TECHNISCHE UNIVERSITÄT DRESDEN

ZENTRUM FÜR INFORMATIONSDIENSTE
UND HOCHLEISTUNGSRECHNEN
PROF. DR. WOLFGANG E. NAGEL

Komplexpraktikum “Paralleles Rechnen”

Algorithmus nach Gauß

Friedrich Zahn
(Mat.-Nr.: 3966130)

Hochschullehrer: Prof. Dr. Wolfgang E. Nagel
Betreuer: Dr.-Ing. Robert Schöne

Dresden, 14. März 2021

Inhaltsverzeichnis

1 Aufgabenstellung

Zur Lösung von linearen Gleichungssystemen ist der Algorithmus nach Gauß in den Programmiersprachen C und Fortran90 zu implementieren!

- Implementieren Sie den Algorithmus in seiner klassischen Form, wobei sowohl bei der Neuberechnung der Matrixelemente A als auch bei der Rücksubstitution der Datenzugriff über die Zeile erfolgt! Messen Sie die Laufzeiten in C und in Fortran90 und diskutieren Sie die Ergebnisse!
- Stellen Sie den Algorithmus so um, dass sowohl bei der Neuberechnung der Matrixelemente A als auch bei der Rücksubstitution der Datenzugriff über die Spalte erfolgt! Implementieren Sie diesen Algorithmus für beide Sprachen, messen Sie die Laufzeit, und diskutieren Sie die Ergebnisse!
- Parallelisieren Sie beide Varianten aus 5.1 des Algorithmus mit OpenMP, ermitteln Sie den Speedup und die parallele Effizienz für $p = 2, 4, 8$ und diskutieren Sie die Ergebnisse!

2 Benchmark-Umgebung

2.1 Software

Die von mir entwickelten Programme wurden auf dem Taurus-Cluster mit den dort vorhandenen Tools kompiliert. Zum Einsatz kamen GCC und gfortran in Version 7.3.0 mit OpenMPI 3.1.1 und OpenMP 4.5. Die build-Umgebung wurde mit cmake 3.11 generiert.

2.2 Hardware

Alle Messungen wurden auf dem HPC-Cluster „Taurus“ des ZIH durchgeführt. Um die Vergleichbarkeit von Messungen sicherzustellen, wurde für alle Benchmarks die „haswell“-Partition genutzt, in welcher alle Nodes über 2 Intel Xeon CPUs, Modell E5-2680 v3, mit jeweils 12 Kernen verfügen[1]. Untereinander sind die Nodes per Infiniband verbunden. Hyperthreading ist auf diesen Nodes deaktiviert, darüber hinaus wurde mittels SLURM die exklusive Nutzung der Nodes angefordert, um Verzerrungen durch anderweitig mitgenutzte Nodes auszuschließen.

3 Lösungsansatz

Konzeptionell ist das Lösen eines linearen Gleichungssystems mithilfe des Algorithmus nach Gauß in zwei wesentliche Schritte zu zerlegen. Zunächst wird das lineare Gleichungssystem $A \cdot x = b$ mittels des eigentlichen Gaußschen Eliminationsverfahren in ein äquivalentes System $R \cdot x = y$ umgeformt[2], wobei R die Form einer rechten Dreiecksmatrix annimmt - alle Einträge links der Diagonalen sind Null. Die erlaubten, da für die Lösung des Systems unschädlichen, Umformungen auf der erweiterten Koeffizientenmatrix $(A|b)$ sind dabei:

- Zeilen miteinander zu vertauschen,
- Alle Einträge in einer Zeile mit einem Faktor ungleich Null zu multiplizieren,
- Ein Vielfaches einer Zeile zu einer anderen zu addieren.

Zur Frage der Existenz einer solchen Darstellung sowie der Äquivalenz der Lösungen, bietet sich die Betrachtung des Problems als LR-Zerlegung an. Es kann gezeigt werden, dass für eine Permutationsmatrix P (Zeilenvertauschungen), und eine linke Dreiecksmatrix (Umformungen), stets folgende Zerlegung existiert:

$$P \cdot A = L \cdot R$$

Damit folgt $PAx = Pb \Leftrightarrow LRx = Pb$ und mit den Bezeichnungen $y := Rx, \hat{b} := Pb$ kann nun zunächst $Ly = \hat{b}$ und schließlich $Rx = y$ gelöst werden. Das Gaußsche Eliminationsverfahren, wie es hier angewandt wird, berechnet dabei L nicht explizit, dies wäre nur von Vorteil wenn die selbe Koeffizientenmatrix für verschiedene rechte Seiten b gelöst werden müsste.
Nachdem das äquivalente System $Rx = y$ gefunden ist, kann es in einem zweiten Schritt mittels Rücksubstitution gelöst werden, welche der weiteren Anwendung der obigen Regeln entspricht, bis die Koeffizientenmatrix die Einheitsmatrix ist - also jede Unbekannte nur noch in einer Gleichung existiert, darin den Koeffizienten Eins hat, und auf der rechten Seite ihr Wert steht.

4 Implementation

Die zur Lösung der Aufgabe geschriebenen Programme erzeugen zunächst aus Zufallszahlen ein dicht besetztes Gleichungssystem, welches als Array dargestellt wird. Dabei müssen mindestens so viele Gleichungen (Zeilen) wie Unbekannte (Spalten weniger 1) angefordert werden.
Für den Erhalt der Lösung wird nur die Dreiecksmatrix R benötigt, welche unabhängig von der anderen berechnet werden kann. Der hier umgesetzte Algorithmus wurde [3] entlehnt, er berechnet die rechte, obere Dreiecksmatrix R speichersparend in-place unter der Nutzung eines maximalen Pivotelements. Damit kann der Algorithmus korrekt mit Null-Einträgen umgehen und ist hinsichtlich von Fließkommaoperationen numerisch stabiler.
Konkret ermittelt der Algorithmus beginnend „oben links“ zunächst die verbleibende Gleichung mit dem betragsmäßig größten Koeffizienten für die betrachtete Unbekannte, und wählt dieses als Pivotelement. Sollte es Null sein, ist für diese Unbekannte nichts weiter zu tun, alle verbleibenden Koeffizienten sind bereits Null und entsprechen damit der gesuchten Dreiecksform. Andernfalls tauscht es die aktuell betrachtete Gleichung mit der, in dem das Pivotelement vorkommt. In allen „unterhalb“ verbleibenden Gleichungen wird nun die betrachtete Unbekannte eliminiert, indem die Pivot-Gleichung mit dem der Eliminierung entspringenden Faktor multipliziert addiert wird. Anschließend beginnt der gleiche Ablauf von vorn, für die nächste Gleichung und die nächste Unbekannte, bis schließlich von mindestens einem der beiden keine weiteren vorhanden sind.
Die anschließende Rücksubstitution arbeitet ebenfalls in-place, und beginnt dabei von „unten rechts“. Waren genügend unabhängige Gleichungen gegeben, enthält diese Gleichung höchstens einen Koeffizienten ungleich Null, und lässt sich entsprechend durch simple Division direkt lösen - ist der Koeffizient

gerade Null, enthält die Gleichung keine Information und muss ausgelassen werden. Die andernfalls damit bestimmte Unbekannte wird nun in allen verbleibenden Gleichungen „darüber“ eingesetzt und entsprechend mit den „rechten“ Seiten der Gleichungen verrechnet. Damit enthält die nächste Gleichung aufgrund der Dreiecksform wieder nur noch einen Koeffizienten ungleich Null, und der Ablauf kann fortgesetzt werden, bis die nötige Zahl an Gleichungen ausgewertet und damit alle Unbekannten bestimmt sind.
Beide Algorithmen müssen mit Fließkommazahlen arbeiten, dabei jedoch zuverlässig Null-Einträge erkennen können. Um diese auch angesichts der Darstellungsunsicherheiten bei Fließkomma-Arithmetik korrekt handhaben zu können, wird ein Betrags-Vergleich mit einem im Quellcode zu setzenden Epsilon anstelle eines direkten Vergleichs auf Null eingesetzt.
Die Zeitmessung erfasst ausschließlich die Rechenzeit für die eigentlichen Lösungsalgorithmen, nicht für die Generierung des Systems oder die Prüfung der Lösung. Im C-Code erfolgt dies über die Routine `clock_gettime(CLOCK_MONOTONIC, ...)` und unter Fortran über `cpu_time(...)` für single-threaded Programme und über `system_clock(...)` für das multi-threaded, parallelisierte Programm.
Der Programmcode für den klassischen Algorithmus in der Speicherdarstellung nach Zeilen, entsprechend Aufgabe 5.1, findet sich für C in Listing 1 und für Fortran in Listing 2.
Für den Speicherzugriff nach Spalten, Aufgabe 5.2, wurden im Wesentlichen konsequent die Indizes für alle Array-Allokation und -Zugriffe vertauscht. Das Vertauschen von Gleichungen musste in C mittels einer Schleife implementiert werden, da in dieser Darstellungsart Gleichungen nicht mehr einen einzigen kontinuierlichen Speicherbereich belegen und entsprechend nicht als solcher kopiert werden können. Unter Fortran waren derartige Anpassungen nicht nötig, da die Sprache über eine dimensionsunabhängige Slicing-Notation verfügt. Die Listings finden sich unter 3 und 4.
Die mittels OpenMP parallelisierten Varianten finden sich in den Listings 5 und 6. Sowohl im eigentlichen Gauß-Algorithmus wie auch bei der Rücksubstitution bieten sich wesentliche Teile für die leicht umzusetzende OpenMP-Direktive `parallel for` bzw. das Fortran-Äquivalent `parallel ... do` an. Beim Gauß-Algorithmus kann die Elimination der Unbekannten in allen Gleichungen „unterhalb“ des Pivot-Elements problemlos parallelisiert werden, da diese Operation stehts nur Abhängigkeiten innerhalb einer Gleichungen hat. Auch die Suche nach dem Pivot-Element ließe sich parallelisieren, jedoch war hierfür kein signifikanter Speedup zu beobachten, möglicherweise weil der OpenMP-Overhead gegenüber den Performance-Gewinnen bei dieser ohnehin Speicherzugriffs-limitierten Routine überwiegt. Im Rahmen der Rücksubstitution wird das Eliminieren der gerade bestimmten Unbekannten in den verbleibenden Gleichungen parallelisiert. Da die Arbeitslast pro Iteration bei beiden Algorithmen innerhalb der jeweiligen parallelisierten Blöcke konstant ist, bieten dynamische Scheduling-Ansätze zur Verteilung der Iterationen hier im Übrigen keinen Vorteil, sondern vergrößern im Gegenteil eher den Overhead. Bei beiden Teilalgorithmen verbleibt ein - zum Teil inhärent - serieller Codeabschnitt, zum Einen das Finden und Anordnen des Pivotelements, zum Anderen die Bestimmung der Unbekannten, sowie allgemein die Auswertung der benötigten Iterations-Logik.

4.1 Bedienungshinweise

Alle Programme erwarten vom Nutzenden drei Argumente: An erster Stelle die Zahl der Gleichungen („Zeilen“), an zweiter Stelle die Zahl der Unbekannten plus Eins („Spalten“ in der üblichen Matrixdarstellung), und zuletzt ob der Debug-Modus aktiviert (1) oder deaktiviert (0) werden soll. Der Debug-Modus gibt die erzeugten Systeme und Zwischen-Lösungen aus, und gleicht außerdem die gefundene Lösung mit der zu Beginn generierten Unbekannten-Belegung ab. Dabei ist zu beachten, dass diese Überprüfung nicht mit unterbestimmten Gleichungssystemen umgehen kann, deren Lösung nicht eindeutig ist. Diese können bei der zufälligen Generierung der Systeme jedoch auftreten.
Abschließend geben die Programme die Laufzeit des Kern-Algorithmus für die einkompilierte Anzahl von Systemen und die geforderte Zahl von Freiheitsgraden (Unbekannten) aus.
Eine Messung auf einem SLURM-Cluster kann also beispielsweise folgendermaßen vorgenommen

werden:

```
$ srun --nodes 1 --ntasks-per-node=1 --cpus-per-task=12 --exclusive --partition=
    haswell ./kppr_5_3 1500 1501 0
```

Für die parallelisierten Programme kann die Zahl der tatsächlich genutzten Kerne über die Umgebungsvariable `OMP_NUM_THREADS` gesteuert werden, für die Messungen hier wurde dazu eine interaktive SLURM-Allokation entsprechend obiger Konfiguration genutzt.

5 Ergebnisse

5.1 Klassischer Algorithmus

Freiheitsgrade	T_{C} [s]	T_{Fortran} [s]
1000	23	378
1500	78	1889
2000	206	5525

Tabelle 1: Auf ganze Sekunden gerundete Laufzeiten T des implementierten Algorithmus für 100 Lösungen zufällig generierter, dicht besetzter linearer Gleichungssysteme mit der angegebenen Anzahl an Freiheitsgraden n, wenn die Gleichungen in einer repräsentativen $n \times (n+1)$ Matrix in Zeilen abgelegt und ausgelesen werden.

Die Ergebnisse der Messungen für den klassischen Algorithmus mit Datenzugriff über die Zeilen sind in Tabelle 1 aufgeführt. Es fällt sofort auf, dass die Laufzeiten der C-Version eine Größenordnung unter der der Fortran-Version liegen. Beide Sprachen arbeiten Hardware-nah und werden vielfach in etablierten Numerik-Frameworks eingesetzt, es ist nicht unüblich, dass eine Bibliothek sowohl C- als auch Fortran-Schnittstellen anbietet. Insofern liegt es fern, dass Fortran eine inhärent weniger performante Sprache zur Lösung derartiger Probleme wäre.
Setzt man die gemessenen Laufzeiten ins Verhältnis zu Potenzen der Anzahl der Freiheitsgrade n, zeigt sich, dass die C-Implementierung grob einem $O(n^3)$ Verhalten folgt, während die Laufzeit unter Fortran eher mit $O(n^4)$ geht. Ein Verhalten von $O(n^3)$ passt gut ins Bild, das Laufzeitverhalten klassischer Lösungsalgorithmen für lineare Gleichungssysteme korreliert stets mit dem der Matrixmultiplikation. Eine naive Implementation dieser zeigt ebenfalls ein Verhalten von $O(n^3)$, und die Kombination von Gauß-Umformungen und Rücksubstitution stellt auch einen verhältnismäßig „naiven“ Ansatz für die Lösung von Gleichungssystemen dar.

5.2 Datenzugriff nach Spalten

Freiheitsgrade	T_{C} [s]	T_{Fortran} [s]
1000	376	23
1500	1887	76
2000	5591	201

Tabelle 2: Auf ganze Sekunden gerundete Laufzeiten T des implementierten Algorithmus für 100 Lösungen zufällig generierter, dicht besetzter linearer Gleichungssysteme mit der angegebenen Anzahl an Freiheitsgraden n, wenn die Gleichungen in einer repräsentativen $(n+1) \times n$ Matrix in Spalten abgelegt und ausgelesen werden.

Die Messungen für die Programme, die ihre Gleichungen in Array-Spalten ablegen, sind in Tabelle 2 aufgelistet. Die Ergebnisse entsprechen den Zahlen nach denen bei der vorhergehenden Messung - jedoch haben C und Fortran ihr Verhalten praktisch getauscht. Dies liefert auch direkt ein starkes Indiz, worauf der erhebliche Laufzeitunterschied beruht. Während unter C Arrays Zeile für Zeile im Speicher abgelegt wird, werden unter Fortran Arrays Spalte für Spalte im Speicher abgelegt[4]. Da die implementierten Algorithmen in erster Linie auf komplette, einzelne Gleichungen zugreifen müssen, ist es sehr viel effizienter diese auch in zusammenhängenden Speicherbereichen abzulegen, um die in Haswell-Prozessoren vorhandenen Caching-Mechanismen zu nutzen. Auch in meiner eigenen Implementierung konnte ich im C-Code das vorteilhafte Zeilen-orientierte Speicherlayout ausnutzen, indem eine ganze Gleichung in einem einzigen memcpy-Befehl kopiert wird - während im Spalten-orientierten Layout eine zusätzliche Schleife dafür eingesetzt werden muss. Die einzige Subroutine, die von ununterbrochenen Spalten im Speicher profitieren

dürfte, ist die Suche nach dem Pivot-Element, diese nimmt jedoch nur einen unwesentlichen Teil der Gesamtlaufzeit ein.
In Summe können diese Ergebnisse als Lehrbuch-Beispiel für die Wichtigkeit eines durchdachten, auf den Anwendungsfall abgestimmten Speicherlayouts betrachtet werden. Wie sich im Weiteren zeigen wird, übertreffen die Performance-Gewinne durch das vorteilhafte Speicherlayout bei Weitem jene, die sich durch ein Aufwenden eines Vielfachen der Rechen-Ressourcen erreichen lassen.

5.3 Parallelisierung

p	T_{C} [s]	S_{C}	T_{Fortran} [s]	S_{Fortran}
1	77	1.00	1887	1.00
2	49	1.57	1071	1.76
4	27	2.86	575	3.28
8	16	4.96	301	6.27

Tabelle 3: Auf ganze Sekunden gerundete Laufzeiten T und Speedups des teilweise parallelisierten Algorithmus auf p Prozessoren für 100 Lösungen zufällig generierter, dicht besetzter linearer Gleichungssysteme mit 1500 Freiheitsgraden und 1500 Gleichungen, bei Speicherung nach Zeilen.

Die Ergebnisse für die parallelisierten Varianten der Programme sind in Tabelle 3 aufgeführt. Neben den reinen Laufzeiten sind dort auch die entsprechenden Speedups angegeben, die für die Betrachtung der Parallelisierungs-Effizienz deutlich aussagekräftiger und einfacher zu lesen sind. Der Speedup S für eine serielle Laufzeit T_s gegenüber einer parallelisierten Laufzeit T_p wird dabei folgendermaßen berechent:

$$S = \frac{T_s}{T_p}$$

Zusätzlich sind die Speedup-Zahlen noch einmal in Abbildung 1 graphisch aufbereitet.
Es zeigt sich, dass Overhead und serielle Anteile einen nicht unerheblichen Einfluss haben, die Speedups bleiben deutlich unter der theoretischen Ideallinie von $S = p$ zurück. Nichtsdestotrotz wird ein angesichts der nicht nach dem Kriterium der Parallelisierbarkeit ausgewählten Algorithmen ein zufriedenstellender Speedup erreicht, der sich auch auf eine mittlere, heute alltagsübliche Zahl an Prozessoren skalieren lässt.
Es fällt auf, dass der Fortran-Code einen höheren Speedup zeigt, als der C-Code. Dies relativiert sich jedoch angesichts der absoluten Laufzeiten, das Fortran-Programm kann lediglich einen Bruchteil der Auswirkungen des nachteiligen Speicher-Layouts wett machen. Da die besonders nachteiligen Speicherzugriffe im parallelisierten Code-Abschnitt liegen, dürften diese Abschnitte im seriellen Programm einen entsprechend größeren Laufzeit-Anteil ausmachen was schließlich zu einem rechnerisch höheren Speedup führt. Es kann daher aber nicht darauf geschlossen werden, dass sich Fortran vorteilhafter parallelisieren ließe.

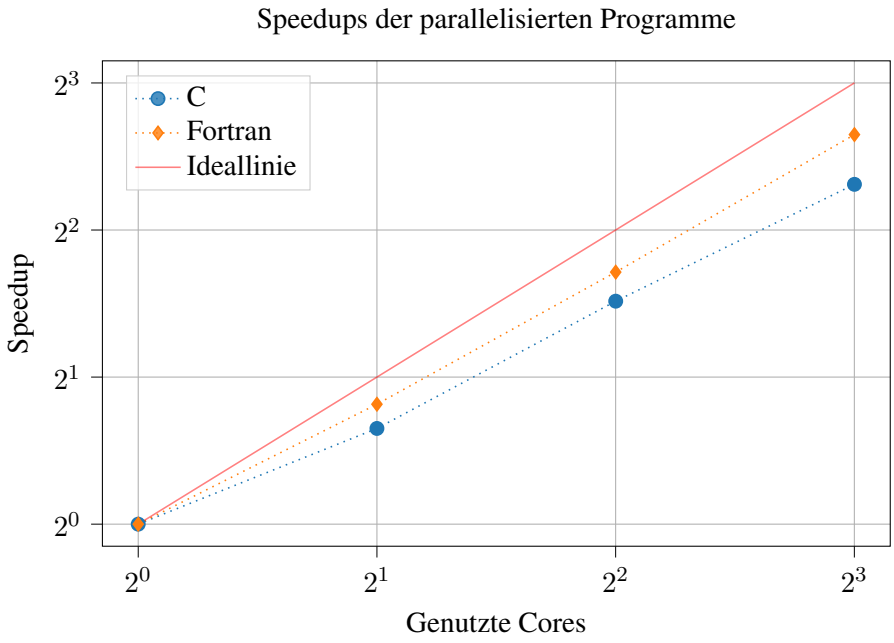

Abbildung 1: Speedups für die parallelisierten Programme in Abhängigkeit von der Anzahl der eingesetzten Prozessoren in einer Log-Log-Darstellung zur Basis 2. Die rote Ideallinie stellt die theoretischen Speedups für eine perfekt parallelisierbare Arbeitslast ohne Overhead oder serielle Anteile dar. Alle drei Kurven entspringen dabei per Definition dem Punkt $p = 1, S = 1$.

Literatur

[1] *Hardware-Informationen zum Taurus-Cluster im ZIH-Kompendium*. https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HardwareTaurus, . – [Online; Stand 18. Januar 2021]

[2] WIKIPEDIA: *Gaußsches Eliminationsverfahren — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=Gau%C3%9Fsches_Eliminationsverfahren&oldid=208565391, 2021. – [Online; Stand 14. März 2021]

[3] WIKIPEDIA CONTRIBUTORS: *Gaussian elimination — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Gaussian_elimination&oldid=1010087474, 2021. – [Online; Stand 4. März 2021]

[4] WIKIPEDIA CONTRIBUTORS: *Row- and column-major order — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Row-_and_column-major_order&oldid=1006237871, 2021. – [Online; Stand 13. März 2021]

A Programmcode

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <string.h>

#define EPS 0.000001
#define REPS 100

int main(int argc, char** argv) {
    int M = atoi(argv[1]);
    int N = atoi(argv[2]);
    int debug = atoi(argv[3]);

    double (*A)[N] = malloc(sizeof(double[M][N]));
    double *sol = malloc((N-1)*sizeof(double));
    double *tmp = malloc(N * sizeof(double));
    double *sol_back = malloc((N - 1) * sizeof(double));

    srandom(time(NULL));
    struct timespec tStart, tEnd;
    double timeTaken = 0;

    for (int r = 0; r < REPS; r++) {
        if (debug) printf("The generated solution vector is:\n");
        for (int n = 0; n < N - 1; n++) {
            sol[n] = (random() % 20) - 10;
            if (debug) printf("%.1f\t", sol[n]);
        }

        for (int m = 0; m < M; m++) {
            A[m][N - 1] = 0;
            for (int n = 0; n < N - 1; n++) {
                A[m][n] = (random() % 20) - 10;
                A[m][N - 1] += A[m][n] * sol[n];
            }
        }

        if (debug) {
            printf("The generated linear system is:\n");
            for (int m = 0; m < M; m++) {
                for (int n = 0; n < N; n++) {
                    printf("%.2f\t", A[m][n]);
                }
                printf("\n");
            }
            printf("\n\n");
        }

        clock_gettime(CLOCK_MONOTONIC, &tStart);
        int h = 0, k = 0;

        while (h < M && k < N) {
            int i_max = h;
            for (int i = h + 1; i < M; i++) {
                if (fabs(A[i][k]) > fabs(A[i_max][k])) {
                    i_max = i;
                }
            }

```

```
61              if (fabs(A[i_max][k]) < EPS) {
62                  k++;
63              } else {
64                  //swap rows h and i_max
65                  memcpy(tmp, &A[h][0], N * sizeof(double));
66                  memcpy(&A[h][0], &A[i_max][0], N * sizeof(double));
67                  memcpy(&A[i_max][0], tmp, N * sizeof(double));
68
69                  for (int i = h + 1; i < M; i++) {
70                      double f = A[i][k] / A[h][k];
71                      A[i][k] = 0;
72                      for (int j = k + 1; j < N; j++) {
73                          A[i][j] = A[i][j] - A[h][j] * f;
74                      }
75                  }
76                  h++;
77                  k++;
78              }
79          }
80
81          if (debug) {
82              printf("The computed echolon form is:\n");
83              for (int m = 0; m < M; m++) {
84                  for (int n = 0; n < N; n++) {
85                      printf("%.2f\t", A[m][n]);
86                  }
87                  printf("\n");
88              }
89              printf("\n\n");
90          }
91
92          int n = N - 2;
93          int m = M - 1;
94          while (n >= 0 && m >= 0) {
95              if (fabs(A[m][n]) < EPS) {
96                  m--;
97              } else {
98                  sol_back[n] = A[m][N - 1] / A[m][n];
99                  for (int i = 0; i < m; i++) {
100                     A[i][N - 1] -= A[i][n] * sol_back[n];
101                 }
102                 m--;
103                 n--;
104             }
105         }
106
107         clock_gettime(CLOCK_MONOTONIC, &tEnd);
108         timeTaken += tEnd.tv_sec - tStart.tv_sec + (tEnd.tv_nsec - tStart.tv_nsec) *
    1e-9; // in seconds
109
110         if (debug) {
111             printf("The computed solution vector is:\n");
112             for (int n = 0; n < N - 1; n++) {
113                 printf("%.1f\t", sol_back[n]);
114             }
115             printf("\n\n");
116
117             for (int n = 0; n < N - 1; n++) {
118                 if (fabs(sol[n] - sol_back[n]) > EPS) {
119                     printf("Solution was either wrong or not unique!\n");
120                     return EXIT_FAILURE;
121                 }
122             }
```

```c
        }
    }
    printf("%d solutions for %d dof took %f seconds.\n", REPS, N-1, timeTaken);
    return EXIT_SUCCESS;
}
```

Listing 1: Der verwendete Programmcode in C zur Aufgabe 5.1.

```
program kppr_5_1
    implicit none
    integer :: M, N, h, i, j, k, debug, r, REPS
    integer :: i_max
    integer :: seed
    doubleprecision :: f, EPS
    doubleprecision, dimension (:,:), allocatable :: A
    doubleprecision, dimension (:), allocatable :: sol, sol_back
    doubleprecision, dimension (:), allocatable :: tmp
    real :: tStart, tEnd, timeTaken

    character(len=32) :: arg
    call get_command_argument(1, arg)
    read (arg,'(I32)') M
    call get_command_argument(2, arg)
    read (arg,'(I32)') N
    call get_command_argument(3, arg)
    read (arg,'(I32)') debug

    allocate (A(M,N))
    allocate (sol(N-1))
    allocate (sol_back(N-1))
    allocate (tmp(N))
    EPS = 0.000001
    REPS = 100
    timeTaken = 0

    call system_clock(seed)
    call srand(seed)

    do r = 1, REPS
        do k = 1, N-1
            sol(k) = modulo(irand(), 20) - 10
        end do

        if (debug == 1) then
            print *, "The generated solution vector is:"
            print *, sol
        end if

        do h = 1, M
            A(h,N) = 0
            do k = 1, N-1
                A(h,k) = modulo(irand(), 20) - 10
                A(h,N) = A(h,N) + A(h,k) * sol(k)
            end do
        end do

        if (debug == 1) then
            print *, "The generated linear system is:"
            do i = 1, M
                print *, A(i,:)
            end do
        end if

```

```
56          call cpu_time(tStart)
57          h = 1
58          k = 1
59          do while (h <= M .and. k <= N)
60              i_max = h
61              do i = h+1, M
62                  if (abs(A(i,k)) > abs(A(i_max,k))) then
63                      i_max = i
64                  end if
65              end do
66
67              if (abs(A(i_max,k)) < EPS) then
68                  k = k + 1
69              else
70                  tmp = A(h,:)
71                  A(h,:) = A(i_max,:)
72                  A(i_max,:) = tmp
73
74                  do i = h+1, M
75                      f = A(i,k) / A(h,k)
76                      A(i,k) = 0
77                      do j=k+1, N
78                          A(i,j) = A(i,j) - A(h,j) * f
79                      end do
80                  end do
81                  h = h+1
82                  k = k+1
83              end if
84          end do
85
86          if (debug == 1) then
87              print *, "The linear system in echolon form is:"
88              do i = 1, M
89                  print *, A(i,:)
90              end do
91          end if
92
93          k = N - 1
94          h = M
95          do while (k > 0 .and. h > 0)
96              if (abs(A(h,k)) < EPS) then
97                  h = h - 1
98              else
99                  sol_back(k) = A(h,N)  / A(h,k)
100                 do i=1, h-1
101                     A(i,N) = A(i,N) - A(i,k) * sol_back(k)
102                 end do
103                 h = h - 1
104                 k = k - 1
105             end if
106         end do
107
108         call cpu_time(tEnd)
109         timeTaken = timeTaken + (tEnd - tStart)
110
111         if (debug == 1) then
112             print *, "The computed solution is:"
113             print *, sol_back
114
115             do i=1, N-1
116                 if (abs(sol(i) - sol_back(i)) > EPS) then
117                     call exit(1)
118                 end if
```

```
            end do
        end if
    end do

    write (*, *) REPS, "solutions for ", N-1, " dof took ", timeTaken, " seconds."
    call exit(0)

end program kppr_5_1
```

Listing 2: Der verwendete Programmcode in Fortran zur Aufgabe 5.1.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <string.h>

#define EPS 0.000001
#define REPS 100

int main(int argc, char** argv) {
    int M = atoi(argv[1]);
    int N = atoi(argv[2]);
    int debug = atoi(argv[3]);

    double (*A)[M] = malloc(sizeof(double[N][M]));
    double *sol = malloc((N-1)*sizeof(double));
    double tmp;
    double *sol_back = malloc((N-1)*sizeof(double));

    srandom(time(NULL));
    struct timespec tStart, tEnd;
    double timeTaken = 0;

    for (int r = 0; r < REPS; r++) {
        if (debug) printf("The generated solution vector is:\n");
        for (int n = 0; n < N - 1; n++) {
            sol[n] = (random() % 20) - 10;
            if (debug) printf("%.1f\t", sol[n]);
        }

        for (int m = 0; m < M; m++) {
            A[N - 1][m] = 0;
            for (int n = 0; n < N - 1; n++) {
                A[n][m] = (random() % 20) - 10;
                A[N - 1][m] += A[n][m] * sol[n];
            }
        }

        if (debug) {
            printf("The generated linear system is:\n");
            for (int m = 0; m < M; m++) {
                for (int n = 0; n < N; n++) {
                    printf("%.2f\t", A[n][m]);
                }
                printf("\n");
            }
            printf("\n\n");
        }

        clock_gettime(CLOCK_MONOTONIC, &tStart);
        int h = 0, k = 0;

```

```
53          while (h < M && k < N) {
54              int i_max = h;
55              for (int i = h + 1; i < M; i++) {
56                  if (fabs(A[k][i]) > fabs(A[k][i_max])) {
57                      i_max = i;
58                  }
59              }
60
61              if (fabs(A[k][i_max]) < EPS) {
62                  k++;
63              } else {
64                  //swap rows h and i_max
65                  for (int j = 0; j < N; j++) {
66                      tmp = A[j][h];
67                      A[j][h] = A[j][i_max];
68                      A[j][i_max] = tmp;
69                  }
70
71                  for (int i = h + 1; i < M; i++) {
72                      double f = A[k][i] / A[k][h];
73                      A[k][i] = 0;
74                      for (int j = k + 1; j < N; j++) {
75                          A[j][i] = A[j][i] - A[j][h] * f;
76                      }
77                  }
78                  h++;
79                  k++;
80              }
81          }
82
83          if (debug) {
84              printf("The computed echolon form is:\n");
85              for (int m = 0; m < M; m++) {
86                  for (int n = 0; n < N; n++) {
87                      printf("%.2f\t", A[n][m]);
88                  }
89                  printf("\n");
90              }
91              printf("\n\n");
92          }
93
94          int n = N - 2;
95          int m = M - 1;
96          while (n >= 0 && m >= 0) {
97              if (fabs(A[n][m]) < EPS) {
98                  m--;
99              } else {
100                 sol_back[n] = A[N - 1][m] / A[n][m];
101                 for (int i = 0; i < m; i++) {
102                     A[N - 1][i] -= A[n][i] * sol_back[n];
103                 }
104                 m--;
105                 n--;
106             }
107         }
108
109         clock_gettime(CLOCK_MONOTONIC, &tEnd);
110         timeTaken += tEnd.tv_sec - tStart.tv_sec + (tEnd.tv_nsec - tStart.tv_nsec) *
    1e-9; // in seconds
111
112         if (debug) {
113             printf("The computed solution vector is:\n");
114             for (int n = 0; n < N - 1; n++) {
```

```c
                printf("%.1f\t", sol_back[n]);
            }
            printf("\n\n");

            for (int n = 0; n < N - 1; n++) {
                if (fabs(sol[n] - sol_back[n]) > EPS) {
                    printf("Solution was either wrong or not unique!\n");
                    return EXIT_FAILURE;
                }
            }
        }
    }
    printf("%d solutions for %d dof took %f seconds.\n", REPS, N-1, timeTaken);
    return EXIT_SUCCESS;
}
```

Listing 3: Der verwendete Programmcode in C zur Aufgabe 5.2.

```fortran
program kppr_5_2
    implicit none
    integer :: M, N, h, i, j, k, debug, r, REPS
    integer :: i_max
    integer :: seed
    doubleprecision :: f, EPS
    doubleprecision, dimension (:,:), allocatable :: A
    doubleprecision, dimension (:), allocatable :: sol, sol_back
    doubleprecision, dimension (:), allocatable :: tmp
    real :: tStart, tEnd, timeTaken

    character(len=32) :: arg
    call get_command_argument(1, arg)
    read (arg,'(I32)') M
    call get_command_argument(2, arg)
    read (arg,'(I32)') N
    call get_command_argument(3, arg)
    read (arg,'(I32)') debug

    allocate (A(N,M))
    allocate (sol(N-1))
    allocate (sol_back(N-1))
    allocate (tmp(N))
    EPS = 0.000001
    REPS = 100
    timeTaken = 0

    call system_clock(seed)
    call srand(seed)

    do r = 1, REPS
        do k = 1, N-1
            sol(k) = modulo(irand(), 20) - 10
        end do

        if (debug == 1) then
            print *, "The generated solution vector is:"
            print *, sol
        end if

        do h = 1, M
            A(N,h) = 0
            do k = 1, N-1
                A(k,h) = modulo(irand(), 20) - 10
                A(N,h) = A(N,h) + A(k,h) * sol(k)
```

```
46          end do
47      end do
48
49      if (debug == 1) then
50          print *, "The generated linear system is:"
51          do i = 1, M
52              print *, A(:,i)
53          end do
54      end if
55
56      call cpu_time(tStart)
57      h = 1
58      k = 1
59      do while (h <= M .and. k <= N)
60          i_max = h
61          do i = h+1, M
62              if (abs(A(k,i)) > abs(A(k,i_max))) then
63                  i_max = i
64              end if
65          end do
66
67          if (abs(A(k,i_max)) < EPS) then
68              k = k + 1
69          else
70              tmp = A(:,h)
71              A(:,h) = A(:,i_max)
72              A(:,i_max) = tmp
73
74              do i = h+1, M
75                  f = A(k,i) / A(k,h)
76                  A(k,i) = 0
77                  do j=k+1, N
78                      A(j,i) = A(j,i) - A(j,h) * f
79                  end do
80              end do
81              h = h+1
82              k = k+1
83          end if
84      end do
85
86      if (debug == 1) then
87          print *, "The linear system in echolon form is:"
88          do i = 1, M
89              print *, A(:,i)
90          end do
91      end if
92
93      k = N - 1
94      h = M
95      do while (k > 0 .and. h > 0)
96          if (abs(A(k,h)) < EPS) then
97              h = h - 1
98          else
99              sol_back(k) = A(N,h)  / A(k,h)
100             do i=1, h-1
101                 A(N,i) = A(N,i) - A(k,i) * sol_back(k)
102             end do
103             h = h - 1
104             k = k - 1
105         end if
106     end do
107
108     call cpu_time(tEnd)
```

```
109         timeTaken = timeTaken + (tEnd - tStart)
110
111         if (debug == 1) then
112             print *, "The computed solution is:"
113             print *, sol_back
114
115             do i=1, N-1
116                 if (abs(sol(i) - sol_back(i)) > EPS) then
117                     call exit(1)
118                 end if
119             end do
120         end if
121     end do
122
123     write (*, *) REPS, "solutions for ", N-1, " dof took ", timeTaken, " seconds."
124     call exit(0)
125
126 end program kppr_5_2
```

Listing 4: Der verwendete Programmcode in Fortran zur Aufgabe 5.2.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <math.h>
5  #include <string.h>
6
7  #define EPS 0.000001
8  #define REPS 100
9
10 int main(int argc, char** argv) {
11     int M = atoi(argv[1]);
12     int N = atoi(argv[2]);
13     int debug = atoi(argv[3]);
14
15     double (*A)[N] = malloc(sizeof(double[M][N]));
16     double *sol = malloc((N-1)*sizeof(double));
17     double *tmp = malloc(N * sizeof(double));
18     double *sol_back = malloc((N - 1) * sizeof(double));
19
20     srandom(time(NULL));
21     struct timespec tStart, tEnd;
22     double timeTaken = 0;
23
24     for (int r = 0; r < REPS; r++) {
25         #pragma omp parallel for
26         for (int n = 0; n < N - 1; n++) {
27             sol[n] = (random() % 20) - 10;
28         }
29
30         if (debug) {
31             printf("Running with a max of %d threads.\n\n", omp_get_max_threads());
32             printf("The generated solution vector is:\n");
33             for (int n = 0; n < N - 1; n++) {
34                 printf("%.1f\t", sol[n]);
35             }
36             printf("\n\n");
37         }
38
39         #pragma omp parallel for
40         for (int m = 0; m < M; m++) {
41             A[m][N - 1] = 0;
42             for (int n = 0; n < N - 1; n++) {
```

```
43                  A[m][n] = (random() % 20) - 10;
44                  A[m][N - 1] += A[m][n] * sol[n];
45              }
46          }
47
48          if (debug) {
49              printf("The generated linear system is:\n");
50              for (int m = 0; m < M; m++) {
51                  for (int n = 0; n < N; n++) {
52                      printf("%.2f\t", A[m][n]);
53                  }
54                  printf("\n");
55              }
56              printf("\n\n");
57          }
58
59          clock_gettime(CLOCK_MONOTONIC, &tStart);
60          int h = 0, k = 0;
61
62          while (h < M && k < N) {
63              int i_max = h;
64              for (int i = h + 1; i < M; i++) {
65                  if (fabs(A[i][k]) > fabs(A[i_max][k])) {
66                      i_max = i;
67                  }
68              }
69
70              if (fabs(A[i_max][k]) < EPS) {
71                  k++;
72              } else {
73                  //swap rows h and i_max
74                  memcpy(tmp, &A[h][0], N * sizeof(double));
75                  memcpy(&A[h][0], &A[i_max][0], N * sizeof(double));
76                  memcpy(&A[i_max][0], tmp, N * sizeof(double));
77
78                  #pragma omp parallel for
79                  for (int i = h + 1; i < M; i++) {
80                      double f = A[i][k] / A[h][k];
81                      A[i][k] = 0;
82                      for (int j = k + 1; j < N; j++) {
83                          A[i][j] = A[i][j] - A[h][j] * f;
84                      }
85                  }
86                  h++;
87                  k++;
88              }
89          }
90
91          if (debug) {
92              printf("The computed echolon form is:\n");
93              for (int m = 0; m < M; m++) {
94                  for (int n = 0; n < N; n++) {
95                      printf("%.2f\t", A[m][n]);
96                  }
97                  printf("\n");
98              }
99              printf("\n\n");
100         }
101
102         int n = N - 2;
103         int m = M - 1;
104         while (n >= 0 && m >= 0) {
105             if (fabs(A[m][n]) < EPS) {
```

```
                m--;
            } else {
                sol_back[n] = A[m][N - 1] / A[m][n];
                #pragma omp parallel for
                for (int i = 0; i < m; i++) {
                    A[i][N - 1] -= A[i][n] * sol_back[n];
                }
                m--;
                n--;
            }
        }

        clock_gettime(CLOCK_MONOTONIC, &tEnd);
        timeTaken += tEnd.tv_sec - tStart.tv_sec + (tEnd.tv_nsec - tStart.tv_nsec) * 1e-9; // in seconds

        if (debug) {
            printf("The computed solution vector is:\n");
            for (int n = 0; n < N - 1; n++) {
                printf("%.1f\t", sol_back[n]);
            }
            printf("\n\n");

            int retval = 0;
            for (int n = 0; n < N - 1; n++) {
                if (fabs(sol[n] - sol_back[n]) > EPS) {
                    printf("Solution was either wrong or not unique!\n");
                    retval = 1;
                }
            }
            return retval;
        }
    }
    printf("%d solutions for %d dof took %f seconds.\n", REPS, N-1, timeTaken);
    return EXIT_SUCCESS;
}
```

Listing 5: Der verwendete Programmcode in C zur Aufgabe 5.3.

```
program kppr_5_3
use OMP_LIB
    implicit none
    integer :: M, N, h, i, j, k, debug, r, REPS, tStart, tEnd, rate
    integer :: i_max
    integer :: seed
    doubleprecision :: f, EPS
    doubleprecision, dimension (:,:), allocatable :: A
    doubleprecision, dimension (:), allocatable :: sol, sol_back
    doubleprecision, dimension (:), allocatable :: tmp
    real :: timeTaken

    character(len=32) :: arg
    call get_command_argument(1, arg)
    read (arg,'(I32)') M
    call get_command_argument(2, arg)
    read (arg,'(I32)') N
    call get_command_argument(3, arg)
    read (arg,'(I32)') debug

    allocate (A(M,N))
    allocate (sol(N-1))
    allocate (sol_back(N-1))
    allocate (tmp(N))
```

```
25      EPS = 0.000001
26      REPS = 100
27      timeTaken = 0
28
29      call system_clock(seed)
30      call srand(seed)
31
32      call system_clock(count_rate=rate)
33
34      do r = 1, REPS
35          !$OMP DO
36          do k = 1, N-1
37              sol(k) = modulo(irand(), 20) - 10
38          end do
39          !$OMP END DO
40
41          if (debug == 1) then
42              print *, "The generated solution vector is:"
43              print *, sol
44          end if
45
46          !$OMP DO
47          do h = 1, M
48              A(h,N) = 0
49              do k = 1, N-1
50                  A(h,k) = modulo(irand(), 20) - 10
51                  A(h,N) = A(h,N) + A(h,k) * sol(k)
52              end do
53          end do
54          !$OMP END DO
55
56          if (debug == 1) then
57              print *, "The generated linear system is:"
58              do i = 1, M
59                  print *, A(i,:)
60              end do
61          end if
62
63          call system_clock(tStart)
64          h = 1
65          k = 1
66          do while (h <= M .and. k <= N)
67              i_max = h
68              do i = h+1, M
69                  if (abs(A(i,k)) > abs(A(i_max,k))) then
70                      i_max = i
71                  end if
72              end do
73
74              if (abs(A(i_max,k)) < EPS) then
75                  k = k + 1
76              else
77                  tmp = A(h,:)
78                  A(h,:) = A(i_max,:)
79                  A(i_max,:) = tmp
80
81                  !$OMP PARALLEL PRIVATE(i,j,f) SHARED(A,h,k,M,N)
82                  !$OMP DO
83                  do i = h+1, M
84                      f = A(i,k) / A(h,k)
85                      A(i,k) = 0
86                      do j=k+1, N
87                          A(i,j) = A(i,j) - A(h,j) * f
```

```
88                      end do
89                  end do
90                  !$OMP END DO
91                  !$OMP END PARALLEL
92                  h = h+1
93                  k = k+1
94              end if
95          end do
96
97          if (debug == 1) then
98              print *, "The linear system in echolon form is:"
99              do i = 1, M
100                 print *, A(i,:)
101             end do
102         end if
103
104         k = N - 1
105         h = M
106         do while (k > 0 .and. h > 0)
107             if (abs(A(h,k)) < EPS) then
108                 h = h - 1
109             else
110                 sol_back(k) = A(h,N)  / A(h,k)
111                 !$OMP PARALLEL PRIVATE(i) SHARED(A,k,h,M,N,sol_back)
112                 !$OMP DO
113                 do i=1, h-1
114                     A(i,N) = A(i,N) - A(i,k) * sol_back(k)
115                 end do
116                 !$OMP END DO
117                 !$OMP END PARALLEL
118                 h = h - 1
119                 k = k - 1
120             end if
121         end do
122
123         call system_clock(tEnd)
124         timeTaken = timeTaken + (tEnd - tStart)/real(rate)
125
126         if (debug == 1) then
127             print *, "The computed solution is:"
128             print *, sol_back
129
130             do i=1, N-1
131                 if (abs(sol(i) - sol_back(i)) > EPS) then
132                     call exit(1)
133                 end if
134             end do
135         end if
136     end do
137
138     write (*, *) REPS, "solutions for ", N-1, " dof took ", timeTaken, " seconds."
139     call exit(0)
140
141 end program kppr_5_3
```

Listing 6: Der verwendete Programmcode in Fortran zur Aufgabe 5.3.