4 Task- und Datenparallelität

Die Parallelisierung von Anwendungen mithilfe von Threads ist ein mühsames Unterfangen. Man muss viel Zeit investieren, um ein Programm so zu parallelisieren, dass es die verfügbare Rechenleistung effizient ausnutzt. Ein Grund dafür ist, dass die Erzeugung und Verwaltung von Threads einen nicht unerheblichen Aufwand verursacht. Hinzu kommt, dass bei einem Threadwechsel der aktuelle Kontext (Prozessorregister etc.) gesichert und der des auszuführenden Threads wiederhergestellt werden muss. Sind wesentlich mehr Threads rechenbereit als Prozessorkerne vorhanden, kommt es zu einer Überbelegung des Systems, die aufgrund der damit verbundenen Kontextwechsel zu Leistungseinbußen führen kann. Deshalb eignen sich Threads in erster Linie für die Parallelisierung von Anwendungen, die sich in wenige, große Teile zerlegen lassen.

Um das Potenzial von Multicore-Prozessoren ausschöpfen zu können, ist es wünschenswert, Parallelität auf einer feineren Ebene nutzbar zu machen. Zu diesem Zweck bieten aktuelle Programmiersprachen, Bibliotheken bzw. Laufzeitumgebungen zunehmend taskbasierte Programmiermodelle an. Tasks erlauben es, ein Problem in kleine »Häppchen« zu zerlegen und parallel auszuführen. Die Erzeugung eines Tasks ist vergleichsweise einfach und geht je nach Implementierung um bis zu hundertmal schneller vonstatten als die Erzeugung eines Threads. Deshalb sind Tasks auch gut für die Implementierung datenparalleler Algorithmen geeignet, auf die wir in Abschnitt 4.2 eingehen. So lassen sich beispielsweise die Iterationen bestimmter Schleifen in Blöcke zusammenfassen und mithilfe von Tasks parallel ausführen. Zunächst widmen wir uns jedoch den Grundlagen der Taskparallelität.

4.1 Taskparallelität

Ein Task ist durch eine parallel zum Aufrufer ausführbare Einsprungfunktion definiert. In der Regel können dieser Funktion ein oder mehrere Argumente übergeben werden. Darüber hinaus bieten Tasks spezielle Mechanismen zur Synchronisation an. Diese Erweiterungen, zusammen mit einer auf Multicore-Systeme optimierten Implementierung des darunter liegenden Laufzeitsystems, machen Tasks zu einem mächtigen Programmiermodell für parallele Systeme [8, 15].

Ein Vorteil von Tasks ist die bessere Skalierbarkeit: Die meisten mit Tasks parallelisierten Programme lassen sich ohne nennenswerte Anpassungen auf Prozessoren mit unterschiedlich vielen Kernen effizient ausführen, vorausgesetzt dass genügend Parallelität zur Verfügung steht. Skalierbarkeit bedeutet zugleich Zukunftssicherheit. Ein Programm, das für eine bestimmte Prozessorgeneration entwickelt wurde, sollte natürlich auch von zukünftigen Prozessoren profitieren können. Die Programmierung mit Threads ist dagegen meist statisch in dem Sinne, dass eine fest vorgegebene Anzahl von Threads für bestimmte Aufgaben erzeugt wird. Ein Programm, das beispielsweise vier Threads erzeugt, läuft auf einem Prozessor mit acht Kernen bei gleicher Taktfrequenz nicht schneller als auf einem Prozessor mit vier Kernen.

Letztlich werden Tasks üblicherweise wieder auf Threads abgebildet, zumal die gängigen Betriebssysteme keine direkte Unterstützung für Tasks bieten. Die Grundidee dabei ist, für jeden Prozessorkern genau einen Thread anzulegen, der nach Möglichkeit immer auf demselben Kern ausgeführt wird. Auf diese Weise wird die Anzahl der Kontextwechsel auf ein Minimum reduziert. Die Zuweisung von Tasks zu den Threads geschieht zur Laufzeit durch einen sogenannten *Task Scheduler*. Abgesehen von der normalerweise einmaligen Erzeugung und Zerstörung der Threads sind dazu keine Betriebssystemaufrufe notwendig. Abbildung 4.1 zeigt schematisch die Zuordnung von Tasks zu Prozessorkernen.

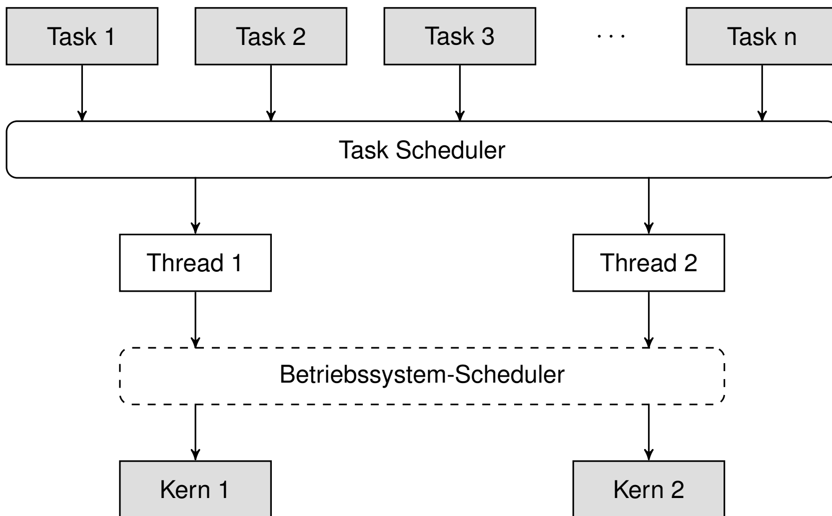


Abbildung 4.1 *Abbildung von Tasks auf die verfügbaren Prozessorkerne*

4.1.1 Erzeugung und Synchronisation von Tasks

Die Erzeugung eines Tasks entspricht dem Einstellen einer Aufgabe in einen Threadpool (siehe Abschnitt 2.1.3). Im Folgenden verwenden wir die Methode `spawn`, um einen Task zu erzeugen. Analog zu `join` bei Threads können wir mittels `wait` auf einzelne Tasks warten. Listing 4.1 zeigt ein einfaches Beispiel für den Umgang mit Tasks. Einige Implementierungen bieten zudem die Möglichkeit, mit nur einem Befehl auf mehrere Tasks zu warten. Wie wir weiter unten sehen werden, kann dies über Aufrufe wie `waitAll` (mit einer Liste von Tasks als Argument), über Eltern-Kind-Beziehungen oder mittels Taskgruppen geschehen. Die jeweilige Umsetzung variiert für verschiedene Sprachen und Bibliotheken. So stellt zum Beispiel Cilk, eine Erweiterung von C, `spawn` und `sync` als eigene Schlüsselwörter zur Verfügung, wobei `sync` auf die zuvor gestarteten Tasks wartet [8].

```
void main() {
  // ... Initialisierung
  // starte einen Task
  Task t = spawn(lambda () {doWork();});
  // ... Hauptprogramm
  // warte auf den Task
  t.wait();
  // ... Aufräumarbeiten
}

doWork() {
  // ... Berechnungen des Tasks
}
```

Listing 4.1 *Arbeiten mit Tasks*

Betrachten wir zum Beispiel die Decodierung von Videoströmen. Ein Strom von Videodaten besteht aus einzelnen Paketen, welche sowohl die Bild- als auch die Toninformationen für einen bestimmten Zeitraum enthalten. Da die Bild- und Toninformationen üblicherweise mit unterschiedlichen Verfahren komprimiert werden, kann die Decodierung parallel erfolgen.

Listing 4.2 zeigt einen vereinfachten Algorithmus für die Decodierung von Videoströmen. Der Algorithmus liest die eingehenden Pakete von einem Strom `e` und schreibt die decodierten Daten in einen Strom `d`. Um die Bild- und Toninformationen parallel decodieren zu können, werden die Methoden `decodeVideo` und `decodeAudio` mittels `spawn` in jeweils einem eigenen Task ausgeführt. Die beiden Aufrufe von `wait` stellen sicher, dass die Daten vor dem Schreiben auf den Ausgabestrom verfügbar sind. Ohne Synchronisation könnte es passieren, dass `v` oder `a` zum Zeitpunkt der Ausführung von `d.write(v, a)` ungültige Daten enthält.

```
EncodedStream e;
DecodedStream d;
// ... Initialisierung
while(!e.end()) {
  Packet p = e.read();
  VideoFrame v;
  AudioFrame a;
  Task tv = spawn(lambda () {v = p.decodeVideo();});
  Task ta = spawn(lambda () {a = p.decodeAudio();});
  tv.wait();
  ta.wait();
  d.write(v, a);
}
```

Listing 4.2 *Parallele Decodierung von Bild- und Toninformationen*

Die parallele Ausführung mehrerer Funktionen bzw. Methoden ist problemlos möglich, wenn diese voneinander unabhängig sind. Vorsicht ist immer dann geboten, wenn sie auf gemeinsame Variablen zugreifen, Ein-/Ausgabeoperationen durchführen oder andere Funktionen (Methoden) aufrufen. In solchen Fällen bedarf die Parallelisierung einer genauen Analyse der Implementierung und ggf. der Verwendung von Synchronisationsoperationen zur Vermeidung von Konflikten. Wir kommen darauf in Abschnitt 4.1.6 zurück. In dem obigen Beispiel muss also sichergestellt sein, dass die Methoden `decodeVideo` und `decodeAudio` nicht ungeschützt auf dieselben Variablen zugreifen. Außerdem dürfen sie keine Seiteneffekte haben, die bei der parallelen Ausführung zu Konflikten führen können.

Tatsächlich ist es nicht notwendig, beide Methoden in jeweils einem eigenen Task auszuführen. Eine parallele Decodierung der Bild- und Toninformationen erreichen wir auch dann, wenn nur für `decodeVideo` ein Task erzeugt wird und `decodeAudio` durch den aktuellen Thread abgearbeitet wird:

```
Task t = spawn(lambda () {v = p.decodeVideo();});
a = p.decodeAudio();
t.wait();
```

Auf diese Weise lässt sich der Mehraufwand für die Taskerzeugung reduzieren, was der Effizienz zugutekommt. Allerdings darf man die beiden Zeilen natürlich nicht vertauschen, da dies einer rein sequenziellen Ausführung gleichkommen würde. Bei den meisten Implementierungen ist der Mehraufwand für die Taskerzeugung jedoch vernachlässigbar. In diesem Buch verwenden wir aus Gründen der Einfachheit und Übersichtlichkeit in der Regel die erste Variante mit jeweils einem `spawn` pro Funktions- bzw. Methodenaufruf.

4.1.2 Parallelisierung rekursiver Algorithmen

Eine nützliche Eigenschaft von Tasks ist die Möglichkeit der verschachtelten Parallelität. Das bedeutet, dass innerhalb eines Tasks weitere Tasks erzeugt werden können. Auf diese Weise lassen sich rekursive Algorithmen nach dem »Teile und herrsche«-Prinzip einfach parallelisieren.

Rekursive Tasks

Beginnen wir mit einem einfachen Beispiel, der Berechnung der Fibonacci-Zahlen, die wie folgt definiert sind:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \text{ für } n \geq 2 \end{aligned}$$

Listing 4.3 zeigt die Implementierung eines parallelen Algorithmus nach der obigen Definition. Dieser Algorithmus ist zwar ziemlich ineffizient aufgrund der mehrfachen Berechnung von Teilergebnissen, aber er eignet sich gut für die Illustration der Parallelisierung rekursiver Algorithmen [15]. Die Grundidee ist wie auch bei dem Beispiel im vorigen Abschnitt, die beiden Funktionsaufrufe parallel auszuführen. Das ist in diesem Fall offensichtlich kein Problem, da die beiden Aufrufe voneinander unabhängig sind und `fibonacci` keine Seiteneffekte hat.

```
int fibonacci(int n) {
  if(n < 2) {
    return n;
  }
  int x, y;
  Task t1 = spawn(lambda () {x = fibonacci(n-1);});
  Task t2 = spawn(lambda () {y = fibonacci(n-2);});
  t1.wait();
  t2.wait();
  return x+y;
}
```

Listing 4.3 *Parallele Berechnung der Fibonacci-Zahlen*

Eltern-Kind-Beziehung

Die verschachtelte Erzeugung von Tasks führt zu einer Eltern-Kind-Beziehung, bei der die neu erzeugten Tasks die Kinder des aufrufenden Tasks sind. Einige Implementierungen nutzen diese »Verwandtschaftsbeziehung« zwischen den Tasks und stellen eine Methode `waitForChildren` (o. Ä.) bereit. Dabei müssen die Tasks, auf die gewartet werden soll, nicht explizit angegeben werden. Wie in Listing 4.4 zu sehen, entfällt damit auch die Notwendigkeit für explizite Taskobjekte. Damit

die Laufzeitumgebung diese erst gar nicht erzeugt, unterscheiden einige Implementierungen beim Erzeugen der Tasks zwischen »normalen« Tasks und Kind-Tasks. In Listing 4.4 deuten wir das mit der Methode `spawnChild` an, die nichts zurückgibt.[1]

```
int fibonacci(int n) {
  if(n < 2) {
    return n;
  }
  int x, y;
  spawnChild(lambda () {x = fibonacci(n-1);});
  spawnChild(lambda () {y = fibonacci(n-2);});
  waitForChildren();
  return x+y;
}
```

Listing 4.4 *Synchronisation von Kind-Tasks*

Abbildung 4.2 zeigt den Aufrufbaum für die Berechnung von `fibonacci(4)`. Jeder Knoten entspricht dabei einem Funktionsaufruf, wobei die Zahlen in den Knoten den Wert des Parameters `n` angeben. Wie aus der Abbildung ersichtlich ist, umfasst der längste Pfad von der Wurzel zu einem Blatt vier Knoten. Das bedeutet, dass das Ergebnis der Berechnung nach vier Schritten vorliegt, sofern genügend Prozessorkerne zur Verfügung stehen. Demgegenüber benötigt die sequenzielle Ausführung neun Schritte.

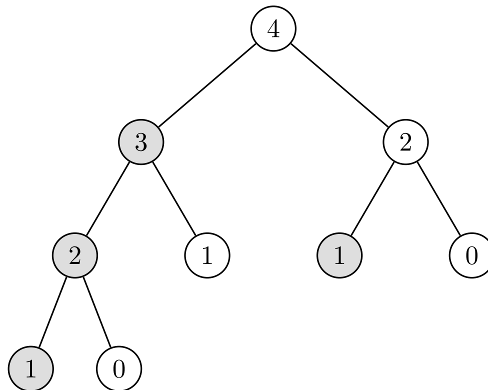


Abbildung 4.2 *Aufrufbaum für die Berechnung von* `fibonacci(4)`

Die Anzahl der erzeugten Tasks lässt sich auch hier wieder reduzieren, indem wir nur für die Berechnung von `fibonacci(n-1)` einen neuen Task erzeugen und

[1] Aus Gründen der Einfachheit gehen wir davon aus, dass es einen impliziten Wurzeltask gibt. In realen Implementierung muss man diesen ggf. explizit erzeugen.

`fibonacci(n-2)` in dem aufrufenden Task berechnen. In diesem Fall wird nur für die grau unterlegten Knoten ein neuer Task erzeugt. Trotz dieser Maßnahme werden für große Werte von `n` sehr viele Tasks erzeugt – in der Regel weit mehr, als Prozessorkerne vorhanden sind. Der Mehraufwand für die Taskerzeugung und -Synchronisation macht den Vorteil der parallelen Ausführung dabei unter Umständen zunichte.

Begrenzen der Rekursionstiefe

Um eine optimale Beschleunigung zu erzielen, genügt es, nur so viele Tasks zu erzeugen, dass alle Prozessorkerne ausgelastet sind. Bei rekursiven Algorithmen ist es deshalb nicht sinnvoll, bis zu den Blättern des Baums immer neue Tasks zu erzeugen. Stattdessen bricht man die parallele Ausführung bei einer gewissen Tiefe ab und berechnet die Teilergebnisse sequenziell. Auf diese Weise wird der Mehraufwand reduziert, ohne die Parallelität über ein sinnvolles Maß hinaus einzuschränken.

Betrachten wir dazu ein realistischeres Beispiel als die rekursive Berechnung der Fibonacci-Zahlen, den Quicksort-Algorithmus [15]. Bei diesem Algorithmus wird das zu sortierende Feld anhand eines Pivotelements zunächst in zwei Teile partitioniert. Ein Teil enthält alle Elemente, die kleiner als das Pivotelement sind, und der andere Teil die Elemente, die größer sind (Elemente, die gleich dem Pivotelement sind, können beliebig verteilt werden). Im zweiten Schritt müssen die beiden Teile sortiert werden. Dazu wird für jeden Teil wieder der Quicksort-Algorithmus aufgerufen. Wenn ein Teil nur noch ein Element enthält, wird die Rekursion abgebrochen.

Listing 4.5 zeigt die parallele Implementierung des Quicksort-Algorithmus (zur Vereinfachung verzichten wir an dieser Stelle auf die Darstellung der Funktion `partition`). Der Algorithmus nimmt eine Referenz auf das Feld sowie den zu sortierenden Bereich in Form eines halboffenen Intervalls `[from, to)` entgegen. Um den Mehraufwand durch die Taskerzeugung zu reduzieren, werden die beiden Aufrufe von `quicksort` nur dann parallel ausgeführt, wenn das zu sortierende Teilfeld mindestens die Größe `k` hat. Es bleibt jedoch die Frage, wie groß `k` in der Praxis sein sollte. Eine Möglichkeit ist, die Größe des Feldes durch die Anzahl der Prozessorkerne zu dividieren. Dabei sollte man eine Reserve einplanen, um Leerlaufzeiten zu vermeiden, die entstehen können, wenn die Tasks unterschiedliche Laufzeiten haben. Bei Quicksort kann es je nach Wahl des Pivotelements nämlich passieren, dass die Teile unterschiedlich groß sind und das Sortieren der Teilfelder somit unterschiedlich viel Zeit in Anspruch nimmt.

Wenn man aus den Parametern der Funktion die Größe der Teilprobleme nicht abschätzen kann, muss man auf andere Heuristiken ausweichen. Ein Ansatz in solchen Fällen besteht darin, die Erzeugung neuer Tasks von der Auslastung des Systems abhängig zu machen. Übersteigt die Anzahl der ausführbereiten Tasks die Anzahl der Prozessorkerne um einen vordefinierten Faktor, wird die

Rekursion abgebrochen. Dieser Ansatz hat den Vorteil, dass er sich dynamisch an das Problem anpasst. Allerdings muss man dazu wissen, wie viele Tasks ausführbereit sind.

```
void quicksort(Type x[], int from, int to) {
  int size = to-from;
  if(size <= 1) {
    return;
  }
  int pivot = partition(x, from, to);
  if(size >= k) {
    spawnChild(lambda () {quicksort(x, from, pivot);});
    spawnChild(lambda () {quicksort(x, pivot+1, to);});
    waitForChildren();
  }
  else {
    quicksort(x, from, pivot);
    quicksort(x, pivot+1, to);
  }
}
```

Listing 4.5 *Parallele Implementierung des Quicksort-Algorithmus*

Unsere Implementierung des Quicksort-Algorithms hat leider noch einen »Schönheitsfehler«: Wir haben nur die rekursive Zerlegung parallelisiert, die Partitionierung der Teilfelder aber nicht weiter betrachtet. Um eine möglichst optimale Beschleunigung zu erreichen, müssen wir dafür Sorge tragen, dass auch die Partitionierung parallel erfolgt. Weiterführende Informationen zu parallelen Sortierverfahren sind in [15, 42] zu finden.

Mehrdimensionale Datenstrukturen

Die Nutzung von Parallelität durch rekursive Zerlegung ist natürlich nicht auf eindimensionale Datenstrukturen beschränkt. Betrachten wir dazu noch ein Beispiel, die Matrizenmultiplikation. Das Produkt C zweier $n \times n$-Matrizen A und B ist wie folgt definiert, wobei c_{ij} das Element in Zeile i und Spalte j der Matrix C angibt (Gleiches gilt für A und B):

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

Eine Methode zur Parallelisierung der Matrizenmultiplikation besteht darin, das Problem in mehrere Teilprobleme zu zerlegen. Dazu werden die Matrizen in gleich große Blöcke unterteilt (n muss eine gerade Zahl sein) [15]:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$
$$= \begin{pmatrix} A_{00} \cdot B_{00} + A_{01} \cdot B_{10} & A_{00} \cdot B_{01} + A_{01} \cdot B_{11} \\ A_{10} \cdot B_{00} + A_{11} \cdot B_{10} & A_{10} \cdot B_{01} + A_{11} \cdot B_{11} \end{pmatrix}$$

Die Berechnung der Teilprodukte lässt sich nach demselben Schema fortsetzen, sodass wir es mit einem klassischen »Teile und herrsche«-Ansatz zu tun haben. In jedem Task erzeugen wir dazu acht Kind-Tasks, bis die Teilmatrizen klein genug sind. Diese können dann gemäß der oben angegebenen Definition sequenziell multipliziert werden. Als Alternative zur rekursiven Zerlegung kann man auch die Schleifen für die Berechnung der Elemente c_{ij} parallelisieren (siehe Abschnitt 4.2).

4.1.3 Taskgruppen

Angenommen wir haben einen irregulären[2] Baum und möchten für jeden Knoten eine Methode `execute` aufrufen. Im sequenziellen Fall durchlaufen wir dazu den Baum rekursiv von der Wurzel zu den Blättern und rufen für jeden besuchten Knoten die Methode `execute` auf (Listing 4.6).

```
class Tree {
private:
  Node root;
  void traverse(Node node) {
    foreach(Node successor in node.getSuccessors()) {
      traverse(successor);
    }
    node.execute();
  }
public:
  void process() {
    traverse(root);
  }
}
```

Listing 4.6 *Sequenzieller Baumdurchlauf*

In einer parallelen Variante möchten wir nun für jeden Aufruf von `traverse` der Klasse `tree` einen eigenen Task starten, um größtmögliche Parallelität zu erreichen. Das Problem ist nur, dass die Methode `process` sehr schnell zurückkehrt, nämlich wenn alle Tasks gestartet sind. Wir interessieren uns aber für den Zeitpunkt, zu dem alle Tasks ihre Arbeit erledigt haben. Mit den bereits bekannten Konstrukten haben wir zwei Möglichkeiten: Entweder legen wir alle Taskobjekte in einer Liste ab und warten auf diese mittels `waitAll`, oder wir nutzen

[2]Ein Baum heißt irregulär, wenn die Anzahl der Kinder eines Knotens nicht fest ist.

die Eltern-Kind-Beziehung der Tasks und fügen spätestens nach `node.execute()` einen Aufruf von `waitForChildren` ein. Beides führt aber dazu, dass viele Taskobjekte unnötig lange am Leben erhalten werden. Für unser Beispiel genügt es zu wissen, wann *alle* Tasks beendet sind.

Für solche Zwecke bieten einige Sprachen bzw. Bibliotheken *Taskgruppen* als Hilfsmittel an. Im Folgenden beschreiben wir Taskgruppen durch Instanzen einer gegebenen Klasse `TaskGroup`. Diese Klasse verfügt analog zu `spawn` und `wait` für Tasks über gleichnamige Methoden. Die Methode `spawn` nimmt eine Lambda-Funktion entgegen und führt diese in einem neuen Task aus. Der Task wird dabei automatisch der entsprechenden Gruppe hinzugefügt. Die Methode `wait` wartet, bis alle Tasks der Gruppe fertig sind.

Listing 4.7 skizziert die Implementierung des parallelen Baumdurchlaufs. Die Methode `traverse` erzeugt für jeden Nachfolgerknoten einen Task und fügt diesen der Gruppe `group` hinzu. Da `traverse` unmittelbar nach der Ausführung von `execute` zurückkehrt, kann es passieren, dass zu diesem Zeitpunkt noch Teilbäume in Bearbeitung sind. Das ist jedoch kein Problem, da durch den Aufruf von `group.wait()` in der Methode `process` das Ende aller Tasks abgewartet wird.

```
class Tree {
private:
  TaskGroup group;
  Node root;
  void traverse(Node node) {
    foreach(Node successor in node.getSuccessors()) {
      group.spawn(lambda () {traverse(successor);});
    }
    node.execute();
  }
public:
  void process() {
    traverse(root);
    group.wait();
  }
}
```

Listing 4.7 *Paralleler Baumdurchlauf mit Taskgruppen*

Intern besteht `TaskGroup` im Wesentlichen aus einem Zähler, der beim Start eines Tasks in dieser Gruppe inkrementiert und am Ende des Tasks dekrementiert wird. Dadurch kann der Task Scheduler Verwaltungsinformationen von beendeten Tasks freigeben. Auf diese Weise lassen sich sehr große Bäume parallel durchlaufen, wohingegen die Synchronisation mittels Eltern-Kind-Beziehung schnell zu einem Stack-Überlauf führen kann, da die Verwaltungsinformationen eines Tasks so lange erhalten bleiben, bis deren Kinder ihre Arbeit erledigt haben. Auch Tasklisten würden eine immense Größe erreichen.

Die Verwendung einer »globalen« Taskgruppe funktioniert jedoch nur dann, wenn die Tasks voneinander unabhängig sind. Während wir also den Quicksort-Algorithmus aus Abschnitt 4.1.2 auf die gleiche Weise parallelisieren können wie den Baumdurchlauf aus Listing 4.7, ist dies bei der Berechnung der Fibonacci-Zahlen so nicht möglich. Der Grund dafür ist, dass das Ergebnis eines Tasks von den Rückgabewerten der Kind-Tasks abhängt und somit eine Synchronisation erforderlich ist. Dennoch lassen sich Eltern-Kind-Beziehungen leicht mit Taskgruppen nachbilden. Dazu legt man für jede »Generation« eine neue Gruppe an. Der folgende Ausschnitt verdeutlicht dies am Beispiel der Berechnung der Fibonacci-Zahlen (vgl. Listing 4.3):

```
TaskGroup group;
group.spawn(lambda () {x = fibonacci(n-1);});
group.spawn(lambda () {y = fibonacci(n-2);});
group.wait();
```

4.1.4 Spekulation

Die parallele Ausführung von Programmteilen ist in der Regel nur dann möglich, wenn zwischen den Programmteilen keine Abhängigkeiten bestehen. Dies betrifft sowohl Datenabhängigkeiten als auch Abhängigkeiten, die durch den Programmablauf (Kontrollfluss) gegeben sind. Manchmal lässt sich jedoch trotz Abhängigkeiten eine parallele Ausführung der Programmteile erreichen. Betrachten wir dazu das Beispiel in Listing 4.8. Obwohl es auf den ersten Blick unmöglich erscheint, die Funktionen `condition`, `fun1`, `fun2` und `use` parallel auszuführen, können wir eine Beschleunigung erreichen, indem wir beide Pfade der Verzweigung *spekulativ* ausführen. Sobald das Ergebnis des Aufrufs von `condition` vorliegt und beide Zweige abgearbeitet sind, wählen wir den entsprechenden Wert aus und übergeben ihn der Funktion `use`. Diese Form der Spekulation ist natürlich nur dann sinnvoll, wenn die Berechnung der Bedingung nicht trivial ist. Außerdem dürfen `fun1` und `fun2` keine Seiteneffekte haben.

```
Type x;
if(condition()) {
  x = fun1();
} else {
  x = fun2();
}
use(x);
```

Listing 4.8 *Bedingte Ausführung zweier Funktionen*

Listing 4.9 zeigt die Implementierung. Für die Funktionen `fun1` und `fun2` wird jeweils ein Kind-Task erzeugt und parallel zu `condition` ausgeführt. Durch den

Aufruf von `waitForChildren` ist sichergestellt, dass `x1` und `x2` vor der Ausführung der Verzweigung gültige Werte enthalten.

```
Type x1, x2;
spawnChild(lambda () {x1 = fun1();});
spawnChild(lambda () {x2 = fun2();});
bool cond = condition();
waitForChildren();
if(cond) {
  use(x1);
} else {
  use(x2);
}
```

Listing 4.9 *Spekulative Ausführung des Programms aus Listing 4.8*

Die spekulative Ausführung geht immer mit unnötigen Berechnungen einher, da nur die Ergebnisse *eines* Zweigs tatsächlich verwendet werden (die des jeweils anderen Zweigs werden verworfen). Der damit verbundene Mehraufwand kann bei einem voll ausgelasteten System sogar zu einer Verlangsamung führen, da wertvolle Rechenzeit, die für andere Aufgaben verwendet werden könnte, verschwendet wird. Aus diesem Grund sollte man vor dem Einsatz von Spekulation Aufwand und Nutzen sorgfältig abwägen.

Betrachten wir noch einmal das obige Beispiel. Legt man die in Tabelle 4.1 gezeigten Ausführungszeiten zugrunde, benötigt die sequenzielle Implementierung (Listing 4.8) 11 Zeiteinheiten, wenn die Bedingung erfüllt ist, und 9 Zeiteinheiten, wenn die Bedingung nicht erfüllt ist (in beiden Fällen ist die Rechenzeit gleich der Ausführungszeit). Die Ausführungszeit der spekulativen Variante (Listing 4.9) hängt von der Verteilung der Tasks auf die verfügbaren Prozessorkerne ab. Abbildung 4.3 zeigt einen möglichen Ablauf.[3] Die Ausführungszeit beträgt 7 Zeiteinheiten, unabhängig davon, ob die Bedingung erfüllt ist oder nicht (die Rechenzeit erhöht sich auf 14 Zeiteinheiten).

Funktion	Ausführungszeit
`condition`	4 Zeiteinheiten
`fun1`	5 Zeiteinheiten
`fun2`	3 Zeiteinheiten
`use`	2 Zeiteinheiten

Tabelle 4.1 *Ausführungszeiten der Funktionen aus dem Programm in Listing 4.8*

[3] Die weißen Kästchen sind alternativ, d. h., es wird entweder `use(x1)` oder `use(x2)` ausgeführt.

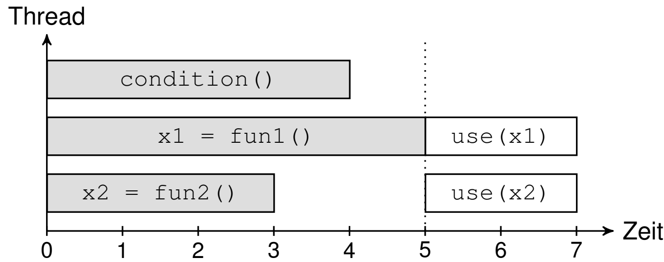


Abbildung 4.3 *Ablauf für das Programm aus Listing 4.9*

Wie in Abbildung 4.3 zu sehen, ist unsere Implementierung noch nicht optimal, da die Funktion `use` erst dann aufgerufen wird, wenn sowohl `fun1` als auch `fun2` ausgeführt wurden. Wenn die Bedingung falsch ist, besteht jedoch keine Notwendigkeit, auf das Ergebnis von `fun1` zu warten. Listing 4.10 zeigt eine bessere Implementierung. Die Idee bei dieser Implementierung besteht darin, in Abhängigkeit der Bedingung entweder nur auf `fun1` oder nur auf `fun2` zu warten.

```
Type x1, x2;
Task t1 = spawn(lambda () {x1 = fun1();});
Task t2 = spawn(lambda () {x2 = fun2();});
if(condition()) {
  t1.wait();
  use(x1);
} else {
  t2.wait();
  use(x2);
}
```

Listing 4.10 *Spekulative Ausführung mit bedingtem Warten*

Der Ablauf in Abbildung 4.4 veranschaulicht die spekulative Ausführung mit bedingtem Warten. Sobald der Wert der Bedingung feststeht, kann mit der Ausführung der Funktion `use` begonnen werden. Ist die Bedingung falsch, reduziert sich in unserem Beispiel die Ausführungszeit von sieben auf sechs Zeiteinheiten.

Nicht immer ist es sinnvoll, beide Pfade einer Verzweigung spekulativ auszuführen. Wenn ein Pfad mit einer hohen Wahrscheinlichkeit eingeschlagen wird, bietet es sich an, nur diesen spekulativ auszuführen und den anderen Pfad wie im sequenziellen Fall abzuarbeiten. Auf diese Weise wird im Mittel eine gute Beschleunigung erreicht und der Mehraufwand reduziert. Ein typisches Anwendungsszenario dieser asymmetrischen Form der Spekulation ist die Behandlung von Sonderfällen. Wenn nur sehr selten ein Sonderfall eintritt, lohnt es sich nicht, den entsprechenden Pfad im Voraus auszuführen. Listing 4.11 demonstriert die Vorgehensweise unter der Annahme, dass die Bedingung in der Regel erfüllt ist.

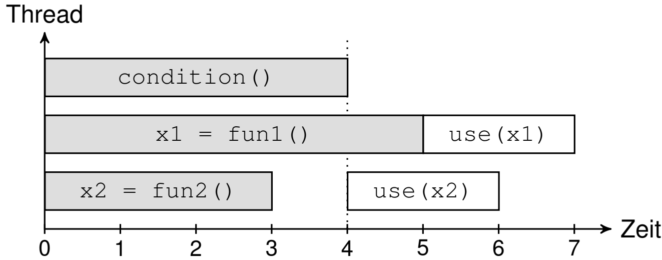


Abbildung 4.4 *Ablauf für das Programm aus Listing 4.10*

```
Type x;
Task t = spawn(lambda () {x = fun1();});
if(condition()) { // häufiger Fall
  t.wait();
  use(x);
} else { // seltener Fall
  use(fun2());
}
```

Listing 4.11 *Asymmetrische spekulative Ausführung*

Abbildung 4.5 zeigt einen möglichen Ablauf des Programm aus Listing 4.11. Im häufigen Fall profitieren wir von einer Reduzierung der Ausführungszeit um vier Zeiteinheiten, ohne den Mehraufwand von fünf Zeiteinheiten für die spekulative Ausführung des anderen Zweigs in Kauf nehmen zu müssen. Falls die Bedingung doch einmal falsch ist, liegt das Ergebnis wie bei der sequenziellen Ausführung nach neun Zeiteinheiten vor.

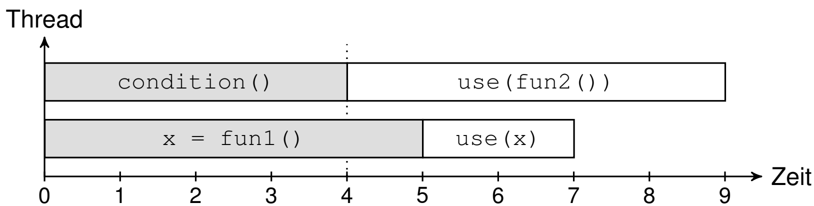


Abbildung 4.5 *Ablauf für das Programm aus Listing 4.11*

4.1.5 Implementierung eines Task-Schedulers

Für viele Programmiersprachen gibt es Bibliotheken bzw. Spracherweiterungen, mit deren Hilfe sich leicht taskbasierte Programme schreiben lassen (im zweiten Teil dieses Buches werden wir einige davon kennenlernen). Als Entwickler oder Architekt paralleler Software muss man sich deshalb normalerweise nicht mit den

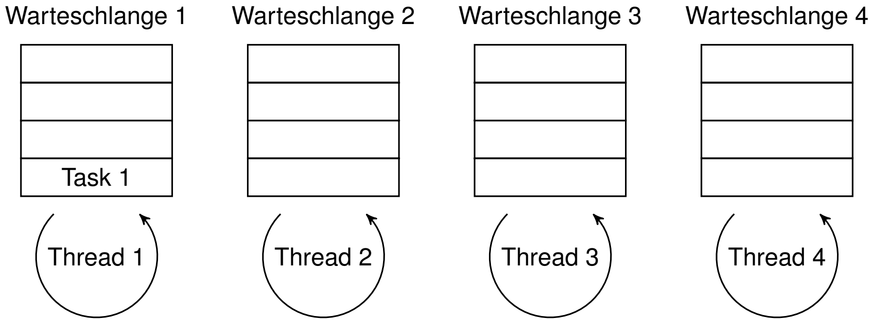
Interna der Taskverwaltung und -Ausführung auseinandersetzen. Dennoch ist es hilfreich, die Grundprinzipien zu kennen, um typische Fallstricke zu vermeiden. Nachdem wir in den vorherigen Abschnitten Tasks aus Benutzersicht betrachtet haben, werfen wir im Folgenden einen Blick hinter die Kulissen. Im Anschluss daran zeigen wir einige Richtlinien auf, die bei der Entwicklung von taskbasierten Programmen zu beachten sind.

Die Abarbeitung von Tasks kann mit Threadpools organisiert werden. Wie wir in Abschnitt 2.1.3 gesehen haben, arbeiten Threadpools normalerweise mit einer globalen Warteschlange, in die die auszuführenden Tasks eingereiht werden. Sobald ein Thread einen Task abgearbeitet hat, entnimmt er der Warteschlange einen neuen Task und führt diesen aus. Mit einer steigenden Anzahl von Prozessorkernen entwickelt sich die globale Warteschlange jedoch schnell zum Flaschenhals. Insbesondere bei kleinen Tasks kann der Aufwand für die Synchronisation beim Zugriff auf die globale Warteschlange erheblich zu Buche schlagen. Außerdem steht der Einsatz einer globalen Warteschlange dem Bestreben nach Lokalität entgegen, da bei jedem Zugriff auf die Warteschlange Änderungen in den Hauptspeicher geschrieben bzw. von diesem geladen werden müssen.

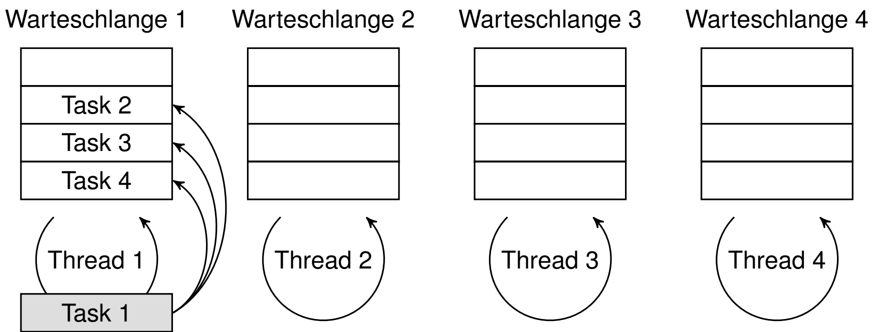
Eine Lösung dieses Problems ist, jedem Thread eine eigene, lokale Warteschlange zuzuteilen. Alle in einem Thread erzeugten Tasks landen zunächst in der lokalen Warteschlange und werden in der Regel von genau diesem Thread abgearbeitet. Auf diese Weise wird der Flaschenhals beim Zugriff auf eine gemeinsame Warteschlange entschärft. Außerdem fördert dies die Datenlokalität: Tasks, die mit hoher Wahrscheinlichkeit auf dieselben Daten zugreifen, werden auf demselben Prozessorkern ausgeführt. Die lokalen Warteschlangen funktionieren üblicherweise nach dem LIFO-Prinzip (engl. *last in, first out*). Das bedeutet, dass der zuletzt eingereihte Task zuerst bearbeitet wird. Das Ziel dabei ist, die Caches »heiß« zu halten, also zu verhindern, dass mit jedem Task, der der lokalen Warteschlange entnommen wird, der Cache neu geladen werden muss.

Es fehlt nur noch ein Mechanismus, um die Last auf die Prozessorkerne zu verteilen. Hier kommt das sogenannte *Work Stealing* zum Tragen [9]. Wenn die Warteschlange eines Threads leer ist, »stiehlt« er einen Task aus der Warteschlange eines anderen Threads. Die Auswahl des »Opfers« erfolgt üblicherweise nach dem Zufallsprinzip, um eine gleichmäßige Auslastung zu erreichen. Das eigentliche Stehlen geht dabei nach dem FIFO-Prinzip (engl. *first in, first out*) vonstatten. Es wird also der Task entnommen, der am längsten auf seine Ausführung wartet. Zu diesem Zweck bietet sich die Verwendung von Double-Ended Queues an, für die effiziente, nichtblockierende Implementierungen existieren [27]. Auf das eine Ende einer Double-Ended Queue wird nur durch den dazugehörigen Thread zugegriffen, während das andere Ende für das Work Stealing benutzt wird.

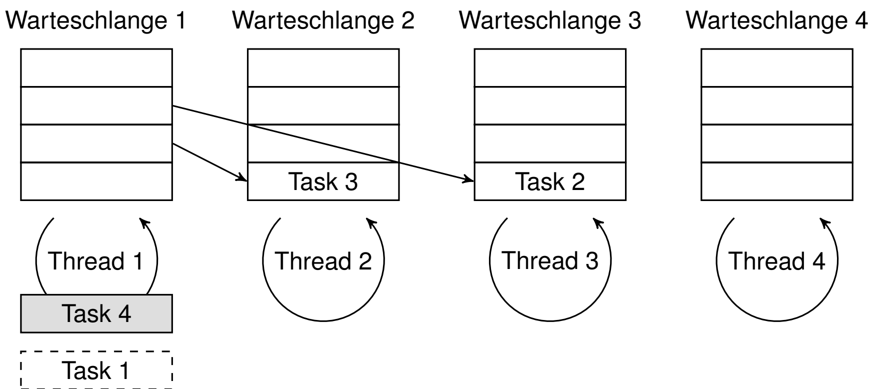
Abbildung 4.6 illustriert die Vorgehensweise beim Work Stealing. Der obere Teil (Abb. 4.6 (a)) zeigt die Ausgangssituation, in der nur ein Task (Task 1) zur Abarbeitung ansteht. Dieser Task wird von Thread 1 der lokalen Warteschlange entnommen und ausgeführt. Wie in Abb. 4.6 (b) zu sehen ist, erzeugt Task 1 drei



(a) Warteschlange 1 enthält einen Task (Task 1)



(b) Thread 1 führt Task 1 aus, der drei neue Tasks erzeugt (Tasks 2 – 4)



(c) Thread 1 führt Task 4 aus, Tasks 2 und 3 werden gestohlen

Abbildung 4.6 *Work Stealing auf einem System mit vier Prozessorkernen*

neue Tasks (Tasks 2 bis 4), die nacheinander in die lokale Warteschlange eingereiht werden. Anschließend wartet er, bis diese Tasks fertig sind. In Abb. 4.6 (c) stehlen die Threads 3 und 2 die beiden ältesten Tasks aus Warteschlange 1. Da Thread 1 nun auch wieder frei ist, beginnt er mit der Ausführung von Task 4.

Es gibt zahlreiche Varianten und Erweiterungen dieses Verfahrens. Beispielsweise kann man zusätzlich zu den lokalen Warteschlangen eine globale Warteschlange einsetzen, in die Tasks durch den Thread des Hauptprogramms oder durch andere Threads eingereiht werden. Die Lastverteilung geschieht dabei durch eine Kombination aus Threadpool und Work Stealing. Wenn eine Warteschlange leer ist, wird ein Task entweder der globalen Warteschlange entnommen oder von einer anderen Warteschlange gestohlen.

4.1.6 Programmierrichtlinien

Obwohl sich Threads und Tasks aus Benutzersicht sehr ähnlich sind, gibt es einige grundlegende Unterschiede. Ein Unterschied ist, dass Tasks aus logischer Sicht nicht preemptiv sind, da dies einen Kontextwechsel nach sich ziehen würde. Das bedeutet, dass ein Task nicht durch einen anderen Task verdrängt werden kann. Tasks werden nur dann unterbrochen, wenn sie auf andere Tasks warten müssen (z. B. durch `wait`, `waitAll`, `waitForChildren`). Die zu einem Task gehörenden Daten bleiben dabei auf dem Stack gespeichert, sodass nach Beendigung der Tasks, auf die gewartet wird, mit den ursprünglichen Daten weitergearbeitet werden kann. Während der Unterbrechung muss der bearbeitende Thread natürlich nicht »Däumchen drehen«, sondern kann solange andere Tasks ausführen.

Eine Konsequenz der Ununterbrechbarkeit von Tasks ist, dass Synchronisationsoperationen wie Mutexe nur mit Bedacht eingesetzt werden sollten. Versucht ein Task beispielsweise, einen bereits gesperrten Mutex zu sperren, wird der entsprechende Thread vom Betriebssystem schlafen gelegt und kann keine anderen Tasks abarbeiten. Als Folge davon kann eine Unterbelegung eintreten, wodurch die Leistung sinkt.[4] Deshalb sollten die Teile einer Anwendung, die häufig blockieren, in eigenen Threads ablaufen. Dazu gehören neben Ein-/Ausgabeoperationen auch solche Teile, die mittels Nachrichtenaustausch kommunizieren und nur sporadisch aktiv sind. In solchen Fällen ist eine Überbelegung eher zu tolerieren als eine Unterbelegung durch schlafende Threads.

Ein weiterer Unterschied zwischen Threads und Tasks betrifft die Zuteilung von Rechenzeit. Bei Threads garantiert das Betriebssystem eine faire Abarbeitung nach dem Zeitscheibenverfahren auch im Falle einer Überbelegung. Das bedeutet, dass jedem rechenbereiten Thread in Abhängigkeit seiner Priorität ein gewisser Teil der zur Verfügung stehenden Rechenzeit zugeteilt wird. Folglich kann es nicht passieren, dass ein Thread »verhungert«. Im Gegensatz dazu sind Task Scheduler in der Regel nicht fair. Wie wir im vorherigen Abschnitt gesehen haben,

[4] Manche Task Scheduler starten in solchen Fällen automatisch zusätzliche Threads (z. B. der .NET-Task-Scheduler, siehe Kapitel 12).

werden Tasks nach dem LIFO-Prinzip abgearbeitet und nach dem FIFO-Prinzip gestohlen. Die Reihenfolge der Ausführung hängt also vom Zeitpunkt der Erzeugung eines Tasks und der Auslastung des Gesamtsystems ab, wobei früh erzeugte Tasks oft erst sehr spät ausgeführt werden. Bei anhaltender Auslastung des Systems kann es sogar vorkommen, dass ein Task gar nicht ausgeführt wird. Kurz gesagt wird die Effizienz von Task-Schedulern durch den Verlust von Fairness erkauft. Dies kann beispielsweise bei Erzeuger-Verbraucher-Problemen dazu führen, dass unentwegt Elemente erzeugt, aber nicht verbraucht werden.

Hinweise

- ➔ Tasks sind für rechenintensive Aufgaben gedacht. Lagern Sie Aufgaben, die mit häufigem Warten verbunden sind, in eigene Threads aus.
- ➔ Vermeiden Sie blockierende Synchronisationsmechanismen und synchronisieren Sie Tasks stattdessen mittels `wait` etc.
- ➔ Alternativ bieten sich atomare Operationen und Spinlocks an. Letztere sind jedoch nur dann sinnvoll, wenn die voraussichtliche Wartezeit kurz ist.
- ➔ Es besteht keine Garantie auf eine faire Abarbeitung der Tasks.

Abschließend noch ein Wort zum Test taskbasierter Programme: Da sich erst zur Laufzeit entscheidet, ob zwei Tasks tatsächlich parallel ausgeführt werden, kann ein mit Tasks parallelisiertes Programm problemlos auch auf einem Prozessor mit nur einem Kern ausgeführt werden, meist ohne nennenswerten Geschwindigkeitsverlust im Vergleich zu einer sequenziellen Implementierung. Allerdings treten wie bei der Programmierung mit Threads manche Fehler erst dann zutage, wenn das Programm auf einem parallelen System ausgeführt wird. Deshalb gilt auch für die Programmierung mit Tasks der Grundsatz, dass man zum Testen einen Multicore-Rechner einsetzen sollte.

4.2 Datenparallelität

Während sich Taskparallelität um die parallele Abarbeitung mehrerer Funktionen dreht, stehen bei der Datenparallelität die zu verarbeitenden Daten im Vordergrund. Charakteristisch für Datenparallelität ist die parallele Ausführung eines sequenziellen Befehlsstroms auf einer Menge gleichartiger Daten. Datenparallele Anwendungen folgen somit dem SIMD-Prinzip und eignen sich gut für die Verarbeitung regulärer Datenstrukturen in Form von Schleifen.

Einige der im zweiten Teil dieses Buches vorgestellten Sprachen und Bibliotheken stellen Konstrukte für die parallele Ausführung von Schleifen zur Verfügung. In C# und TBB gibt es zum Beispiel vorgefertigte »Skelette« für parallele Schleifen, die man als Entwickler nur noch mit »Fleisch« füllen muss. Im Folgenden