

Parallelverarbeitung – Architekturen, Sprachen, Algorithmen

Daniel Seither

Zusammenfassung

Parallelverarbeitung spielt in informationsverarbeitenden Systemen eine immer größere Rolle: Mehrkernprozessoren ersetzen im Personal Computer den althergebrachten Einkernprozessor, erste Supercomputer bestehen aus über 100.000 CPUs und selbst die Vektoreinheiten in Grafikkarten werden für die Beschleunigung wissenschaftlicher Berechnungen genutzt. Dieser Artikel gibt einen Überblick über parallele Architekturen, Sprachen zur Programmierung dieser Systeme und Algorithmen, die Parallelität effizient nutzen.

1. Architekturen

1.1. Von-Neumann-Architektur

John von Neumann entwickelte 1945 an der University of Pennsylvania die grundlegende Struktur des Digitalrechners EDVAC [15]. Die verallgemeinerte Form dieser Architektur ist heute als Von-Neumann-Architektur bekannt.

Ein Von-Neumann-Rechner besteht aus einer zentralen Steuereinheit (control), einer arithmetisch-logischen Einheit (ALU), Speicher, Eingabe- und Ausgabegeräten (Abbildung 1). Ein solcher Rechner führt Befehle nacheinander aus.

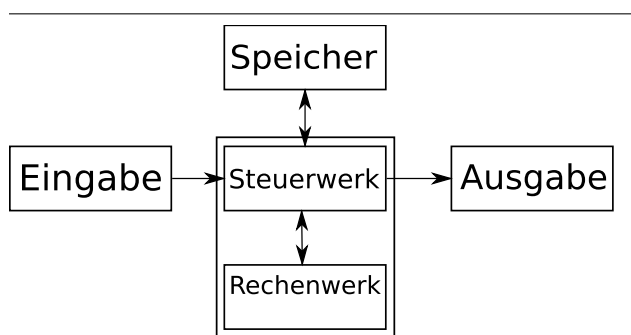


Abbildung 1. Von-Neumann-Rechner

Viele Prozesse aus beispielsweise Natur oder Wirtschaft laufen jedoch gleichzeitig ab; für die Simulation auf einem Von-Neumann-Rechner müssen diese also sequenzialisiert werden. Zudem lassen sich viele Probleme effizienter berechnen, wenn sie in Teilprobleme aufgeteilt und diese von mehreren Recheneinheiten gleichzeitig bearbeitet werden.

1.2. Flynn'sche Klassifikation

Rechner können nach Michael J. Flynn [6] in vier Klassen eingeteilt werden:

- Single Instruction, Single Data (SISD)
- Multiple Instruction, Single Data (MISD)
- Single Instruction, Multiple Data (SIMD)
- Multiple Instruction, Multiple Data (MIMD)

Von-Neumann-Rechner sind SISD-Systeme, da sie nur einen Programmfluss besitzen und pro Instruktion nur einen Datensatz verarbeiten können. MISD, SIMD und MIMD hingegen implizieren eine parallele Verarbeitung. Da für die Klasse MISD kein praktisches Beispiel bekannt ist, wird sie hier nicht betrachtet.

SIMD beschreibt Systeme, die eine Operation gleichzeitig auf viele Datensätze anwenden können (synchrone Parallelität). Wenn verschiedene Befehlsfolgen gleichzeitig auf unterschiedlichen Daten ausgeführt werden können, spricht man von MIMD-Systemen (asynchrone Parallelität).

1.3. MIMD

MIMD-Systeme bestehen aus mehreren unabhängigen Prozessoren¹, die jeweils einen eigenen Programmfluss besitzen. Diese Prozessoren können entweder einen gemeinsamen Adressraum verwenden oder jeweils eigenen Speicher besitzen (gegenübergestellt in Abbildung 2).

Gemeinsamen Speicher verwenden Prozesse, die auf verschiedenen Prozessoren ausgeführt werden, parallel. Die

¹ Prozessorkerne unterscheiden sich in den für diesen Artikel relevanten Eigenschaften nicht wesentlich von einzelnen Prozessoren, so dass das Wort „Prozessoren“ hier sowohl Prozessoren als auch Kerne zusammenfasst.

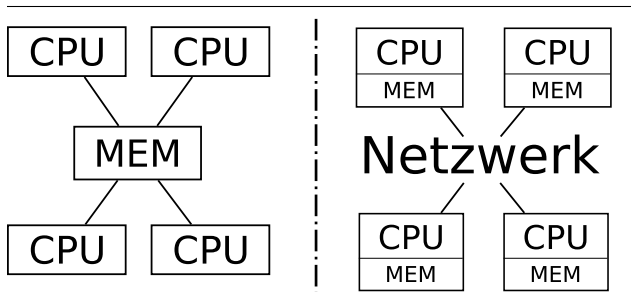


Abbildung 2. MIMD-System

Prozesse können über gemeinsame Speicherbereiche miteinander kommunizieren. Gemeinsamer Speicher kommt vor allem bei Multicore-CPU's und Servern oder Workstations mit mehreren Prozessoren vor.

Falls kein gemeinsamer Speicher existiert, muss die Kommunikation zwischen Prozessen (inter-process communication, IPC) durch Nachrichten geschehen, die über das Verbindungsnetzwerk der Prozessoren ausgetauscht werden.

Auch hybride Lösungen sind denkbar. So kann beispielsweise ein Rechencluster aus vielen Rechenknoten bestehen, die mit mehreren CPUs und eigenem Speicher ausgestattet und über ein Netzwerk verbunden sind. Die CPUs innerhalb eines Knoten verfügen hier über gemeinsamen Speicher während die Kommunikation zwischen Prozessoren verschiedener Knoten über Nachrichten geschehen muss.

1.4. SIMD

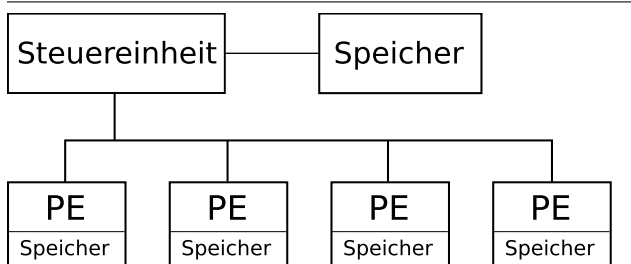


Abbildung 3. SIMD-System

Ein SIMD-System besteht aus mehreren Recheneinheiten mit jeweils lokalem Speicher, einer Steuereinheit, globalem Speicher für Programmcode und Kommunikationsverbindungen zwischen den Recheneinheiten (Abbildung 3). Die Steuereinheit lädt in jedem Zyklus den nächsten Befehl aus dem globalen Speicher, dekodiert ihn, reicht ihn an die Recheneinheiten (auch processing elements, kurz PEs genannt) weiter und aktualisiert den Programmzähler. Die

PEs führen diesen Befehl synchron auf ihren lokalen Daten aus. Im Gegensatz zu MIMD-Systemen gibt es hier also nur einen Programmfluss. Diese Daten hingegen sind typischerweise von PE zu PE unterschiedlich.

Die PEs können auch untereinander Daten austauschen. Manchmal [3] wird in diesem Zusammenhang zwischen Vektor- und Arrayrechnern unterschieden: Vektorrechner haben eine sehr eingeschränkte Verbindungsstruktur. Es ist beispielsweise lediglich ein Datenaustausch zwischen benachbarten PEs möglich, so dass Schiebeoperationen stattfinden können. Arrayrechner hingegen besitzen ein komplexes Verbindungsnetzwerk. Üblicherweise werden die Ausdrücke Vektor- und Arrayrechner jedoch synonym gebraucht.

SIMD-Technik wird in digitalen Signalprozessoren (DSPs) oder modernen Grafikkarten eingesetzt, da hier viele Datensätze – z. B. Audio-Samples oder Geometrie-Punkte – gleichartig transformiert werden müssen. Auch in PC-Prozessoren finden sich SIMD-Erweiterungen, die beschleunigte Multimediafunktionen versprechen (MMX und SSE von Intel, 3DNow! von AMD, AltiVec von Motorola/IBM).

1.5. Synchronisationsalgorithmen

Wenn Prozesse in MIMD-Systemen Daten gemeinsam nutzen oder austauschen möchten oder Zugriff auf eine Ressource benötigen, die nur von einem Prozess gleichzeitig genutzt werden darf, kann dies zu Problemen führen.

Wenn mehrere Prozesse simultan lesend auf einen Speicherbereich zugreifen, so ist dies unkritisch. Wenn jedoch gleichzeitig gelesen und geschrieben wird kann es geschehen, dass die lesenden Prozesse teilweise geänderte und damit inkonsistente Daten erhalten. Für mehrere gleichzeitig schreibende Prozesse gilt ähnliches. Es muss also exklusiver Zugriff eines einzigen Prozesses auf den zu schreibenden Speicherbereich vereinbart werden. Allgemeiner: Der kritische Abschnitt, in dem eine exklusiv zu nutzende Ressource verwendet wird, darf nur von einem Prozess zur gleichen Zeit betreten werden.

Eine solche Synchronisationslösung muss also folgendes garantieren:

- *wechselseitigen Ausschluss (mutual exclusion)*
- *keine Verklemmungen (deadlock/livelock)*: Die Prozesse sollen sich nicht gegenseitig so blockieren, dass keiner in den kritischen Abschnitt eintreten kann.
- *kein Verhungern (starvation)*: Wenn ein Prozess darauf wartet, den kritischen Abschnitt betreten zu können, bekommt er nach endlicher Zeit die Gelegenheit dazu.

Ein einfaches Beispiel für einen Synchronisationsalgorithmus für zwei Prozesse ist *Petersons Algorithmus* [13]. Die im Beispiel in Zeile 4 verwendete `await`-Operation wartet so lange in einer Schleife, bis ihre Bedingung wahr ist (aktives Warten).

Erster Prozess:

```

1 ...
2 flag0 = true
3 turn = 1
4 await (flag1 == false) OR (turn == 0)
5 // kritischer Abschnitt
6 flag0 = false
7 ...

```

Zweiter Prozess:

```

1 ...
2 flag1 = true
3 turn = 0
4 await (flag0 == false) OR (turn == 1)
5 // kritischer Abschnitt
6 flag1 = false
7 ...

```

Man kann mit einem model checker wie SPIN [7] beweisen, dass dieses Verfahren die oben vorgestellten Anforderungen an ein Synchronisationsverfahren erfüllt. Durch Hardwareunterstützung kann der Code weiter vereinfacht werden. In Intels IA-32-Architektur beispielsweise existiert der Befehl `XCHG`, der den Inhalt einer Speicherzelle mit dem eines Registers vertauscht und dabei dafür sorgt, dass kein anderer Prozessor zum gleichen Zeitpunkt auf die betreffende Speicherzelle zugreifen kann [1]. Wenn nun die Variable `lock` angibt, ob die gewünschte Ressource gerade belegt ist und die Funktion `xchg` den ursprünglichen Speicherinhalt zurückgibt und durch `true` ersetzt, so kann in beliebig vielen Prozessen folgender Code verwendet werden:

```

1 ...
2 await xchg(lock) == false
3 // kritischer Abschnitt
4 lock = false
5 ...

```

Durch weitere Konstrukte wie Semaphoren und Monitore kann die Synchronisation weiter optimiert werden. Semaphoren beschränken das aktive Warten auf ein Minimum, Monitore vereinfachen mittels eines höheren Abstraktionsniveaus die korrekte Anwendung.

1.6. Verbindungsstrukturen

Um miteinander kommunizieren zu können benötigen PEs und CPUs ein Verbindungsnetzwerk. Es gibt sehr unter-

schiedliche Ansätze, um diese Kommunikation zu ermöglichen. Die grundlegenden Designziele für solch ein Netzwerk umfassen:

- *kurze Pfade*: Bei einem Datenaustauschvorgang soll die Anzahl der Zwischenstationen auf dem Weg zwischen den Kommunikationspartnern minimal sein.
- *hohe Parallelität*: Es sollen möglichst viele voneinander unabhängige Verbindungen gleichzeitig aktiv sein können.
- *geringe Kosten*: Die Anzahl der Verbindungsleitungen und Verknüpfungspunkten soll minimal sein.

Zudem unterscheidet man zwischen *statischen* und *dynamischen* Strukturen. Dynamische Netzwerke können – im Gegensatz zu statischen Netzwerken – im laufenden Betrieb umkonfiguriert werden.

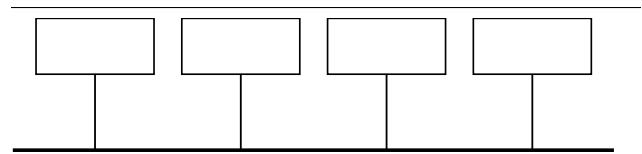


Abbildung 4. Bus

1.6.1. Statische Netze Ein sehr einfaches Beispiel für eine statische Verbindungsstruktur ist der Bus: Alle Prozessoren sind mit dem Bus verbunden (Abbildung 4), wodurch nur eine Leitung pro Prozessor benötigt und die Pfade kurz gehalten werden. Der große Nachteil dieser Lösung ist die äußerst geringe Parallelität, da der Bus nur eine Verbindung zur gleichen Zeit ermöglicht. Dies disqualifiziert den Bus als Verbindungsstruktur für kommunikationsintensive und große Systeme.

Komplexere statische Strukturen sind die Punkt-zu-Punkt-Netze. Teilnehmer sind hier üblicherweise mit mehreren anderen Teilnehmern verbunden. Wenn das Ziel nicht direkt erreicht werden kann, müssen die Daten so lange durch erreichbare Teilnehmer weitergeleitet werden, bis sie das Ziel erreichen. Einfache, wenn auch nicht sehr effiziente bzw. günstige Strukturen sind der Ring und der vollständige Graph. Bei ersterem können jeweils nur die Nachbarn erreicht werden, so dass die Pfadlänge zum Ziel bei n Knoten in $O(n)$ liegt. Im vollständigen Graph kann zwar jeder Knoten direkt erreicht werden, die Anzahl der Verbindungsleitungen liegt jedoch in $O(n^2)$. Einen Kompromiss bilden Gitter und Hypercubes.

In einem m -dimensionalen Gitter besitzt jeder der Knoten $2m$ Nachbarn, mit denen er direkt verbunden ist (Abbildung 5 links). Die Gesamtzahl der Verbindungsleitungen liegt somit in $O(m * n)$ und die maximale Entfernung

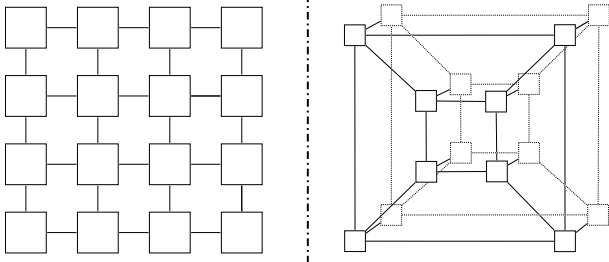


Abbildung 5. 2-D-Gitter und 4-D-Hypercube

zwischen zwei Knoten beträgt $\sqrt[n]{n}$. Effizienter ist ein *m-dimensionaler Hypercube* (Abbildung 5 rechts), in dem sowohl die Zahl der Leitungen als auch die maximale Entfernung in $O(\log n)$ liegen.

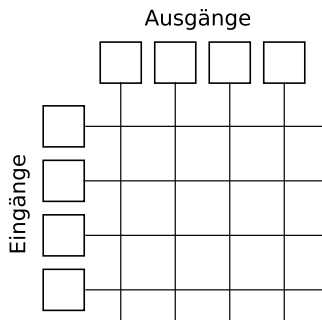


Abbildung 6. Kreuzschienenverteiler

1.6.2. Dynamische Netze Ein einfaches dynamisches Netz ist der *Kreuzschienenverteiler* (Abbildung 6). An jedem der Kreuzungspunkte zwischen Eingängen und Ausgängen kann eine Verbindung konfiguriert werden. Dadurch sind beliebige Verschaltungen bei minimaler Entfernung zwischen den Knoten möglich, jedoch liegt die Zahl der Verbindungspunkte in $O(n^2)$.

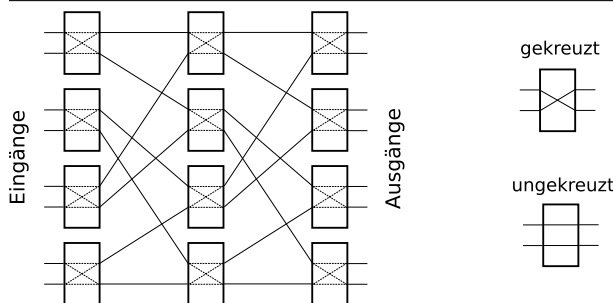


Abbildung 7. Delta-Netzwerk

Für große Knotenmengen muss also ein effizienteres Netz zum Einsatz kommen. Mittels *Delta-Netzwerk* kann bei nur wenig größeren Pfadlängen ($O(\log n)$) die Menge der Schalter erheblich auf $O(n \log n)$ reduziert werden. Jeder Schalter kann seine zwei Eingänge entweder gekreuzt oder ungekreuzt mit seinen beiden Ausgängen verbinden, wobei diese so verknüpft sind, dass von jedem Eingang aus jeder Ausgang erreicht werden kann. Allerdings sind nun nicht mehr beliebige Kommunikationsverbindungen gleichzeitig möglich, da es vorkommen kann, dass zwei Verbindungen den gleichen Schalter in unterschiedlichen Zuständen benötigen. Um dieses Problem zu lösen können Kreuzschienenverteiler und Delta-Netze zu *Clos-Netzwerken* kombiniert werden. Als Grundlage wird ein Delta-Netz verwendet, dessen Schalter durch Kreuzschienenverteiler ersetzt werden.

2. Sprachen

Bei Sprachen für die Entwicklung paralleler Programme muss zwischen verschiedenen Ansätzen unterschieden werden:

Imperative Sprachen dominieren weite Teile der Softwareentwicklung – sowohl bei sequentieller als auch bei paralleler Programmierung. Programme, die in *funktionalen* oder *logischen* Sprachen geschrieben wurden lassen sich besonders einfach parallel ausführen.

2.1. Imperative Sprachen (MIMD)

Um parallele Programme für die Ausführung auf MIMD-Architekturen zu entwickeln, werden häufig C/C++ Java verwendet.

C/C++ besitzen keine eigenen Möglichkeiten, Parallelität zu nutzen. Über Betriebssystemaufrufe oder Bibliotheken, die von den Betriebssystemschnittstellen abstrahieren, kann trotzdem parallel programmiert werden.

Ein Beispiel für Betriebssystemschnittstellen ist die `fork()/wait()`-Kombination unter POSIX-kompatiblen Systemen [8]. `fork()` dupliziert den aufrufenden Prozess, so dass zwei Prozesse mit unabhängigem Kontrollfluss existieren. Sie sind durch den Rückgabewert von `fork()` zu unterscheiden.

Im folgenden Beispiel wird ein zweiter Prozess erzeugt (Zeile 3), so dass die Funktionen `doSomething1()` und `doSomething2()` parallel ablaufen können. Danach wartet der erste Prozess (Zeile 10) auf das Ende des zweiten Prozesses (Zeile 6).

```

1 int main() {
2     // neuer Prozess wird erzeugt
3     if (fork()) {
4         // neuer Prozess
5         doSomething2();

```

```

6         exit(0);
7     } else {
8         // alter Prozess
9         doSomething1();
10        wait(NULL);
11    }
12    return 0;
13 }

```

Zur Kommunikation stehen sowohl Betriebssystem-schnittstellen (für POSIX beispielsweise Semaphoren via `sem_init` etc.) als auch Bibliotheken zur Verfügung. Besonders populär ist die sprachunabhängige Schnittstelle *Message Passing Interface* (MPI), von der Implementierungen für diverse Programmiersprachen existieren.

MPI [10] ermöglicht nachrichtenbasierte Kommunikation sowohl über gemeinsamen Speicher als auch über Netzwerke. Die wichtigsten Operationen sind `MPI_Send` zum blockierenden Senden, `MPI_Recv` zum blockierenden Empfangen, `MPI_Isend` zum nicht-blockierenden Senden und `MPI_Irecv` zum nicht-blockierenden Empfangen von Nachrichten. Blockierend bedeutet in diesem Zusammenhang, dass der Programmfluss nach der betreffenden Anweisung erst dann fortgeführt wird, wenn die Sende- bzw. Empfangsoperation ausgeführt wurde, die Kommunikation also stattgefunden hat.

Ein weiterer Ansatz für C-Programme ist die halbautomatische Parallelisierung mittels OpenMP [11]. Um ein Programm mit parallelen Abläufen zu versehen werden spezielle Compilerdirektiven in den fertigen Code eingefügt, die den OpenMP-kompatiblen Compiler auf Stellen hinweisen, die parallelisiert werden sollen. So können beispielsweise Schleifen oder voneinander unabhängige Berechnungen gleichzeitig ausgeführt werden, ohne dass der Programmierer manuell weitere Prozesse oder Threads erzeugen und verwalten muss.

Der folgende Code[2] zeigt ein Beispiel für die Anwendung von OpenMP in C:

```

1 main () {
2 int nthreads, tid;
3
4 /* Fork a team of threads giving
5  them their own copies of variables */
6 #pragma omp parallel private(tid)
7 {
8
9  /* Obtain and print thread id */
10 tid = omp_get_thread_num();
11 printf("I'm thread = %d\n", tid);
12
13 /* Only master thread does this */
14 if (tid == 0)
15 {

```

```

16     nthreads = omp_get_num_threads();
17     printf("Threads = %d\n", nthreads);
18 }
19
20 } /* All threads join master thread
21    and terminate */
22 }

```

In vielen Programmiersprachen gibt es spezielle Konstrukte, die unabhängig vom Betriebssystem parallele Programmierung ermöglichen. Dies soll hier am Beispiel Java gezeigt werden, da diese Sprache sowohl in der Wirtschaft als auch an den Hochschulen sehr große Verbreitung erlangt hat.

Klassen, die von `java.lang.Thread` abgeleitet wurden oder das Interface `java.lang.Runnable` implementieren, enthalten eine Methode `run()`, die in einem eigenen Thread ausgeführt werden kann. Ein einfaches Beispiel ist durch folgenden Code gegeben:

```

1 class MyThread extends Thread {
2     public void run() {
3         // berechne etwas
4     }
5 }
6
7 // Der Thread kann mit folgendem Aufruf
8 // gestartet werden:
9 new MyThread().run();

```

Ein Thread kann, wenn er auf ein Ereignis wartet oder gerade keine Rechenzeit benötigt, mittels `wait()` entweder für eine gewisse Zeitspanne oder bis zum Eintreffen eines Ereignisses auf Rechenzeit verzichten. Dieses Ereignis kann dem Thread mittels `interrupt()` mitgeteilt werden.

2.2. Imperative Sprachen (SIMD)

Da SIMD-Rechner nur einen Programmfluss besitzen, jedoch viele Datensätze gleichzeitig verarbeiten können, muss die Parallelisierung anders als bei MIMD-Architekturen durchgeführt werden. So liegt der Schwerpunkt auf der Nutzung von in Vektoren oder Matrizen organisierten Daten, deren Elemente parallel und gleichartig verarbeitet werden können. Üblich sind Operationen auf Vektoren oder Matrizen, die wieder auf einen Vektor oder eine Matrix abbilden. Zudem gibt es sogenannte Reduktionsoperationen, die einen skalaren Wert zurückgeben. Ein Beispiel für Reduktion ist die Aufaddierung aller Vektorelemente.

SIMD-Rechner werden oft für wissenschaftliche Berechnungen oder Simulationen eingesetzt. In diesem Bereich hat Fortran auch heute noch einen großen Stellen-

wert. Die Codebeispiele in diesem Abschnitt sind [4] entnommen.

Um drei Matrizen A, B und C mit 23x42 Einträgen anzulegen, verwendet man folgendes Codefragment:

```
1 INTEGER, DIMENSION(23,42) :: A, B, C
```

Nach dieser Definition können die Matrizen bezüglich vieler Operationen wie Skalare verwendet werden, eine Addition zum Beispiel kann wie folgt durchgeführt werden:

```
1 A = B + C
```

Diese Anweisung kann nun je nach Zielplattform entweder sequentiell (SISD) oder auf einem SIMD-System parallel auf mehreren PEs ausgeführt werden. Neben der Addition sind weitere Operationen wie zum Beispiel die Matrixmultiplikation mittels `MATMUL (Mat_A, Mat_B)` vordefiniert.

Es können auch Teile einer vektoriellen Variable verwendet werden. Wenn beispielsweise jedes Element innerhalb des Rechteckes (1, 2) bis (7, 6) einer Matrix M auf den Wert 13 gesetzt werden soll, kann dies so geschehen:

```
1 M(1:7, 2:6) = 13
```

Um vektorielle Daten auf einen skalaren Wert zu reduzieren existieren verschiedene Funktionen:

- `SUM` addiert alle Feldelemente auf
- `PRODUCT` multipliziert alle Feldelemente miteinander
- `MINVAL` und `MAXVAL` bestimmen Minimum bzw. Maximum der Werte
- `ALL` und `ANY` entsprechen einem logischen UND bzw. ODER über die Elemente
- `COUNT` zählt die Elemente mit dem Wahrheitswert „wahr“

Um also das Skalarprodukt zweier fünfelementigen Vektoren zu berechnen, ohne die vordefinierte Funktion `DOTPRODUCT` zu verwenden, reicht folgender Code aus:

```
1 INTEGER, DIMENSION(5) :: A, B, TMP
2 INTEGER RESULT
3 ...
4 TMP = A * B
5 RESULT = SUM(TMP)
```

Auch in gängigen Heimrechnern findet sich SIMD-Technologie. Als Beispiel seien die in Abschnitt 1.4 bereits erwähnten SIMD-Erweiterungen von Intel und AMD genannt, die seit 1997 in praktisch allen PC-Prozessoren Verwendung finden.

Eine sehr viel stärker ausgeprägte Parallelität wird hingegen in frei programmierbaren Grafikprozessoren genutzt,

die spätestens seit der Einführung darauf aufbauender grafischer Oberflächen in MacOS X und Windows Vista stark an Bedeutung gewinnen. Die durch Verwendung dieser Einheiten erzielbare Beschleunigung wurde vom Folding@Home-Projekt der Universität Stanford [12] eindrucksvoll gezeigt. Es handelt sich dabei um ein Projekt zur verteilten Berechnung von Proteinfaltung auf einer hohen Zahl von Rechnern, die von Privatpersonen auf der ganzen Welt zur Verfügung gestellt werden. Seit 2006 gibt es eine Version des Programmes, die auf der Grafikkarte ausgeführt wird und damit ein Vielfaches der Berechnungsgeschwindigkeit der Version für Standardprozessoren erzielt.

Moderne Grafikkarten enthalten eine hohe Zahl an *Shadern* (PEs), die in der Grafikprogrammierung zwei Aufgaben haben: Im ersten Schritt transformieren sie die Koordinaten aller Eckpunkte (Vertices) der 3D-Objekte. Danach wird das Bild von nicht-programmierbarer Hardware gerastert. Im letzten Schritt berechnen sie die Farbe, die jedem einzelnen Bildpunkt (Pixel oder auch Fragment genannt) zugeordnet ist. Der Shader-Programmierer muss also zwei Programme zur Verfügung stellen: Code für die Vertex- und die Pixel-Shader. Diese Programme werden häufig ebenfalls Shader genannt und bei jedem Bildaufbau pro Eck- bzw. Bildpunkt einmal ausgeführt. Da die einzelnen Berechnungen weitestgehend unabhängig voneinander sind, kann die SIMD-Architektur dies durch massiv-parallele Ausführung stark beschleunigen. Durch die freie Programmierbarkeit und die Möglichkeit, Daten wieder zurück zum Hauptspeicher des PC transferieren zu können, können nicht nur Grafik- sondern auch Simulationsdaten für wissenschaftliche Berechnungen verarbeitet werden.



Abbildung 8. Toon Shading

Shader können in verschiedenen Sprachen programmiert werden. Unter OpenGL wird häufig die C-ähnliche *OpenGL Shading Language* (GLSL) verwendet. Als einfaches Beispiel soll folgendes Shader-Paar dienen, das Toon Shading (Comic-artige Einfärbung) für 3D-Objekte berechnet. Da-

bei wird der Farbumfang stark verkleinert, so dass nur noch wenige scharf von einander abgegrenzte Farbflächen übrig bleiben (Abbildung 8, Abbildung und Code entnommen aus [5]).

```

1 // Vertex-Shader
2
3 uniform vec3 lightDir;
4 varying float intensity;
5
6 void main()
7 {
8   vec3 ld;
9   intensity = dot(lightDir,gl_Normal);
10  gl_Position = ftransform();
11 }

```

Dieser Vertex-Shader bestimmt die Intensität des Lichtes, das auf ein Vertex trifft, indem er das Skalarprodukt von Einfallrichtung des Lichts und Normale der Fläche berechnet (Zeile 9). Danach führt er eine vordefinierte Transformation auf den Vertex aus, um den Eckpunkt an die vorgesehene Stelle der Szene zu bewegen (Zeile 10).

```

1 // Fragment-Shader:
2
3 varying float intensity;
4
5 void main()
6 {
7   vec4 color;
8
9   if (intensity > 0.95)
10      color = vec4(1.0,0.5,0.5,1.0);
11  else if (intensity > 0.5)
12      color = vec4(0.6,0.3,0.3,1.0);
13  else if (intensity > 0.25)
14      color = vec4(0.4,0.2,0.2,1.0);
15  else
16      color = vec4(0.2,0.1,0.1,1.0);
17
18  gl_FragColor = color;
19 }

```

Der zugehörige Fragment-Shader verwendet die vom Vertex-Shader errechnete und für den zu berechnenden Pixel automatisch interpolierte Intensität, ordnet sie in Stufen ein (Zeilen 9, 11, 13, 15) und wählt je nach Stufe eine Farbe aus. Schließlich wird der aktuelle Pixel entsprechend eingefärbt (Zeile 18).

2.3. Funktionale und logische Sprachen

Neben den bisher betrachteten imperativen Sprachen² werden auch solche verwendet, in denen das funktiona-

le oder logische Programmierparadigma zur Anwendung kommt. Deren wichtigster Unterschied zum imperativen Paradigma ist, dass nicht die Abfolge der auszuführenden Befehle vorgeschrieben ist sondern Eigenschaften der gewünschten Lösung definiert werden. Für die automatische Parallelisierung ist dies sehr nützlich, da verschiedene Zweige der Berechnung häufig ohne weiteres gleichzeitig ausgeführt werden können.

2.3.1. Funktionale Sprachen. In *funktionalen Sprachen* zu programmieren bedeutet, vordefinierte oder selbstgeschriebene Funktionen zu verknüpfen. Dabei werden meist keine globalen Variablen verwendet, so dass fast alle Funktionen ohne Seiteneffekte ausführbar sind. Die Parameter für einen Funktionsaufruf sind stattdessen entweder Konstanten oder aber Ergebnisse anderer Funktionsaufrufe. Da üblicherweise keine Kontrollstrukturen existieren, werden Fallunterscheidungen als spezielle Funktionen und Schleifen mittels Rekursion implementiert.

Wenn ein Funktionswert bestimmt werden soll, müssen zunächst die Werte der übergebenen Parameter bestimmt werden, falls es sich bei diesen wiederum um Funktionsaufrufe handelt. Mehrere Parameter können parallel evaluiert werden, falls keine der aufgerufenen Funktionen einen Seiteneffekt (Ein-/Ausgabe, Verwendung von globalen Variablen etc.) besitzt, was sich auf Parallelrechnern für eine Beschleunigung ohne die bei imperativen Sprachen meistens nötige Synchronisation nutzen lässt.

Ein häufig verwendeter Vertreter der funktionalen Sprachen ist die Sprache LISP, deren Spezifikation [9] schon 1960 von John McCarthy veröffentlicht wurde. Jede Funktion in LISP wird in Präfixnotation und mit Klammerung geschrieben, hat also das Schema (name param1 param2 ...), wobei auch Funktionen ohne Parameter (Konstanten, ohne Klammern) möglich sind. Auch die wenigen Schlüsselwörter werden mit wenigen Ausnahmen in dieser Form notiert.

Das nächste Beispiel ist in dem heute oft verwendeten Dialekt *Common Lisp* geschrieben und berechnet rekursiv die n -te Fibonacci-Zahl.

```

1 (defun fib (n)
2   (if (< n 2)
3       1
4       (+ (fib (- n 1)) (fib (- n 2))))
5   ))
6
7 (print (fib 5))

```

In Zeile 1 wird zunächst die Funktion `fib` mit dem Parameter n definiert. In Zeile 2 wird nun geprüft, ob das Ende der Rekursion erreicht ist (`fib(n)` hat nach Definition

² Objektorientierte Sprachen sind im eigentlichen Sinne nicht imperativ, besitzen aber ebenfalls eine feste Befehlsabfolge und werden hier deshalb nicht gesondert betrachtet.

den Wert 1 für $n = 0$ und $n = 1$). Falls dies der Fall ist, wird 1 zurückgegeben (Zeile 3). Anderenfalls wird das Ergebnis zweier rekursiver Aufrufe addiert. Diese rekursiven Aufrufe können nun parallel ausgeführt werden, da sie unabhängig voneinander sind.

Der eigentliche Aufruf der Funktion `fib` findet in Zeile 7 statt. Das Ergebnis 8 wird von der Funktion `print` auf dem Bildschirm ausgegeben.

2.3.2. Logische Sprachen. Ein logisches Programm besteht aus einer Menge von Fakten und Regeln, die aus bekannten Aussagen neue Aussagen erzeugen können. Die folgenden Beispiele beziehen sich auf die logische Sprache Prolog.

Der Fakt `mother(alice, bob)` beispielsweise soll bedeuten, dass Alice die Mutter von Bob ist. Nun können Regeln definiert werden, die besagen, dass wenn A Mutter von B oder A Vater von B ist, A ein Elternteil von B ist:

```
1 parent(A, B) :- mother(A, B).
2 parent(A, B) :- father(A, B).
```

`:-` ist dabei wie ein Implikationspfeil \leftarrow zu lesen, mehrere Regeln mit gleicher linker Seite entsprechen einer Disjunktion. Konstanten und Namen von Regeln oder Fakten werden klein geschrieben, Variablen müssen mit einem Großbuchstaben beginnen. Regeln können auch rekursiv definiert werden oder andere Regeln verwenden, wie folgende Regel zeigt, die bestimmt, ob A Großvater von B ist:

```
1 grandfather(A, B) :-
2   father(A, C), parent(C, B).
```

Ein Komma zwischen Regeln entspricht der Konjunktion. Um nun beispielsweise herauszufinden, ob Peter Großvater von Anna ist stellt man die Anfrage `grandfather(peter, anna)` an die Datenbasis. Damit wurden die Variablen A und B schon mit den beiden Personen belegt. Der Wert von C ist jedoch noch offen. Um zu bestimmen, ob `grandfather(peter, anna)` wahr ist, muss mindestens eine Belegung für C gefunden werden, für die sowohl `father(peter, C)` als auch `parent(C, anna)` wahr sind.

Es wird nun deutlich, dass Prolog-Code parallel ausgeführt werden kann. Bei Disjunktionen kann für jede Regel parallel geprüft werden, ob sie unter der konkreten Eingabe wahr ist bzw. ob es Belegungen gibt, die sie wahr machen, falls nicht alle Variablen belegt sind. In diesem Fall sind alle Belegungen gültige Lösungen, die mindestens eine Regel wahr machen. Ähnlich gilt dies für Konjunktionen, jedoch muss hier die Schnittmenge der Lösungsmengen betrachtet werden. Auf obiges Beispiel bezogen: Genau dann, wenn die beiden Ergebnismengen für `father(peter, C)` und `parent(C, anna)` eine nichtleere Schnittmenge besitzen, ist Peter ein Großvater von Anna.

Funktionale und logische Programmiersprachen eignen sich also besonders für *implizite Parallelisierungstechniken*, wohingegen für eine effiziente Parallelisierung von imperativem Code meist die Mithilfe des Programmierers nötig ist.

3. Algorithmen

Parallele Architekturen erfordern, damit sie Berechnungen im Vergleich zu sequentiellen Systemen beschleunigen können, parallele Algorithmen. Im einfachsten Fall handelt es sich dabei um eine nur marginal modifizierte Version eines entsprechenden sequentiellen Algorithmus. Häufig muss für eine effiziente Nutzung der Ressourcen jedoch ein explizit paralleler Algorithmus entwickelt werden. In diesem Abschnitt sollen drei solcher Algorithmen vorgestellt werden.

3.1. Summation

Der intuitive Algorithmus zur Summation einer Liste von Zahlen ist der folgende:

```
1 sum(list): ergebnis {
2   ergebnis = 0;
3   foreach (ele in list)
4     ergebnis += ele;
5 }
```

Dieses Verfahren läuft auf einem SISD-System für eine n -elementige Liste in $O(n)$, kann aber durch einen Parallelrechner nicht beschleunigt werden, da jede Rechnung auf dem Ergebnis der vorhergehenden Rechnung aufbaut.

Auf einem SIMD-System kann der folgende Code effizient umgesetzt werden, falls die Liste die Länge $n = 2^m$ (m ist Ganzzahl) besitzt. Listen, die diese Voraussetzung nicht erfüllen, können durch Anhängen von Nullen auf diese Form gebracht werden.

```
1 sum(list): ergebnis {
2   ergebnis = 0;
3   n = length(list);
4   step = 2;
5
6   while (step <= n) {
7     for (i = 0; i < n; i += step)
8       list[i] += list[i + step/2];
9     step *= 2;
10  }
11 }
```

Die Grundidee dieses Algorithmus ist die Traversierung eines binären Baumes, wobei bei den Blättern begonnen wird. In den Blättern stehen die ursprünglichen Werte. In jedem Durchlauf der `while`-Schleife werden alle Blätterpaare (Blätter, die gemeinsamen Vaterknoten besitzen) addiert

(Zeile 8), das Ergebnis im Vaterknoten gespeichert (ebenda) und die unterste Ebene des Baumes abgeschnitten (Zeile 9). Da die Eingabeliste nicht erhalten bleiben muss werden die Additionsergebnisse an der Speicherstelle des ersten Summanden gespeichert.

Der Algorithmus liefert in $O(\log n)$ Schritten ein Ergebnis, da die for-Schleife bei $n/2$ vorhandenen PEs in konstanter Zeit ausgeführt werden kann und somit für jede Ebene des Baumes nur eine (komplexe) Operation durchgeführt werden muss.

3.2. Quicksort

Quicksort ist ein Beispiel für einen Algorithmus, der sich leicht auf eine parallele Architektur übertragen lässt, da er auf dem „Teile und herrsche“-Prinzip basiert.

Zunächst wird ein Pivot-Element ausgewählt. Alle Elemente der Liste, die kleiner als dieses Element sind, werden an den Anfang der Liste verschoben, alle größeren ans Ende. Nun wird das Pivot-Element zwischen den beiden Teillisten eingefügt. Im nächsten Schritt werden rekursiv die Bereiche vor und nach diesem Element sortiert. Die Rekursion endet, wenn alle Teillisten nur noch höchstens 1 Element besitzen. In diesem Zustand ist die Liste sortiert.

Auf einem MIMD-System mit gemeinsamem Speicher lässt sich dieses Verfahren wie im Folgenden beschrieben implementieren: Ein Prozess beginnt mit der Vorsortierung der Liste nach dem ersten Pivot-Element. Wenn danach die ersten beiden Teillisten vorliegen schreibt er die Begrenzungen einer der Listen in eine Tabelle im globalen Speicher, die andere verarbeitet er rekursiv. Jeder andere Prozess wartet, bis mindestens eine zu sortierende Liste im globalen Speicher vermerkt wurde, entfernt den entsprechenden Eintrag aus der Tabelle und verfährt mit der betreffenden Liste analog zum ersten Prozess. Wenn ein Prozess nur noch eine null- oder einelementige Liste zu sortieren hat, entnimmt er eine neue Aufgabe aus der Tabelle.

Dieser Algorithmus benötigt relativ viel Zeit, bis alle Prozesse arbeiten können, da die ersten und aufgrund der Listenlänge aufwendigsten Schritte nur auf wenige Prozesse verteilt werden können.

3.3. Bitonische Sortierung

Auf SIMD-Systemen kann dank günstiger Kommunikationsoperationen ein effizienterer Algorithmus implementiert werden, der eine Liste in $O(\log n)$ sortiert (zum Vergleich: Quicksort benötigt $O(n \log n)$ Schritte): Die bitonische Sortierung.

Eine *bitonische Folge* ist eine Folge von Zahlen, die entweder bis zu einem Element monoton steigend und ab diesem Element monoton fallend ist oder sich durch eine Rotation zu einer solchen Folge umformen lässt. Hier wird zur

Vereinfachung nur der Fall einer Liste mit $n = 2^m$ Elementen betrachtet (m sei Ganzzahl).

Bitonische Folgen lassen sich sortieren, indem sie rekursiv gemischt werden. Beim Mischen wird die Folge in zwei gleich große Teillisten aufgeteilt und dann jedes Element der ersten Teilliste mit seinem korrespondierenden Element in der zweiten Teilliste getauscht, falls es größer als letzteres ist (für aufsteigende Sortierung). Falls eine absteigende Sortierung gewünscht wird, muss dann getauscht werden, wenn das Element der ersten Liste kleiner ist.

```

1 // Mischen einer 2n-elementigen
2 // bitonischen Folge (aufsteigende
3 // Sortierung)
4 for (j = 0; j < n; j++)
5   if lst[j] > lst[j+n] then
6     swap(lst[j], lst[j+n])

```

Da die einzelnen Durchläufe der Schleife unabhängig voneinander sind können alle Vertauschungen parallel durchgeführt werden, wodurch das Mischen einer einzelnen Folge in konstanter Zeit abläuft. Um eine bitonische Folge zu sortieren muss das Mischen rekursiv auf die Teillisten angewendet werden, bis die betrachtete Folge nur noch zwei Elemente hat und diese sortiert wurden.

Zunächst muss jedoch aus der unsortierten Liste eine bitonische Folge erzeugt werden. Dazu teilt der Algorithmus eine unsortierte Liste in $n/4$ vierelementige Teillisten auf und wandelt jede dieser Listen in bitonische Folgen um:

```

1 for (j = 0; j < n/4; j += 4) {
2   if lst[j] > lst[j+1] then
3     swap(lst[j], lst[j+1]);
4   if lst[j+2] < lst[j+3] then
5     swap(lst[j+2], lst[j+3]);
6 }

```

Dabei entstehen Zahlenpärchen, bei denen abwechselnd die erste und die zweite Zahl die größere ist. Je zwei solcher Pärchen ergeben eine bitonische Folge.

Nun können diese Folgen wie oben beschrieben durch Mischen sortiert werden, wobei dies bei jeder zweiten Folge absteigend geschehen muss. Dadurch entstehen bitonische Listen doppelter Länge. Diese werden wiederum rekursiv gemischt und dadurch sortiert, bis schließlich nur noch eine einzige bitonische Folge existiert, die eine Permutation der zu sortierenden Liste ist. Nun wird diese durch rekursiv gemischt. Nach Abschluss dieser Operationen ist die Sortierung abgeschlossen.

Weitere Details zum den hier vorgestellten Sortierverfahren finden sich in [14].

Fazit

Mit der Nutzung paralleler Systeme geht häufig eine erhöhte Komplexität im Aufbau von Rechnern, bei der Entwicklung von Sprachen und (optimierenden) Compilern und schließlich auch bei der Entwicklung von Algorithmen einher. Bei durchdachtem Einsatz der besprochenen Techniken ist Parallelisierung jedoch in vielen Fällen ein adäquates Mittel, um Anwendungen zu beschleunigen.

Literatur

- [1] Intel 64 and IA-32 architectures software developer's manuals. 2007.
- [2] B. Barney. OpenMP tutorial. 2007.
- [3] T. Bräunl. *Parallele Programmierung. Eine Einführung*, Seite 8. Vieweg, 1993.
- [4] T. Bräunl. *Parallele Programmierung. Eine Einführung*, Seite 156ff. Vieweg, 1993.
- [5] A. R. Fernandes. *GLSL Tutorial*. <http://www.lighthouse3d.com/opengl/glsl/>.
- [6] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21, 1972.
- [7] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
- [8] IEEE and The Open Group. The open group base specifications issue 6. 2004.
- [9] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 1960.
- [10] Message Passing Interface Forum. MPI: A message-passing interface standard. 1995.
- [11] OpenMP Architecture Review Board. OpenMP application program interface. 2005.
- [12] Pande lab, Stanford University. *Folding@home on ATI's GPUs: a major step forward*. <http://folding.stanford.edu/English/FAQ-ATI>.
- [13] G. L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.
- [14] M. J. Quinn. *Algorithmenbau und Parallelcomputer*, Seite 96ff. McGraw-Hill, 1988.
- [15] J. von Neumann. First draft of a report on the EDVAC. 1945.