Computer-Architektur. Sequentielle und parallele Algorithmen

Christian Benjamin Ries

Christian_Benjamin.Ries@fh-bielefeld.de

Fachbereich 3 - Ingenieurwissenschaften und Mathematik

Fachhochschule Bielefeld

5. April 2010

In den letzten Jahren entwickelte sich der Ansatz zur modellgetriebenen Entwicklung in allen Bereichen der Industrie. Zur Umsetzung von neuen Produkten werden im Vorfeld immer weniger reale Modelle, sondern Simulationsmodelle erstellt. Diese Simulationsmodelle werden an Computern modelliert und durch physikalische Eigenschaften erweitert. Je mehr Eigenschaften die Simulationsmodelle erhalten, je aufwendiger werden und länger dauern die Berechnungen dieser Modelle. Diese Ausarbeitung liefert einen Einblick in Technologien und Algorithmen um solche Probleme bewältigen zu können. Große Simulationsmodelle sind in kleinere Modelle teilbar und durch das Zusammenführen der Einzelergebnisse schneller lösbar. Zur Verteilung und Verwaltung solcher Modelle wurden und werden Werkzeuge entwickelt, die bei der Umsetzung hilfreich sind.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen von Parallelrechnern **2**
2.1 Parallele Programmiermodelle . . . 2
2.2 High Performance Computing . . . 2
2.2.1 Multi-Thread-Programmierung . . . 3
2.2.2 Open Multi-Processing . . . 3
2.2.3 Message Passing Interface . . . 4
2.2.4 Grid Computing - Spinhenge@home . . . 4

3 Parallelisierung von Algorithmen **4**
3.1 Ebenen der Parallelität . . . 4

4 Modellierung paralleler Applikationen **5**
4.1 Beispiel: Kosinus-Integration . . . 5
4.2 Beispiel: Matrix-Vektor-Multiplikation . . . 6
4.3 Beispiel: Lineare Gleichungssysteme . . . 8
4.3.1 Grundlagen des Jacobi-Verfahrens . . . 8
4.3.2 Parallele Realisierung des Jacobi-Verfahrens . . . 10

5 Ausblick **10**

1 Einleitung

Diese Arbeit enthält weite Teile aus [5], welche noch heute einen sehr hohen Einfluss auf die Entwicklung von parallel arbeitenden Applikationen haben. Wer sich für den Kauf eines neuen Rechners entscheidet, der bekommt

nicht nur einen Prozessor. Moderne Prozessoren besitzen mehr als einen Kern[1]. Die Anzahl der Kerne bestimmt die Geschwindigkeit zur Lösung von größeren mathematischen Problemstellungen (vgl. [1], [3]). Dazu werden die Problemstellungen in Teilaufgaben zerlegt, die parallel zueinander ausgeführt werden können. Typischerweise sind die erzeugten Teilaufgaben nicht vollkommen unabhängig voneinander, sondern können durch Daten- und Kontrollabhängigkeiten gekoppelt sein. Diese Abhängigkeiten müssen durch Synchronisationen[2] zwischen den Teilaufgaben geregelt werden, so dass weitere Berechnungen folgen können.

2 Grundlagen von Parallelrechnern

Parallelrechner sind mittlerweile seit vielen Jahren im Einsatz, wobei bei der Realisierung dieser Rechner viele unterschiedliche Architekturansätze verfolgt werden. Die nachfolgende Definition liefert einen ersten Schritt zum Verständnis von Parallelrechnern:

> *Ein Parallelrechner ist eine Ansammlung von Berechnungseinheiten (Prozessoren), die durch koordinierte Zusammenarbeit große Probleme schnell lösen können.*

Diese Definition ist bewusst vage gehalten, um die Vielzahl der entwickelten Parallelrechner zu erfassen und lässt daher auch viele z.T. wesentliche Details offen. Dazu gehören z.B. die Anzahl und Komplexität der Berechnungseinheiten (BE), die Struktur der Verbindungen zwischen den BE, die Koordination der Arbeit der BE und die wesentlichen Eigenschaften der zu lösenden Probleme. Eine Klassifizierung der Parallelrechner nach wichtigen Charakteristika ist nützlich.

2.1 Parallele Programmiermodelle

Das Erstellen eines parallelen Programms orientiert sich stark an dem zu benutzenden parallelen Rechnersystem[3]. Rechnersystemen können durch Definitionen von *Modellen* klassifiziert werden. Es werden folgende Modelle unterschieden: Maschinenmodelle (engl. *machine models*), Architekturmodelle (engl. *architectural models*), Berechnungsmodelle (engl. *computational models*) und Programmiermodelle (engl. *programming models*). **Maschinenmodelle** stellen die niedrigste Abstraktionsstufe dar und bestehen aus einer hardwarenahen Beschreibung des Rechners und des Betriebssystem. **Architekturmodelle** stellen Abstraktionen von Maschinenmodellen dar und beschreiben die Topologie des Verbindungsnetzwerkes oder Speicherorganisation. Ein **Berechnungsmodell** beschreibt u.a. Algorithmen zum Zugriff auf Speicherresourcen, z.B. sequentieller Zugriff auf den *Random-Access-Memory* (RAM). Das **Programmiermodell** bildet eine Abstraktion des Berechnungsmodell und beschreibt ein paralleles Rechnersystem aus Sicht einer Programmiersprache oder Programmierumgebung. Der Abschnitt 2.2ff enthält Programmiermodelle zur Umsetzung von verteilten Berechnungsproblemen.

2.2 High Performance Computing

Im Bereich des *High Performance Computing* (HPC) wird generell zwischen zwei Umsetzungsprinzipien unterschieden - gemeinsam und verteilt genutzter Adressraum.

1. **Gemeinsamer Adressraum:** In Applikationen mit gemeinsamen Adressraum teilen sich die beteiligten Prozesse einen Speicherbereich. In diesem Speicherbereich werden die Daten für die Berechnungen gehalten. Die Applikation ist für die Verwaltung und Synchronisation der Zugriffe auf diese Bereiche verantwortlich. Jeder Prozess kann **alles** „sehen“ und modifizieren. Die Technologien *Multi-Thread-Programmierung* (vgl. Abs. 2.2.1) und *Open Multi-Processing* (vgl. Abs. 2.2.2) basieren auf dem Prinzip des gemeinsamen Adressraums.

2. **Verteilter Adressraum:** In Applikationen mit verteilten Adressraum besitzen die jeweiligen Prozesse einen eigenen Speicherbereich. Dieser kann nur von den jeweiligen Prozessen modifiziert werden. Der Applikationsentwickler ist dafür zuständig, dass das mathematische Problem von den jeweiligen Prozessen gelöst werden kann. Alle Informationen zur erfolgreichen Lösung müssen den jeweiligen Prozessen zur Verfügung gestellt werden.

[1] Einheit eines Prozessors zur Ausführung von Befehlen und Berechnungen.
[2] Techniken für die Synchronisation werden im Rahmen dieser Arbeit nicht behandelt.
[3] Allgemeiner Ausdruck für die Gesamtheit von Hardware und Systemsoftware (Betriebssystem, Programmiersprache, Compiler, Laufzeitbibliotheken).

2.2.1 Multi-Thread-Programmierung

Eine der Möglichkeiten zur Programmierung mit gemeinsamen Adressraum ist die Verwendung von Threads, falls diese durch das Betriebssystem unterstützt werden. Viele heutige Betriebssysteme wie UNIX und Windows unterstützen die Programmierung mit Threads als Multi-Thread-Programmierung (*multithreaded programming*) und stellen für den Benutzer Schnittstellen (API, *Application Programming Interface*) zur Verfügung. Threads werden als „leichte Prozesse“ bezeichnet, vgl. [7]. Ein Thread ist eine Art Kopie des Prozesses innerhalb eines Prozesses, der nur einen Teil des Prozesses dupliziert. Dieser beinhaltet u.a. folgende gemeinsam genutzte Daten innerhalb eines Prozesses: Prozessanweisungen, Speicherbereiche. Jeder Thread besitzt seine eigene Kennung, Register-Sets inklusive Programmzähler, Stack-Zeiger und Prioritäten. Alle Threads innerhalb eines Prozesses nutzen den gleichen globalen Hauptspeicher. Damit wird die gemeinsame Nutzung von Informationen erleichert, wobei in diesem Zusammenhang das Problem der *Synchronisation* auftaucht. Durch die Erstellung von Threads können verteilte Applikationen erstellt und den Anforderungen entsprechend priorisiert werden. Bei einem Prozessor mit mehr als einem Kern kann eine Verteilung der Threads auf die jeweiligen Kerne stattfinden. Dies kann die Ausführung beschleunigen, aber auch zu einem höheren Verwaltungs- und Synchronisationsaufwand führen.

2.2.2 Open Multi-Processing

Open Multi-Processing (OpenMP, vgl. [4]) ist eine Spezifikation von Übersetzungsdirektiven, Bibliotheksfunktionen und Umgebungsvariablen, die von einer Gruppe Soft- und Hardwarehersteller mit dem Ziel entworfen wurde, einen einheitlichen Standard für die Programmierung von Parallelrechnern mit gemeinsamen Adressraum zur Verfügung zu stellen. Das Programmiermodell von OpenMP basiert auf parallel arbeitenden Threads. Die Abarbeitung eines mit Hilfe von OpenMP formulierten Programms beginnt mit der Ausführung eines sogenannten *Master-Threads*, der das Programm so lange sequentiell ausführt, bis das erste parallel-Konstrukt auftritt. Mit den durch OpenMP zur Verfügung gestellten Mechanismen zur Steuerung der Parallelität können Programme formuliert werden, die sowohl sequentiell als auch parallel ausgeführt werden können. Es können Programme geschrieben werden, die nur bei einer parallelen Ausführung das gewünschte Ergebnis errechnen. Der Programmierer ist dafür verantwortlich, dass die Programme korrekt arbeiten. Dies gilt auch für die Vermeidung von Konflikten, *Deadlocks* oder zeitkritischen Abläufen (vgl. [8]).

```
1   ...
2   #pragma omp parallel private(row, col, i)
3   {
4    #pragma omp for schedule(static)
5    for( row=0; row < size; row++ ) {
6    #pragma omp parallel shared(MA, MB, MC, size)
7    {
8      #pragma omp for schedule(static)
9      for( col=0; col < size; col++ ) {
10     MC[row][col] = 0.0;
11     for( i=0; i < size; i++ )
12       MC[row][col] += MA[row][i] * MB[i][col];
13     }
14   }
15   }
16  }
17  ...
```

Listing 1: Ausschnitt eines OpenMP-Programm zur parallelen Berechnung einer Matrix-Multiplikation

Die verwendeten OpenMP Anweisungen haben folgende Bedeutungen (vgl. [4]):

#pragma omp parallel private(...) Diese Direktive bewirkt, dass der angegebene Anweisungsblock parallel ausgeführt wird. Wird die Arbeit nicht explizit verteilt, so führen alle Threads die gleichen Berechnungen mit evtl. unterschiedlichen privaten Daten aus. Private Daten werden mit dem Parameter `private(...)` definiert. In diesem Beispiel sind die Variablen `row`, `col`, `i` in jedem Prozess eigenständig und beeinflussen sich nicht gegenseitig.

#pragma omp parallel shared(...) Diese Direktive ist ähnlich der vorangegangenen mit dem Unterschied, dass hier von den Prozessen gemeinsam genutzte Variablen mit `shared(...)` definiert werden. Dies betrifft die Matrix Variablen `Ma`, `MB`, `MC` und die Dimension der Matrizen, die durch `size` definiert wird.

#pragma omp for schedule(...) Innerhalb eines parallelen Bereichs können die durchzuführenden Berechnungen mit Hilfe von speziellen Direktiven zur Verteilung der Arbeit auf die ausführenden Threads verteilt werden. Diese Direktive beschreibt eine `for`-Schleife, im Listing 1 mit statischer Verteilung der Schrittweite in der `for`-Schleife.

2.2.3 Message Passing Interface

Message Passing Interface (MPI) hat sich quasi als Industriestandard durchgesetzt. MPI ist eine Abstraktion eines Parallelrechners mit verteilten Speicher. Ein MPI-Programm besteht aus einer Anzahl von Prozessen mit zugeordneten lokalen Daten. Jeder Prozess kann auf seine lokalen Daten zugreifen und mit anderen Prozessen Informationen durch das explizite Verschicken von Nachrichten austauschen. Nachrichten können mit Hilfe von Punkt-zu-Punkt und globalen Kommunikationsoperationen ausgetauscht werden. Die Beispiele in Abschnitt 4ff sind mit MPI umgesetzt und bieten weitere Veranschaulichungen der MPI Technologie.

2.2.4 Grid Computing - Spinhenge@home

Grid Computing (GC) ist eine Technologie zur Erstellung von Rechnerverbünden (engl. *Clustern*). Ein sehr bekanntes GC basierendes Projekt ist Seti@home[4] zur Suche nach außerirdischer Intelligenz. Das Prinzip des CG ist, dass mathematische Problemstellungen in Arbeitspakete unterteilt, an Rechner im Cluster verschickt, dort berechnet und die Ergebnisse zurückgesendet werden. Die Arbeitspakete müssen unabhängig von anderen Arbeitspaketen lösbar sein. GC sind Server-Client basierend und der Server ist für die Validierung und das Zusammenführen der zurückgeschickten Ergebnisse zuständig. Die Fachhochschule Bielefeld hat mit Spinhenge@home[5] ein solches Cluster aufgebaut.

3 Parallelisierung von Algorithmen

Der Parallelisierung eines gegebenen Algorithmus oder Programms liegt immer ein paralleles Programmiermodell (vgl. Abs. 2.1) zugrunde. In vielen Fällen liegt eine Beschreibung der von einem parallelen Programm durchzuführenden Berechnungen in Form eines sequentiellen Programms oder eines sequentiellen Algorithmus vor. Zur Realisierung des parallelen Programms ist eine **Parallelisierung** erforderlich. Das Ziel besteht meist darin, die Ausführungszeit des sequentiellen Programms durch die Parallelisierung so weit wie möglich zu reduzieren. Die Parallelisierung kann in mehrere Schritte (vgl. Abb. 1) zerlegt werden:

1. **Zerlegung der durchzuführenden Berechnungen.** Die Berechnung des Algorithmus werden in parallel ausführbare Einheiten (Tasks) zerlegt. Tasks sind die kleinsten Einheiten der Parallelität. Eine Task ist eine beliebige Folge von Berechnungen, die von einem einzelnen Prozessor ausgeführt wird.

2. **Zuweisung von Tasks an Prozesse.** Ein Prozess ist ein abstrakter Begriff für einen Kontrollfluss, der von einem physikalischen Prozessor ausgeführt wird und der nacheinander verschiedene Tasks ausführen kann. Jeder Prozess sollte etwa gleich viele Berechnungen zugeteilt bekommen, so dass eine gute Lastverteilung entsteht. Dieses wird durch das sog. *Scheduling*[6] ermöglicht, dass ein Umschalten zwischen Prozessen umsetzt. Es wird zwischen *statischen*[7] und *dynamischen*[8] *Scheduling* unterschieden.

3. **Abbilden von Prozessen auf physikalische Prozessoren. (Mapping)** Im einfachsten Fall existiert für jeden Prozessor ein Prozess, so dass die Abbildung einfach durchzuführen ist. Gibt es mehr Prozesse als Prozessoren, müssen mehrere Prozesse auf einen Prozessor abgebildet werden. Bei weniger Prozessen als Prozessoren bleiben die übrigen Prozessoren unbeschäftigt.

3.1 Ebenen der Parallelität

Prinzipiell kann zwischen vier Ebenen unterschieden werden: (1) Instruktionsebene, (2) Datenebene, (3) Schleifenebene und (4) Funktionsebene.

1. Bei der Abarbeitung eines Programms können oft mehrere Instruktionen gleichzeitig ausgeführt werden. Dies ist dann der Fall, wenn die Instruktionen unabhängig voneinander sind.

2. In vielen Programmen werden dieselben Operationen auf unterschiedliche Elemente einer Datenstruktur angewendet. Im einfachsten Fall sind dies die Elemente eines Feldes. Wenn die angewendeten Operationen unabhängig voneinander sind, kann diese verfügbare Parallelität dazu genutzt werden, die Operationen zur Manipulation der Daten auf verteilte Prozesse aufzuteilen. Dieses Prinzip wird in den Beispielen in Abschnitt 4 mit MPI veranschaulicht.

[4] http://setiathome.ssl.berkeley.edu
[5] http://spin.fh-bielefeld.de
[6] Zeitablaufsteuerung mehrerer unterschiedlicher Tasks.
[7] Zuteilung der Tasks beim Programmstart.
[8] Zuteilung der Tasks an Prozesse während der Abarbeitung möglich.

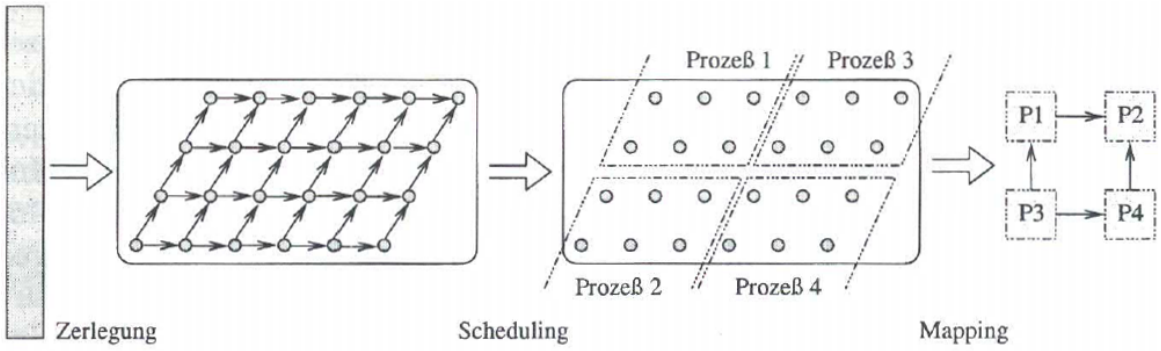

Abb. 1: Schritte zur Parallelisierung eines Anwendungsalgorithmus (vgl. [5])

3. Wenn zwischen den Iterationen einer Schleife keine Abhängigkeiten bestehen, können diese parallel zueinander von verschiedenen Prozessoren ausgeführt werden. In diesem Fall kann die Schleife als parallele Schleife im Programm dargestellt werden. Es wird zwischen drei Schleifentypen unterschieden (vgl. [10]): (1) **for**-Schleife, (2) forall-Schleifen und (3) dopar-Schleife. Für das Verständnis dieser Ausarbeitung ist ein Verständnis der Unterschiede dieser Schleifentypen nicht relevant.

4. In vielen sequentiellen Programmen sind verschiedene Programmteile unabhängig und können parallel zueinander ausgeführt werden. Bei den Programmteilen kann es sich um einzelne Anweisungen, Grundblöcke (engl. *basic blocks*), unterschiedliche Schleifen oder Funktionsaufrufen handeln.

4 Modellierung paralleler Applikationen

Dieser Abschnitt enthält Beispiele für die Parallelisierung von Berechungsapplikationen. Es werden Umsetzungen mit MPI betrachtet, die Quelltexte der Anwendungen sind unter [6] zu finden.

4.1 Beispiel: Kosinus-Integration

Die Integration einer Kosinus-Funktion kann exakt oder numerisch durchgeführt werden. Die Funktion $y = cos(x)$, integriert nach $\frac{dy}{dx}$, liefert folgende Stammfunktion $y' = -sin(x)|_{Untergrenze}^{Obergrenze} = sin(Untergrenze) - sin(Obergrenze)$. Für $Untergrenze = 0$ und $Obergrenze = 2\pi$ ist die exakte Lösung $y' = 0$. In der Regel ist die Stammfunktion nicht so trivial wie die der Kosinus-Funktion. In technischen Anwendungen sind meist gekoppelte Differentialgleichungen in einem Gleichungssystem (GLS) vorhanden, wie z.B. die Navier-Stokes Gleichungen. Diese sind nicht einfach exakt lösbar und benötigen ein numerisches Lösungsverfahrens. Ein Beispiel für die Lösung von GLS wird in Abschnitt 4.3 beschrieben.
Das Integral der Kosinus-Funktion kann numerisch gelöst werden. Zu diesem Zweck werden kleine Rechteckte in den Grenzen $Untergrenze, Obergrenze$ gebildet, die die Fläche innerhalb der Kosinus-Funktion berechnen. Die Summe der Rechtecksflächen ist eine Annäherung an die Lösung des Integrals. Je mehr Unterteilungen in den jeweiligen Prozessen berechnet werden, desto genauer wird das Ergebnis. Das Listing 2 liefert eine exemplarische Umsetzung einer numerischen Integration mit Hilfe von MPI. In den Zeilen $12 - 15$ wird MPI initialisiert und für die jeweiligen Prozesse die Identifizierungsnummer (ID) ermittelt, da die Funktion MPI_Init(&argc,&argv) die Prozesse für die verteilten Rechner erstellt. Mit der ID kann in Zeile 20 die Untergrenze der Prozessunterteilungen berechnet werden. Der linke Teil in Zeile 19 ermittelt die Schrittweite der Iterationen, diese entsprechen der Breite eines Rechtecks. Der rechte Teil in derselben Zeile berechnet den Abstand zur jeweiligen Obergrenze in einer Iteration. Die Zeilen $22 - 29$ werden auf dem Rechner ausgeführt, der die Ausführung des Programms gestartet hat und empfängt in der Zeile 25 die Einzelergebnisse der jeweiligen Prozesse aus Zeile 30 und addiert diese in Zeile 26 auf. Das Ergebnis wird in Zeile 28 ausgegeben.

```
1  double integral(double lowerBound, int n, double step) {
2        double integ = 0.0, step2 = step/2.;
3        for(int j=0; j<n; j++) {
4          double lowerBound_ij = lowerBound + j*step;
5          integ += cos(lowerBound_ij + step2)*step;
6        }
7        return (integ);
8  }
9
10 int main(int argc, char**argv) {
```

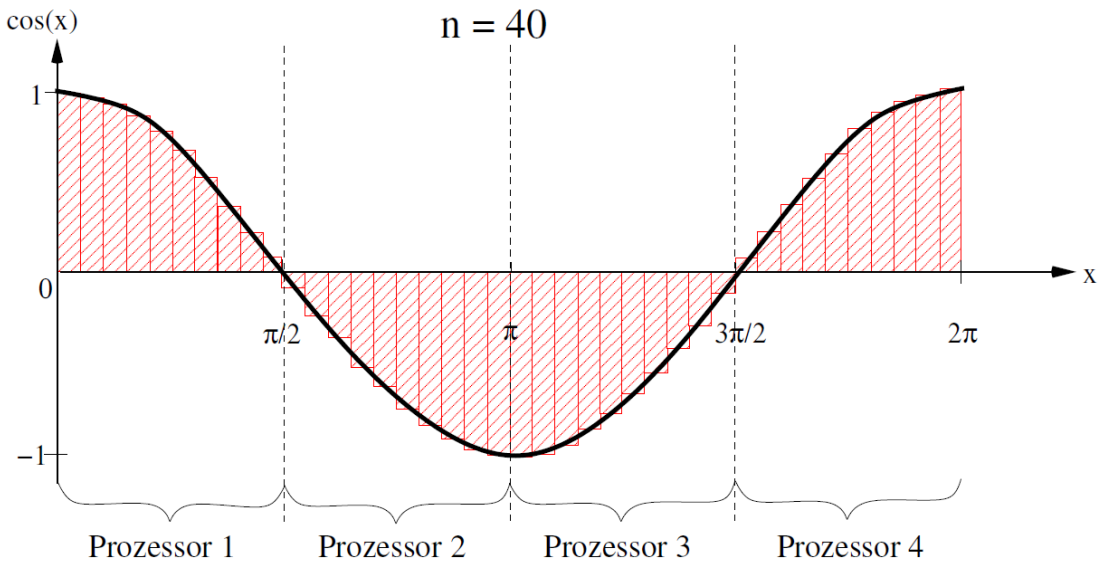

Abb. 2: Numerische Integration einer Kosinus-Funktion, aufgeteilt auf vier Prozesse (vgl. [2])

```
11    ...
12    MPI_Status status;
13    MPI_Init(&argc,&argv);
14    MPI_Comm_rank(MPI_COMM_WORLD, &myID);
15    MPI_Comm_size(MPI_COMM_WORLD, &prozessAnzahl);
16
17    int num = n / prozessAnzahl;
18
19    step = (b-a) / n;                my_range = (b-a) / prozessAnzahl;
20    my_a = a + myID * my_range;      my_result = integral( my_a, num, step);
21
22    if(myID == 0) {
23      my_result = 0.0f;
24      for (int i=1; i < prozessAnzahl; i++) {
25        MPI_Recv(&my_result, 1, MPI_REAL, i, 11111, MPI_COMM_WORLD, &status);
26        result += my_result;
27      }
28      printf("Ergebnis der MPI Berechnung = %.18f\n", result);
29    } else {
30      MPI_Send(&my_result, 1, MPI_REAL, 0, 11111, MPI_COMM_WORLD);
31    }
32    MPI_Finalize();
33  }
```

Listing 2: Ausschnitt einer numerischen Kosinus-Integration mit MPI

4.2 Beispiel: Matrix-Vektor-Multiplikation

Das Beispiel in Listing 5 beinhaltet eine mögliche Lösung für die Parallelisierung einer Matrix-Vektor-Multiplikation für den sequentiellen Algorithmus $c_i = \sum_{j=1}^{m} a_{ij} b_j, i = 1, \ldots, n, \mathbf{c} = (c_1, \ldots, c_n) \in \mathbb{R}^n$. Eine Matrix-Vektor-Multiplikation kann auf verschiedene Weisen durchgeführt werden, z.B. durch die **Verteilung der Skalarprodukte** (vgl. Listing 5 und 3) oder **Verteilung der Linearkombinationen** (vgl. Listing 4). Abbildung 3 verdeutlicht die Prinzipien. Beide Varianten werden für Rechner mit verteilten Speicher betrachtet.

```
1  local_n = n/p;
2  for( i=0; i<local_n; i++ ) local_c[i] = 0.0;
3    for( i=0; i<local_n; i++ )
4      for( j=0; j<m; j++ )
5        local_c[i] += local_A[i][j] * b[j];
6  multi_broadcast(local_c, local_n, c);
```

Listing 3: Realisierung der Matrix-Vektor-Multiplikation durch **Verteilung der Skalarprodukte**

Das Listing 3 enthält ein Schema welches die Datenverteilung von $\mathbf{A}$ und $\mathbf{b}$ so vornimmt, dass ein Prozessor, der die Berechnung eines inneren Produktes $(\mathbf{a}_i, \mathbf{b})$ mit $i \in 1, \ldots, n$ ausführt, die entsprechende Zeile $\mathbf{a}_i$ von $\mathbf{A}$ und den Vektor $\mathbf{b}$ im lokalen Speicher vorliegen hat. Da $\mathbf{b} \in \mathbb{R}^m$ für alle inneren Produkte benötigt wird, wird dieser Vektor repliziert[9] gehalten. Für die Matrix $\mathbf{A}$ ist eine zeilenorientierte streifenweise Datenverarbeitung sinnvoll.

[9] Kopie auf jedem Rechner.

Jeder Prozessor berechnet diejenigen inneren Produkte, für die ihm die zugehörige Zeile von **A** in seinem lokalen Speicher zur Verfügung steht. Da die Matrix-Vektor-Multiplikation häufig als Teilberechnung innerhalb einer größeren Anwendung vorkommt, gibt es oft Anforderungen an die Verteilung des Ergebnisvektors. In einem Iterationsverfahren wird für den Ergebnisvektor **c** meist replizierte Verteilung verwendet. In Abbildung 3(1) wird der Ablauf der Verteilung der Skalarprodukte mit replizierten Ergebnisvektor verdeutlicht.

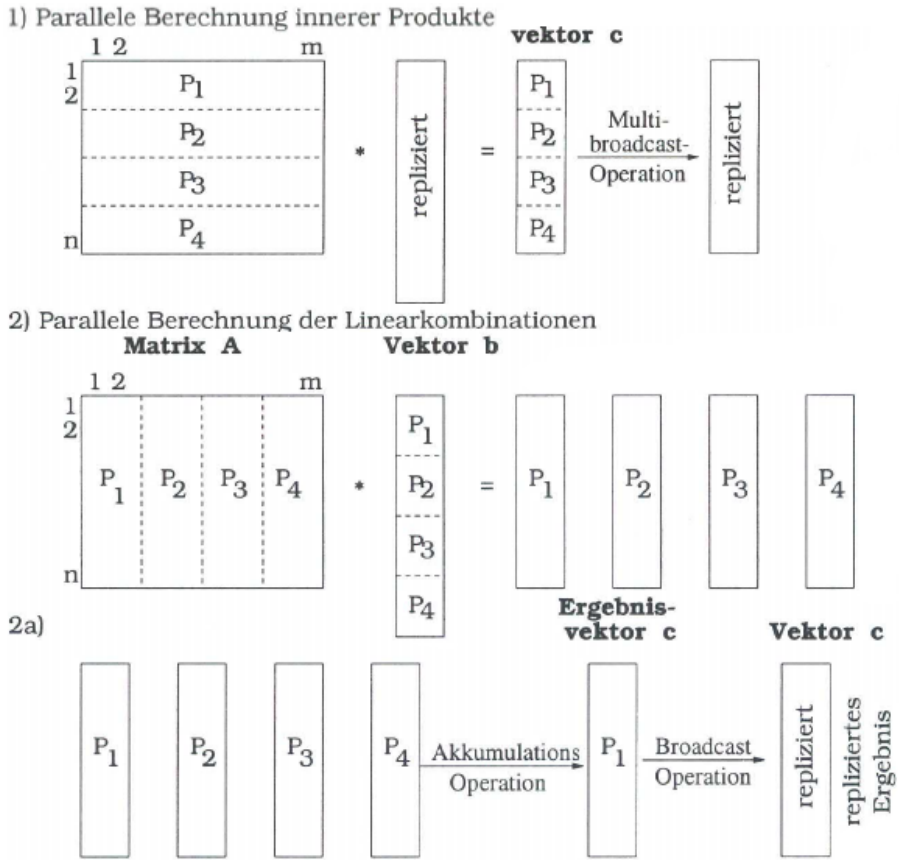

Abb. 3: Schemata der Matrix-Vektor-Multiplikationen mit 1) **Verteilung der Skalarprodukte** 2) **Verteilung der Linearkombinationen**

```
1 local_m = m/p;
2 for( i=0; i<n; i++ ) d[i] = 0;
3 for( j=0; j<local_m; j++)
4   for( i=0; i<n; i++ )
5     d[i] += local_b[j] * local_A[i][j];
6 single_accumulation( d, local_m, c, ADD, 1);
```

Listing 4: Realisierung der Matrix-Vektor-Multiplikation durch **Verteilung der Linearkombinationen**

Die Verteilung der Linearkombinationen entspricht einer spaltenorientierten streifenweise Datenverteilung der Matrix **A**. Das Schema zur Umsetzung ist in Listing 4 nachzulesen und in Abbildung 3(2ff) abgebildet. Es wird in jedem Prozess eine Teil-Linearkombination berechnet: $\mathbf{d}_k = \sum_{j=1}^{m} b_j \mathbf{a}_j$. Dazu benötigt der Prozessor P_k nur einen Block der Komponenten des Vektors **b**. Nach der Berechnung hat jeder Prozessor P_k einen Vektor $\mathbf{d}_k$ der Länge n als Ergebnis. Für die Errechnung der gesamten Linearkombination, die sich durch Addition der Ergebnisse der Teil-Linearkombinationen ergibt, kann eine Akkumulationsoperation mit einer Addition als Reduktionsoperation verwendet werden, zu der jeder Prozessor P_k die von ihm errechnete Teil-Linearkombination $\mathbf{d}_k$ beiträgt. Der Vektor **d** ist ein privater Vektor, den jeder Prozessor zur Berechnung eines Teilergebnisses nutzt. Zur Akkumulation `single_accumulation( d, local_m, c, ADD, 1)` trägt jeder Prozessor seinen lokal errechneten Vektor **d** mit `local_m` Komponenten bei. Als Reduktionsoperation wird eine Addition (ADD) verwendet. Prozessor P_1 ist die Wurzel der Akkumulation und speichert die aktuellen Werte in seinem lokalen Feld `c`.

```
1 #define N 3
2 #define CHEF 0
3
4 int main( int argc, char **argv ) {
5   MPI_Init(&argc, &argv);
```

```
6    int id = 0;       MPI_Comm_rank(MPI_COMM_WORLD, &id);
7    int count = 0;    MPI_Comm_size(MPI_COMM_WORLD, &count);
8
9    double A[N*N] = {
10     1.0, 2.0, 3.0,
11     4.0, 5.0, 6.0,
12     7.0, 8.0, 9.0
13   };
14   double localA[N] = { 0.0 };
15   double x[N] = { 1.0, 2.0, 3.0 };
16   double B[N] = { 0.0 };
17
18   if(N==count) {
19     MPI_Scatter( A, N, MPI_DOUBLE,
20                  localA, N, MPI_DOUBLE,
21                  CHEF, MPI_COMM_WORLD );
22     double result = 0.0;
23     for( int i=0; i<N; i++ ) {
24       result += localA[i] * x[i];
25     }
26     MPI_Gather( &result, 1, MPI_DOUBLE,
27                 B, 1, MPI_DOUBLE,
28                 CHEF, MPI_COMM_WORLD );
29   } else {
30     cout << "Must specify " << N << " processors." << endl;
31   }
32   MPI_Finalize();
33 }
```

Listing 5: Ausschnitt einer Matrix-Vektor-Multiplikation mit MPI durch Verteilung der Skalarprodukte

Das Listing 5 enthält eine MPI Umsetzung einer **Verteilung der Skalarprodukte**. Das Beispiel ist auf eine 3×3 Matrix beschränkt. Die MPI-Funktion `MPI_Scatter(...)` sorgt für die Verteilung der Zeilen der Matrix $\mathbf{A}$. Jeder Prozess erhält eine Zeile. Der Vektor $\mathbf{x}$ ist der Multiplikator und global initialisiert und automatisch in jedem Prozess vorhanden. In den Zeilen $23 - 25$ werden die lokalen Skalarprodukte der jeweiligen Prozesse errechnet. Das Ergebnis steht in der lokalen Variablen `result`. Die Funktion `MPI_Gather(...)` fügt die Teilergebnisse in den Vektor $\mathbf{B}$ zusammen. Das Ergebnis der Ausführung ist in Listing 6 zu finden.

```
1 cr@VisualGrid:~/mpi_ries_matvec$ make run
2 mpiexec -np 3 ./main
3    14   32   50
```

Listing 6: Ergebnis einer Matrix-Vektor-Multiplikation mit MPI und **Verteilung der Skalarprodukte**

4.3 Beispiel: Lineare Gleichungssysteme

Dieser Abschnitt liefert einen Ansatz für die Lösung eines linearen Gleichungssystem (GLS) $\mathbf{A} * \vec{x} = \vec{b}$ mit gegebener $(n \times n)$-Matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ und gegebenen Vektor $\vec{b} \in \mathbb{R}^n$. $\vec{x} \in \mathbb{R}^n$ ist eine Folge von Approximationsvektoren die gegen die Lösung $\vec{x}^* \in \mathbb{R}^n$ konvergiert. An dieser Stelle wird exemplarisch das *Jacobi-Verfahren* (JAC) behandelt. Es wird nicht auf die Eigenschaften der Konvergenzgeschwindigkeit eingegangen oder unter welchen Vorraussetzungen das Verfahren überhaupt konvergiert.

4.3.1 Grundlagen des Jacobi-Verfahrens

Klassische Iterationsverfahren zur Lösung von GLS basieren auf einer Zerlegung der Matrix $\mathbf{A}$ in $\mathbf{A} = \mathbf{M} - \mathbf{N}$, mit $\mathbf{M}, \mathbf{N} \in \mathbb{R}^{n \times n}$, wobei $\mathbf{M}$ eine nichtsinguläre Matrix ist, für die die inverse Matrix $\mathbf{M}^{-1}$ zu berechnen ist. Die gesuchte Lösung $\vec{x}^*$ des GLS $\mathbf{A} * \vec{x} = \vec{b}$ erfüllt die Gleichung $\mathbf{M} * \vec{x}^* = \mathbf{N} * \vec{x}^* + \vec{b}$. Diese Gleichung induziert eine Iterationsvorschrift $\mathbf{M} * \vec{x}^{(n+1)} = \mathbf{N} * \vec{x}^{(n)} + \vec{b}, n = 0, 1, \ldots,$ die typischerweise in folgender Form geschrieben wird $\vec{x}^{(n+1)} = \mathbf{C} * \vec{x}^{(n)} + \vec{d}$ mit $\mathbf{C} := \mathbf{M}^{-1} * N$ und $\vec{d} := \mathbf{M}^{-1} * \vec{b}$.
Das JAC basiert auf der Zerlegung $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{R}, (D, L, R \in \mathbb{R}^{n \times n})$, wobei $\mathbf{D}$ die Diagonale, $-\mathbf{L}$ die untere und $-\mathbf{R}$ die obere Dreicksmatrix (jeweils ohne Diagonale) mit den entsprechenden Elementen von $\mathbf{A}$. Alle übrigen Einträge sind Null.

Die Veranschaulichung in [9] beschreibt die Anwendung für ein GLS mit $N = 3$.

$$\mathbf{A} * \vec{x} = \vec{b} \Rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{1}$$

Unter der Vorraussetzung, dass $a_{ii} \neq 0, (i = 1, 2, 3)$, kann jeweils nach der i-te Gleichung x_i aufgelöst werden.

$$x_1 = \frac{1}{a_{11}} (b_1 - a_{12}x_2 - a_{13}x_3) \tag{2}$$

$$x_2 = \frac{1}{a_{22}} (b_2 - a_{21}x_1 - a_{23}x_3) \tag{3}$$

$$x_3 = \frac{1}{a_{33}} (b_3 - a_{31}x_1 - a_{32}x_2) \tag{4}$$

Wenn eine Startnäherung $\vec{x}^{(n)} = \begin{pmatrix} x_1^{(n)} & x_2^{(n)} & x_3^{(n)} \end{pmatrix}^T$ gegeben ist, kann eine Näherung für $\vec{x}^{(n+1)}$ errechnet werden. Dies sieht folgendermaßen aus:

$$x_1^{(n+1)} = \frac{1}{a_{11}} \left(b_1 - a_{12}x_2^{(n)} - a_{13}x_3^{(n)}\right) \tag{5}$$

$$x_2^{(n+1)} = \frac{1}{a_{22}} \left(b_2 - a_{21}x_1^{(n)} - a_{23}x_3^{(n)}\right) \tag{6}$$

$$x_3^{(n+1)} = \frac{1}{a_{33}} \left(b_3 - a_{31}x_1^{(n)} - a_{32}x_2^{(n)}\right) \tag{7}$$

Dies ist der erste Schritt des *Gesamtschritt-* oder *Jacobi-Verfahren* (JAC).
Für diese Art der Berechnung haben sich Schemata entwickelt. Im exemplarischen Beispiel werden folgende Werte angenommen, die eine angenäherte Lösung $\vec{x} = (0.598 \;\; 0.741 \;\; 0.506)^T$ besitzen.

$$\mathbf{A} * \vec{x} = \vec{b} \Rightarrow \begin{pmatrix} 10 & -4 & -2 \\ -4 & 10 & -4 \\ -6 & -2 & 12 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix} \tag{8}$$

Tabelle 1 enthält das Schema zur Berechnung der Jacobi-Iterierten. Dabei besteht der obere linke 3×3 Block aus den negativen nichtdiagonalen Matrixelementen in transponierter Form. Darunter steht horizontal die rechte Seite des GLS und darunter die Diagonalelemente. Ab der vierten Spalte stehen die (zu berechnenden) Komponenten der iterierten Näherungswerte für die Lösung. In diesem Beispiel berechnet sich $\vec{x}^{(n+1)}$ nach den

			$\vec{x}^{(0)}$	$\vec{x}^{(1)}$	$\vec{x}^{(2)}$	$\vec{x}^{(3)}$	...
0	4	6	0	0.200	0.401	0.499	...
4	0	2	0	0.380	0.559	0.651	...
2	4	0	0	0.247	0.377	0.441	...
2	3	1					
10	10	12					

Tab. 1: Schema zur Berechnung der Jacobi-Iterierten mit einer Startnäherung $\vec{x}^{(0)} = \vec{0}$

Gleichungen 9 bis 11.

$$x_1^{(n+1)} = \left(\underbrace{0 * x_1^{(n)} + 4 * x_2^{(n)} + 2 * x_3^{(n)}}_{erste\ Spalte\ des\ Schemas\ x^{(n)}} + \underbrace{2}_{b_1} \right) / \underbrace{10}_{a_{11}} \tag{9}$$

$$x_2^{(n+1)} = \left(\underbrace{4 * x_1^{(n)} + 0 * x_2^{(n)} + 4 * x_3^{(n)}}_{zweite\ Spalte\ des\ Schemas\ x^{(n)}} + \underbrace{3}_{b_2} \right) / \underbrace{10}_{a_{22}} \tag{10}$$

$$x_3^{(n+1)} = \left(\underbrace{6 * x_1^{(n)} + 2 * x_2^{(n)} + 0 * x_3^{(n)}}_{dritte\ Spalte\ des\ Schemas\ x^{(n)}} + \underbrace{1}_{b_3} \right) / \underbrace{12}_{a_{33}} \tag{11}$$

4.3.2 Parallele Realisierung des Jacobi-Verfahrens

Aus den vorherigen Gleichungen ist erkennbar, dass die Berechnungen der verschiedenen Komponenten $x_i^{(n+1)}$ für $i = 1, \ldots, n$ unabhängig voneinander und parallel zueinander ausgeführt werden können. Jeder Iterationsschritt kann von ebenso vielen Prozessoren wie Gleichungen parallel ausgeführt werden. Der berechnete neue Wert $x_i^{(n+1)}$ ist bei Rechnern mit verteiltem Speicher zunächst nur auf dem berechnenden Prozessor verfügbar. Zur Berechnung einer Komponente eines Iterationsvektors $\vec{x}^{(n+1)}$ wird der komplette vorherige Vektor $\vec{x}^{(n)}$ benötigt. Es muss hierfür eine replizierte Verteilung verwendet werden. In Listing 7 wird in den Zeilen $20 - 28$ der Wert für $x_i^{(n+1)}$ berechnet und in Zeile 30 an den Prozess mit der Identifizierungsnummer 0 gesendet. Dieser Prozess fasst die Ergebnisse im Vektor `x_next` zusammen. Die MPI Funktionen in den Zeilen $15 - 18$ versenden an die erstellten Prozesse die relevanten Daten für die Berechnung der nächsten Iterationsvektoren. Diese Daten sind aus den vorherigen Gleichungen 9 bis 11 zu entnehmen. Die Namen der Vektoren und Matrizen aus Listing 7 entsprechen den Namen in diesen Gleichungen. Zu beachten ist, dass die mathematischen Gleichungen mit einem Zugriffsindex *Eins* beginnen und in der Programmierung mit *Null*.

```
1  int main( int argc, char **argv ) {
2    static double A[] = { 10.0, -4.0, -2.0, -4.0, 10.0, -4.0, -6.0, -2.0, 12.0 };
3    static double b[] = { 2.0, 3.0, 1.0 };
4    double x_next[N] = { 0.0 };
5    double x_prev[N] = { 0.0 };
6    double localA[N] = { 0.0 };
7    double localb = 0.0;
8
9    int id, pcount;
10   MPI_Init( &argc, &argv );
11   MPI_Comm_rank( MPI_COMM_WORLD, &id);
12   MPI_Comm_size( MPI_COMM_WORLD, &pcount);
13
14   for( int steps=0; steps<MAXIT; steps++ ) {
15     MPI_Scatter( A, 3, MPI_DOUBLE, localA, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
16     MPI_Scatter( b, 1, MPI_DOUBLE, &localb, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
17     MPI_Bcast( x_next, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
18     MPI_Bcast( x_prev, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
19
20     double a_diagonal = localA[id]; localA[id] = 0.0;
21     for( int i=0; i<N; i++ ) {
22       localA[i] *= -1.0;
23     }
24     double localx = localb;
25     for( int i=0; i<N; i++ ) {
26       localx += (localA[i] * x_prev[i]);
27     }
28     localx /= a_diagonal;
29
30     MPI_Gather( &localx, 1, MPI_DOUBLE, x_next, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
31
32     duplicateVector(x_next, x_prev, N);
33   }
34   if(id == 0) {
35     displayVector<double>(x_next, N);
36   }
37   MPI_Finalize();
38   return 0;
39 }
```

Listing 7: Ausschnitt des Jacobi-Verfahrens mit MPI

Das Lisiting 8 enthält das Kommando zum Aufrufen der MPI Berechnung mit dem JAC-Verfahren. Das Ergebnis (engl. *result*) der Berechnung ist eine sehr gute Näherung und entspricht dem Ergebnis aus Abschnitt 4.3.1.

```
1 cr@VisualGrid:~/mpi_jacobi_03$ make run
2 mpiexec -np 3 ./main
3 result: 0.598  0.741  0.506
```

Listing 8: Ergebnis des Jacobi-Verfahrens mit MPI

5 Ausblick

MPI ist quasi ein Industriestandard für das Verteilen von Berechnungen in einem Cluster. Simulationswerkzeuge wie COMSOL[10] unterstützen von Haus aus eine MPI Kommunikation bei vorhandenen Cluster. Die vorgestellten Beispiele sind nur eine kleine Auswahl von Berechnungsproblemen, in denen eine Verteilung umgesetzt

[10] http://www.comsol.de - COMSOL:Multiphysics Modeling and Simulation

werden kann. Das Ziel der Hardwarehersteller ist nicht mehr höhere Taktleistungen einzelner Prozessoren herzustellen, sondern mehrere Kerne in einem Prozessor zu verbinden. Die Rechenleistungen von Grafikkarten ist ein weiterer Sektor für Berechnungsaufgaben in der Forschung und Industrie. ATI[11] und NVidia[12] sind zwei Grafikkartenhersteller die Schnittstellen[13] für die Entwicklung von Applikationen bereitstellen, welche auf Grafikkarten Berechnungen durchführen können. Eine Kombination von MPI Applikationen, mit der Möglichkeit Berechnungen auf Grafikkarten auszuführen, stellt den nächsten Schritt zur Erhöhung der Rechenleistung dar. Einige *Grid Computing* Projekte[14] unterstützen die Nutzung von Grafikkartenberechnungen, so sind schnellere Ergebnisse erzielbar.

Literatur

[1] Amdahl, Gene M.: **Validity of the single processor approach to achieving large scale computing capabilities**, 30:483-485, 1967

[2] T.Englisch: **High Performance Computing**, Fachhochschule Bielefeld, Energietag 2010

[3] Gustafson, John L.: **Reevaluating Amdahl's Law**, Comm. ACM, 31:532-533, 1988

[4] **OpenMP Specification**, Version 3.0, Mai 2008

[5] Rauber, Rünger: **Parallele und verteilte Programmierung**, Springer-Verlag Berlin Heidelberg, 2000

[6] C.B.Ries: **http://ti.fh-bielefeld.de/~cries**

[7] W.Richard Stevens: **Programmieren von UNIX-Netzwerken**, 2. Auflage, Carl Hanser Verlag München Wien, 2000

[8] Tanenbaum, Andrew S.: **Modern Operating Systems**, 2. Auflage, Prentice Hall International, 2001

[9] Trottenberg: **Gesamt- und Einzelschrittverfahren bzw. Jacobi- und Gauss-Seidel-Verfahren für N=3**, Fraunhofer Institut, Mai 2005

[10] M.Wolfe: **High Performance Compilers for Parallel Computing**, Addison-Wesley, 1995

[11] http://ati.amd.com
[12] http://www.nvidia.com
[13] ATI: http://developer.amd.com/GPU - NVidia: http://www.nvidia.com/object/cuda_home_new.html
[14] http://www.gpugrid.net - http://folding.stanford.edu