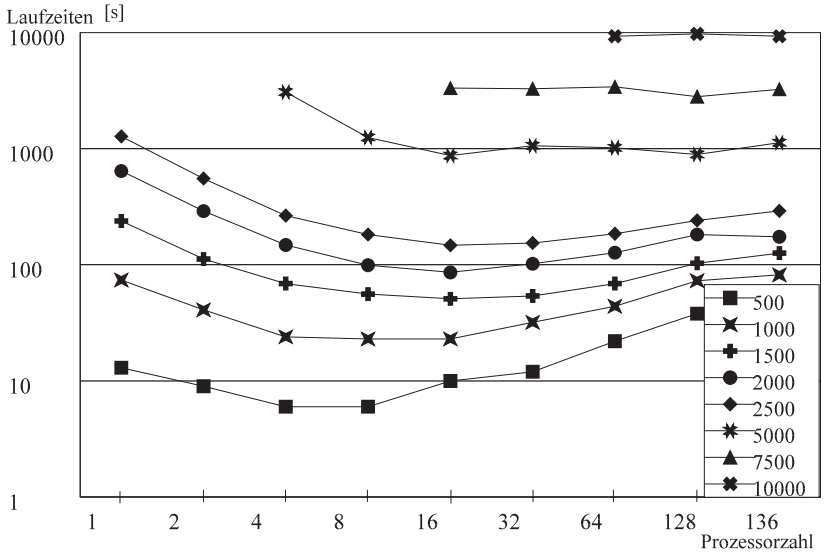

Abbildung 11: Laufzeiten für verschiedene Prozessorzahlen

In Abb. 11 sind die Abb. 7 und 8 sozusagen invertiert worden. Die Laufzeiten sind jetzt in Abhängigkeit von den Knotenzahlen dargestellt, jede Kurve beinhaltet die Ergebnisse für eine bestimmte Problemgröße (die Laufzeiten sind zur besseren Übersicht logarithmisch aufgetragen). Man kann aus dieser Abbildung die optimale Laufzeit in Abhängigkeit von der Problemgröße ablesen. Diese Werte sind in Tab. 1 aufgeführt. Zwischen den Problemgrößen 5000 und 7500 zeigt die Tabelle einen großen Sprung in der Knotenzahl. Abb. 11 relativiert dieses Ergebnis: Ab einer Problemgröße von 5000 unterscheiden sich die Laufzeiten für Knotenzahlen ab 16 kaum noch. Deswegen ist in Tab. 1 eine dritte Spalte hinzugefügt worden mit den empfohlenen Knotenzahlen für die Konfiguration des Parallelrechners, wie er im ZAM existiert.

Das wesentliche Problem, das bei den Untersuchungen mit dem ProSolver-DES aufgetreten ist, betrifft die relativ schlechten Ergebnisse bei Läufen mit großen Prozessorzahlen. Die Lösung dieses Problems liegt sehr wahrscheinlich in der Konfiguration des PFS. Um bei großen Prozessorzahlen gute Ergebnisse zu erzielen, müßten folgende Maßnahmen getroffen werden:

1. Erhöhung der Anzahl der I/O-Knoten, die das PFS steuern. Dabei scheint ein Verhältnis von Rechenknoten zu I/O-Knoten von 5:1 bis 6:1 ein Maximum zu sein. Die Anzahl der verfügbaren I/O-Knoten auf dem Paragon-Rechner im ZAM müßte also von 5 auf 20 oder 25 erhöht werden.

Problemgröße	**Knotenzahl mit kleinster Laufzeit**	**Empfohlene Knotenzahl**
500	4	4
1000	8	8
1500	16	16
2000	16	16
2500	16	16
5000	16	16
7500	128	16
10000	136	64

Tabelle 1: Optimale Knotenzahlen in Abhängigkeit von den Problemgrößen.

2. Installation des PFS nach dem Muster, wie es im Handbuch des ProSolver-DES beschrieben wird (vgl. Abschnitt 2.2). Obwohl diese Installation nicht gestestet werden konnte, ist zu erwarten, daß durch ihre Verwirklichung eine Verbesserung der Leistungen eintritt.

Literatur

[1] R. Berrendorf, H. Burg, U. Detert, *Leistungscharakteristika von Parallelrechnern: Fallstudie Intel Paragon*, it+ti (Informationstechnik und Technische Informatik) Themenheft Leistung von Parallelrechnern, No. 2, 1995, pp. 37 – 45

[2] R. Berrendorf, H. Burg, U. Detert, R. Esser, M. Gerndt, R. Knecht, *Intel Paragon XP/S — Architecture, Software Environment and Performance*, Interner Bericht KFA-ZAM-IB-9409, Forschungszentrum Jülich GmbH, Zentralinstitut f. Angew. Math., Jülich 1994

[3] J. Choi, J. Dongarra, R. Pozo and D. W. Walker, *ScaLAPACK : A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers*, Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia). IEEE Computer Society Press, Los Alamitos, California, 1992

[4] R.T. Gregory and D.L. Karney, *A collection of matrices for testing computational algorithms*, Robert E. Krieger, Huntington 1978

[5] Intel Supercomputers Systems Division, *Paragon Prosolver-DES Manual*, Beaverton 1994

[6] G. Zavagli, *Untersuchung des parallelen Gleichungslösers Paragon ProSolver-DES — slab solver — auf dem Rechner Paragon XP/S-10*, Studienarbeit an der RWTH Aachen, Lehrstuhl für Technische Informatik und Computerwissenschaften, Aachen 1995

A Testprogramm

Das folgende Programm wurde für die Messungen mit dem ProSolver-DES eingesetzt. Für das Gleichungssystem wurden die Testmatrizen von Gregory und Karney [4] verwendet (s. Anhang B).

```
      program deststst
c Test program for the Intel ProSolver DES
c size of A
      integer*4 nrows
      integer*4 ncols
      parameter nrows = 1000
      parameter ncols = nrows
c sizes of decomposition blocks
      integer*4 barow
      integer*4 bacol
      integer*4 bbrow
      integer*4 bbcol
c max. size of a decomposition block
      integer*4 maxbrow
      integer*4 maxbcol
      parameter maxbrow = 1000
      parameter maxbcol = maxbrow
c variables for DES_put
      integer*4 row_index
      integer*4 col_index
c size of block that shall be written
      integer*4 numrows
      integer*4 numcols
c pathnames
      character*26 pathname
      character*33 pathfactor
      character*32 pathsolve
      character*22 pathlog
      character*6  factor
      character*5  solve
c file where to write results
      character*10 ergdat
      logical      exs
c matrix ids
      integer*4 id_a
      integer*4 id_b
c buffer for DES_put and DES_get
      complex*16 buff(maxbrow*maxbcol)
c identifier for asynchronous message passing (not used by this program)
      integer*4 io_id
c processor identifier
```

```
      integer*4 me
      integer*4 mynode
      integer*4 np
      integer*4 numnodes
c loop variables
      integer*4 i
      integer*4 j
c error counter and local value
      integer*4  error
      complex*16 zloc
c solution vector
      complex*16 sol(nrows)
c variables for error measure
      real*8     maxerr
      real*8     averr
      integer*4  maxnode
      complex*16 dsol
      real*8     adsol
      integer*4  tskmsg
      integer*4  ptype
      integer*4  myptype
c variables for time measurements
      real*8 dclock
      real*8 tstart
      real*8 tstop
      real*8 tdiff
      real*8 loadtime
      real*8 factime
      real*8 soltime
      real*8 runtime
c initialize pathnames
      pathname   = "/pfs/zamex/zam068/solution"
      pathfactor = "/pfs/zamex/zam068/solution/factor"
      pathsolve  = "/pfs/zamex/zam068/solution/solve"
      pathlog    = "/home/zamex/zam068/tst"
      factor     = "factor"
      solve      = "solve"
      ergdat     = "deststst.erg"
      me     = mynode()
      np     = numnodes()
      tskmsg = 10
      ptype  = myptype()
c compute tdiff
      tstart = dclock()
      tstop  = dclock()
      tdiff  = tstop - tstart
c -----
```

```
c create DES-case
      call DES_create(pathname)
c set maximal size of decomposition block
      call DES_set_param("ncol", maxbcol)
      call DES_set_param("nrow", maxbrow)
c set solver to slabsolver
      call DES_set_param("solver", 2)
c open A and b
      call DES_mopen(nrows, factor, id_a, barow, bacol)
      call DES_vopen(id_a, 1, solve, id_b, bbrow, bbcol)
c -----
c write data to coefficient matrix
      call gsync()
      tstart = dclock()
c nodes write data one after another (see text)
      if (me.gt.0) call crecv(tskmsg, i, 0)
c every processor starts with column 1
      col_index = 1
c calculate row_index for each node
      row_index = me*barow + 1
c if rows are in padded area, skip operation
      if (row_index.gt.nrows) goto 11
c calculate numrows for each node
      if ((row_index+barow-1).le.nrows) then
         numrows = barow
      else
         numrows = nrows - row_index + 1
      endif
 10   continue
c calculate how many columns to write
         if ((col_index+bacol-1).le.ncols) then
            numcols = bacol
         else
            numcols = ncols - col_index + 1
         endif
c compute elements of block
         do 20 i = col_index, (col_index+numcols-1)
            do 30 j = row_index, (row_index+numrows-1)
               if (j.gt.i) then
                  zloc = cmplx(nrows-j+1,nrows-j+1)
               else
                  zloc = cmplx(ncols-i+1,ncols-i+1)
               endif
               buff(numrows*(i-col_index)+(j+1-row_index)) = zloc
 30         continue
 20      continue
c put block in DES-file
```

```
         call DES_put(id_a, numrows, numcols, row_index, col_index,
     >                buff, 16*numrows*numcols, io_id)
c increment col_index and quit if past end of matrix
         col_index = col_index + bacol
         if (col_index.gt.ncols) goto 11
         goto 10
 11   continue
c tell next node to write data
      if (me.lt.(np-1)) call csend(tskmsg, i, 0, me+1, ptype)
      call gsync()
c -----
c fill right hand side
      if (me.gt.0) call crecv(tskmsg, i, 0)
      col_index = 1
c compute row_index for each node
      row_index = me*bbrow + 1
c check if rows are in padded area
      if (row_index.gt.nrows) goto 41
c calculate numrows for each node (numcols = 1)
      if ((row_index+bbrow-1).le.nrows) then
         numrows = bbrow
      else
         numrows = nrows - row_index + 1
      endif
      do 40 j = row_index, (row_index+numrows-1)
         buff(j-row_index+1) = cmplx(j,j)
 40   continue
c put data to RHS
      call DES_put(id_b, numrows, 1, row_index, 1, buff,
     >             16*numrows, io_id)
 41   continue
c tell next node to write
      if (me.lt.(np-1)) call csend(tskmsg, i, 0, me+1, ptype)
      call gsync()
      tstop = dclock()
      loadtime = tstop - tstart - tdiff
c ---
c check data in coefficient matrix
c each node reads his data and reports to node 0
      error  = 0.d0
      maxerr = 0.d0
      call gsync()
      col_index = 1
c calculate row_index
      row_index = me*barow + 1
c if node in padded area, skip operation
      if (row_index.gt.nrows) goto 110
```

```
c calculate numrows for each node
      if ((row_index+barow-1).le.nrows) then
         numrows = barow
      else
         numrows = nrows - row_index + 1
      endif
 100  continue
c calculate how many columns to read
         if ((col_index+bacol-1).le.ncols) then
            numcols = bacol
         else
            numcols = ncols - col_index + 1
         endif
c get block from DES-file
         call DES_get(id_a, numrows, numcols, row_index, col_index,
     >               buff, 16*numrows*numcols, io_id)
c check elements of block
         do 200 i = col_index, (col_index+numcols-1)
            do 300 j = row_index, (row_index+numrows-1)
               if (j.gt.i) then
                  zloc = cmplx(nrows-j+1,nrows-j+1)
               else
                  zloc = cmplx(ncols-i+1,ncols-i+1)
               endif
               dsol = zloc - buff(numrows*(i-col_index)+(j+1-row_index))
               adsol = sqrt(dsol*conjg(dsol))
               if (adsol.gt.(0.d-0)) then
                  error = error + 1
                  if (adsol.gt.maxerr) maxerr = adsol
               endif
 300        continue
 200     continue
c increment col_index and quit if past end of matrix
         col_index = col_index + bacol
         if (col_index.gt.ncols) goto 110
         goto 100
 110  continue
      call gsync()
      if (error.gt.0) error = 1
c every node sends report to node 0.
      if (me.eq.0) then
         do i = 1, np - 1
            call crecv(tskmsg, j, 4)
            if (j.gt.0) error = 1
         enddo
c send report back to all nodes
         call csend(tskmsg, error, 4, -1, ptype)
```

```
      else
         call csend(tskmsg, error, 4, 0, ptype)
         call crecv(tskmsg, error, 4)
      endif
c if error = 1 then stop execution on all nodes
      if (error.gt.0) then
         call DES_close(id_a)
         call DES_close(id_b)
         call DES_delete(pathname)
         STOP
      endif
c ---
c start factorisation
      call gsync()
      tstart = dclock()
      call DES_factor(id_a)
      call gsync()
      tstop = dclock()
      factime = tstop - tstart - tdiff
c solve system of equations
      tstart = dclock()
      call DES_solve(id_a, id_b)
      call gsync()
      tstop = dclock()
      soltime = tstop - tstart - tdiff
      runtime = factime + soltime
c ---
c read solution vector and write it to buffer sol
      call gsync()
      col_index = 1
c compute row_index for each node
      row_index = me*bbrow + 1
c if rows are in padded area, skip operation
      if (row_index.gt.nrows) goto 4010
c calculate numrows for each node (numcols = 1)
      if ((row_index+bbrow-1).le.nrows) then
         numrows = bbrow
      else
         numrows = nrows - row_index + 1
      endif
c get data from RHS
      call DES_get(id_b, numrows, 1, row_index, 1, buff,
     >             16*numrows, io_id)
      do 4000 j = row_index, (row_index+numrows-1)
         sol(j) = buff(j-row_index+1)
 4000 continue
      if (me.eq.0) write(*,*) ' RHS read.'
```

```
 4010 continue
c ---
c check solution
c solution vector is:    -1      on first row
c                     nrows + 1  on last row
c                         0      on all other rows
      call gsync()
      maxerr = 0.d0
      averr  = 0.d0
      if (row_index.gt.nrows) goto 501
      do 500 j = row_index, (row_index+numrows-1)
c zloc is exact solution
         if (j.eq.1) then
            zloc = (-1.d0,0.d0)
         elseif (j.eq.nrows) then
            zloc = (dble(nrows+1),0.d0)
         else
            zloc = (0.d0,0.d0)
         endif
         dsol = zloc - sol(j)
         adsol = sqrt(dsol*conjg(dsol))
         averr = averr + adsol
         if (adsol.gt.maxerr) maxerr = adsol
 500  continue
      averr = averr/nrows
 501  continue
c send maxerr and averr to node 0
      if (me.eq.0) then
         maxnode = 0
         do i = 1, np - 1
            call crecv(tskmsg, buff, 2*16)
            if (real(buff(1)).gt.maxerr) then
               maxerr  = real(buff(1))
               maxnode = real(buff(2))
            endif
            averr = averr + aimag(buff(1))
         enddo
      else
         buff(1) = cmplx(maxerr, averr)
         buff(2) = cmplx(me, 0.d0)
         call csend(tskmsg, buff, 2*16, 0, ptype)
      endif
      call gsync()
c -----
c close A and b
      call DES_close(id_a)
      call DES_close(id_b)
```

```
c delete DES
      call DES_delete(pathname)
      call DES_delete(pathfactor)
      call DES_delete(pathsolve)
      call DES_delete(pathlog)
c -----
c write results to file
      if (me.eq.0) then
         inquire(file=ergdat, EXIST=exs)
         if (exs.eq..TRUE.) then
            open(1, file=ergdat, status='old')
 2000       continue
            read(1, '()', END=2001)
            goto 2000
 2001       continue
         else
            open(1, file=ergdat, status='new')
            write(1,*) '********** DES-RUN log file **********'
         endif
         write(1,*) ' '
         write(1,*) ' ***** DES-RUN *****'
         write(1,*) '  processors    :', np
         write(1,*) '  nrows, ncols  :', nrows, ncols
         write(1,*) '  loadtime      :', loadtime
         write(1,*) '  factor time   :', factime
         write(1,*) '  solve time    :', soltime
         write(1,*) '  runtime       :', runtime
         write(1,*) '  maximum error :', maxerr
         write(1,*) '  average error :', averr
         close(1)
      endif
      end
```

B Testmatrix nach Gregory und Karney

Eine Testmatrix zur Lösung von Gleichungssystemen ist Gregory und Karney [4] entnommen. A ist eine symmetrische $(n \times n)$ Matrix mit

$$a_{i,j} = a_{j,i} = n + 1 - i \quad f\ddot{u}r \quad i \geq j$$

A hat also folgende Gestalt:

$$\begin{pmatrix} n & n-1 & n-2 & \cdots & 2 & 1 \\ n-1 & n-1 & n-2 & \cdots & 2 & 1 \\ n-2 & n-2 & n-2 & \cdots & 2 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 2 & 2 & 2 & \cdots & 2 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{pmatrix}$$

Die Inverse dieser Matrix lautet:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \ddots & \vdots \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix}$$

Mit der rechten Seite

$$b = \begin{pmatrix} 1 \\ 2 \\ \vdots \\ n \end{pmatrix}$$

ist die exakte Lösung x^* des Gleichungssystems

$$Ax = b$$

$$x^* = A^{-1}b = \begin{pmatrix} -1 \\ 0 \\ \vdots \\ 0 \\ n+1 \end{pmatrix}$$

.

Die Genauigkeit der vom ProSolver-DES errechneten Lösung kann also untersucht werden, indem sie mit der exakten Lösung verglichen wird.