

2

Allgemeine Konzepte

Im ersten Kapitel ging es in erster Linie darum, theoretische Konzepte und die Zielsetzung von Parallel Computing zu erläutern. Basierend auf diesem Wissen wird nun untersucht und beschrieben, welche Richtlinien bei der Umsetzung eines guten parallelen Systems beachtet werden sollten. Spezielle Probleme, die im Zusammenhang mit Parallel Computing auftreten, werden erläutert. Des Weiteren werden Architektur und Umsetzungskriterien erarbeitet, die bei einer späteren parallelen Implementierung von Bedeutung sind. Dabei geht es in erster Linie darum, allgemeine Grundüberlegungen anzustellen, um später eine erfolgreiche Implementierung zu gewährleisten. In diesem Kapitel werden daher noch keine konkreten Implementierungen durchgeführt. Basierend auf den theoretischen Grundlagen wird ab Kapitel 3 mit den praktischen Implementierungen begonnen.

2.1 Regeln für erfolgreiches Parallel Computing

In Kapitel 1.5, „Performance Indikatoren und Gesetzmäßigkeiten“, wurden bereits wichtige Kennzahlen und Berechnungsmodelle beschrieben, um eine Erfolgsmessung der parallelen Lösung durchzuführen. Wichtige Faktoren sind demnach:

- Speedup
- Effizienz
- Laufzeitverhalten
- Skalierbarkeit

Im optimalen Fall muss der Speedup durch das Hinzufügen von CPU-Kernen steigen, die Effizienz erhöht werden und das Laufzeitverhalten der Anwendung muss sich verbessern. Das zu erreichen, erfordert eine gute Planung und eine fehlerfreie Architektur der Anwendung. Zudem soll eine gute Skalierbarkeit gewährleistet werden, d. h. stehen mehr CPU-Kerne zur Verfügung, müssen diese auch automatisch von der Anwendung verwendet werden. Dabei ist zu beachten, dass kein Slowdown-Effekt (Kap.1.5.6) eintritt. Bevor damit begonnen wird, eine parallele Lösung zu entwerfen, sollte untersucht werden, ob sich der Aufwand für das spezielle Szenario lohnt. Es gibt einige Einsatzfelder, in denen Parallel Computing sehr hilfreich und angebracht ist, demgegenüber existieren aber auch einige Bereiche, die für Parallel Computing nur bedingt geeignet sind. Geeignete Bereiche sind z. B.:

- Bildmanipulationen
- Aufwendige Berechnungen (z. B. Wetterprognose, Data Mining)
- Künstliche Intelligenz (AI)
- Simulationen (z. B. Finanz- und Geschäftssimulation)
- Webroboter (Indizierung des Internets)

Es kann nicht generell festgelegt werden, dass alle mathematischen Probleme ohne Einschränkung parallelisiert werden können. Jeder Problemfall muss auf die Möglichkeiten der Parallelisierung hin bewertet werden. Folgendes mathematisches Problem soll das demonstrieren:

$$F(k+2) = F(k+1) + F(k)$$

Die obere Formel beschreibt die Regel zur Erzeugung einer Fibonacci-Reihe. Eine Fibonacci-Zahl wird bestimmt, indem die Summe der beiden vorherigen Zahlen addiert wird. Dieses Problem ist nicht parallelisierbar, da zum Fortschritt jeweils die vorherigen Rechenergebnisse benötigt werden.

PROFITIPP: Je weniger Abhängigkeiten zwischen den zu lösenden Teilaufgaben bestehen, umso höher ist die Möglichkeit der Parallelisierung.

HINWEIS: Werden Problemfälle durch den Einsatz von Parallel Computing gelöst, die eigentlich nicht dafür geeignet sind, verschlechtert sich wahrscheinlich die Laufzeit gegenüber der reinen sequenziellen Lösung. Im schlimmsten Fall führt die parallele Umsetzung zu falschen Ergebnissen. Das ist oft auf fehlende oder falsche Sperren auf gemeinsame Objekte zurückzuführen.

Als Beispiel für ein gut parallelisierbares Problem kann die Manipulation von Bildpunkten eines Bilds genannt werden. Soll z. B. ein Farbfilter auf ein Bild angewendet werden, so kann jeder Bildpunkt unabhängig voneinander berechnet werden.

Wurde ein Problemfall als gut parallelisierbar befunden, muss eine Architektur entworfen werden, um die wesentlichen Kriterien einer parallelen Anwendung zu genügen. Dazu müssen einige Punkte beachtet werden, die im Folgenden erläutert werden.

2.1.1 Arbeitsverteilung

Um die zu parallelisierende Gesamtarbeit auf mehrere Prozesse verteilen zu können, muss die Arbeit in kleinere Arbeitspakete geteilt werden. Dazu wird die ermittelte Arbeitsmenge, die parallelisierbar ist, in einzelne Arbeitsaufgaben zerlegt. Hierbei können drei Arten der Aufteilung unterschieden werden:

- Domänenzerlegung (Domain Decomposition)
- Funktionale Zerlegung (Functional Decomposition)
- Master-Slave-Modell (Master-Slave Model)

Diese drei Arten werden im Folgenden etwas näher erläutert.

Domänenzerlegung (Domain Decomposition)

Bei dieser Art der Aufteilung stehen die Daten im Mittelpunkt. Die zu verarbeitenden Daten werden in einzelne Datenpakete geteilt, die nach Möglichkeit getrennt voneinander berechnet werden können. Abbildung 2.1 zeigt, wie ein komplettes Datenpaket in mehrere Teilpakete aufgeteilt werden kann.

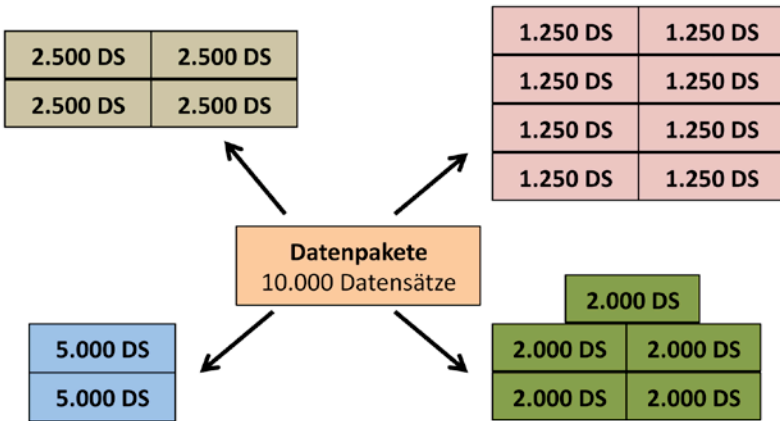


Abbildung 2.1: Aufteilung von Arbeitspaketen

Wie zu erkennen ist, ergeben sich verschiedene Möglichkeiten der Paketaufteilung. Auch wenn die Aufteilung trivial und einfach erscheint, können hier sehr viele Fehler gemacht werden. Wichtig ist, eine optimale Aufteilung der Datenpakete zu erzielen. Geschieht das nicht, werden schon bei dem Design der Anwendung so genannte „kritische Bereiche“ (Kap. 1.5.6) erzeugt, die unter Umständen zu einem Slowdown (Kap. 1.5.7) der Anwendung führen können. Optimal heißt in diesen Fall:

- Einzelne Datenpakete sollten gleich groß sein
- Anzahl der Arbeitspakete gemäß verfügbaren Prozessoren

Wenn die Aufgaben auf den verfügbaren Kernen eines Prozessors verteilt werden, sollten die Arbeitsbelastungen pro Prozessorkern gleich

sein. Ansonsten kann es schnell dazu führen, dass einige Prozessorkerne zu viel Arbeit und einige zu wenig Arbeit haben. Abbildung 2.2 verdeutlicht den Effekt.

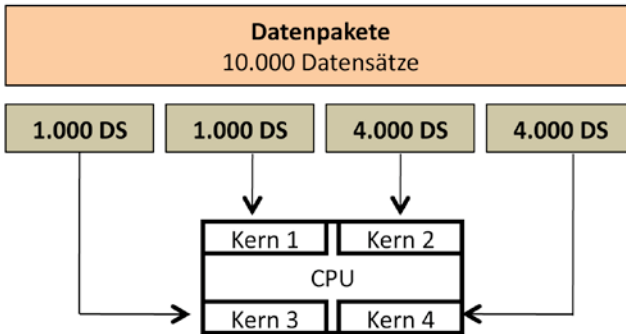


Abbildung 2.2: Falsche Aufteilung der Datenpakete

Da die Arbeitspakete nicht gleich groß sind, ist zunächst Kern 1 und kurze Zeit später Kern 3 ohne Arbeit. Demgegenüber sind die Kerne 2 und 4 voll ausgelastet. Durch eine bessere Aufteilung der Arbeitslast, kann eine bessere Lastverteilung erfolgen. Abbildung 2.3 zeigt das Ergebnis nach einer erneuten Neuauflteilung der Arbeit.

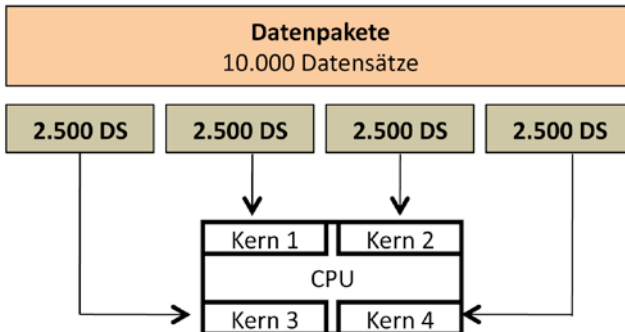


Abbildung 2.3: Optimale Aufteilung der Arbeitspakete

Nun sind alle Prozessoren bis zum Schluss mit Arbeit versorgt. Das führt gleichzeitig zu einem besseren Laufzeitverhalten der Anwendung. Die Aufteilung der Arbeitspakete sollte immer zur Laufzeit und unter Berücksichtigung der verfügbaren Prozessoranzahl automatisch erfolgen. Statische Festlegungen der Paketgrößen im Programmcode sollte vermieden werden. Außerdem sollten nicht zu viele Pakete erzeugt werden. Werden sehr viel mehr Pakete erzeugt, als Prozessoren bzw. Kerne zur Verfügung stehen, entsteht ein erhöhter Arbeitsaufwand, um die Pakete den Prozessoren zuzuweisen. Soll z. B. auf einem Dual-core eine Datenmenge von 500 Elementen verarbeitet werden, so sind theoretisch folgende Datenpakete möglich:

- 2 x 250 Elemente
- 4 x 125 Elemente
- 5 x 83 Elemente und 1 x 85 Elemente
- 7 x 63 Elemente und 1 x 59 Elemente

Da auf einem Dual-core nie mehr als zwei Aufgaben zur gleichen Zeit erledigt werden können, bietet sich die 2-x-250-Elemente-Aufteilung an. Handelt es sich jedoch um speicherintensive Daten, so ist auch die 4-x-125-Elemente-Aufteilung interessant, da dann vielleicht für jede Aufgabe alle Daten in den Cache-Speicher geladen werden können. Die richtige Kombination ist nicht einfach zu finden und bedarf teilweise mehrerer Testläufe um die richtigen Werte zu ermitteln. Dabei darf nie vergessen werden, dass sich im Laufe der Zeit mit hoher Wahrscheinlichkeit die Datenmenge ändert.

Funktionale Zerlegung (Functional Decomposition)

Die zuvor dargestellte Domänenzerlegung stellt die zu bearbeitenden Daten in den Mittelpunkt und teilt diese auf. Die Funktionale Zerlegung legt den Schwerpunkt auf Funktionen und zerlegt ein komplexes Problem in mehrere Funktionen, die ausgeführt werden müssen. Mehrere Funktionen können dann zeitgleich auf unterschiedlichen Prozessoren ausgeführt werden. Statt von Funktionen zu sprechen, kann auch der Be-

griff „Aufgabe“ verwendet werden. Teilweise operieren die Funktionen auf gemeinsame Daten (Shared Memory), aus Leistungsgründen sollte die Menge gemeinsamer Daten jedoch gering gehalten werden. Abbildung 2.4 zeigt die Zerlegung einer Funktion in mehrere Teilfunktionen.

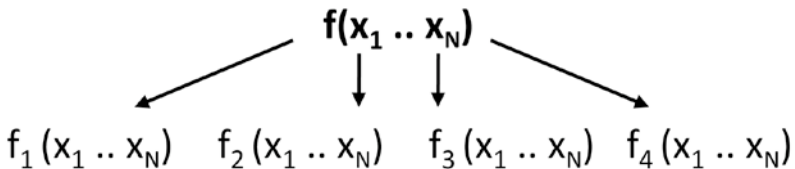


Abbildung 2.4: Funktionale Zerlegung

Da die einzelnen Teilfunktionen unabhängig voneinander auf unterschiedlichen Prozessoren ausgeführt werden sollen, sollten keine bzw. nur geringe Abhängigkeiten zwischen den Teilfunktionen bestehen. Die Zerlegung einer umfassenden Funktion in Teilfunktionen ist schon seit Langem im Bereich der synchronen Programmierung üblich. Mithilfe von Refactoring-Werkzeugen kann das sogar teilweise automatisiert erfolgen. Die Funktionale Zerlegung ist gegenüber der Datenzerlegung in Bezug auf Skalierbarkeit beschränkt. Da die Anzahl der Funktionen nicht variiert, kann nicht flexibel auf Hardwareänderungen (zusätzliche CPU-Kerne) reagiert werden. Daher ist eine flexible Lastverteilung schwierig umsetzbar.

Master-Slave-Modell (Master-Slave Model)

Das Master-Slave-Modell ist ein oft verwendetes Muster, um parallele Systeme umzusetzen. Die Abfolge kann in drei Sequenzen/Phasen unterteilt werden:

- Vorarbeiten (Preprocessing)
- Erledigung der Aufgabe (Processing)
- Nacharbeiten (Postprocessing)

Wie der Name des Modells vermuten lässt, werden Prozesse in Master- und Slave-Prozesse unterteilt. Soll eine Aufgabe gelöst werden, wird die Aufgabe an einen Master-Prozess weitergereicht. In der ersten Phase (Preprocessing) ist der Master-Prozess dafür verantwortlich, die benötigten Daten für die anstehende Aufgabe zu ermitteln und einen neuen Thread zu initiieren. Ist diese Vorarbeit abgeschlossen, wird die Aufgabe an einen untergeordneten Prozessor (Slave) weitergegeben. In dieser zweiten Phase wird die Aufgabe bearbeitet. Nach Beendigung der Aufgabe sendet der Slave das Ergebnis an den Master zurück. Der Master empfängt das Ergebnis und führt gegebenenfalls nötige weitere Modifikationen durch (Postprocessing). Am Ende wird das Ergebnis dem Anfrager zur Verfügung gestellt.

Im Bereich von verteilten Systemen, wo einzelne Rechnerknoten zu einem Verbund (Cluster) zusammengefügt werden, bietet sich das Muster an, um die einzelnen Knoten mit Arbeit zu versorgen. Einige Knoten agieren dabei als Masterknoten und verteilen die Arbeit an untergeordnete Knoten des Verbundes. Dabei ist auf eine gleichmäßige Lastverteilung zu achten.

Wichtig bei allen drei Möglichkeiten der Aufteilung, die zu einer parallelen Lösung führen, ist die optimale Verteilung der Arbeitslast auf alle vorhandenen Ressourcen. Dabei ist die Nutzung aller verfügbaren CPU-Kerne enorm wichtig. Das mit den bisherigen Bordmitteln unter .NET 2.0 bzw. .NET 3.5 effektiv umzusetzen, wäre prinzipiell möglich, aber sehr komplex. Mit den Erweiterungen unter .NET 4.0 sind viele der hier genannten Algorithmen bereits integriert. Ab Kapitel 6 werden diese Neuerungen ausführlich dargestellt.

2.1.2 Zustandsverwaltung (Shared State)

Ein weiteres Problem, das mit Parallel Computing einhergeht, ist die Verwaltung von Zuständen. Bei der Umsetzung eines rein sequenziellen Programms werden Zustände nur vom Prozess-Thread (Master Thread) modifiziert. Bei einer parallelen Lösung werden jedoch gemein-

same Zustände von mehreren, parallel aktiven Threads gelesen und schreibend verändert. Da diese Zugriffe gleichzeitig erfolgen können, muss sichergestellt werden, dass eine gemeinsame Speicherstelle nur von einem Thread zu einem Zeitpunkt verändert werden kann. Kann das nicht gewährleistet werden, wird früher oder später eine so genannte Race Condition (Data Race) während der Programmausführung erzeugt. Typischerweise treten Race Conditions nicht deterministisch auf und können nicht mittels klassischen (synchronen) Debug-Tools gefunden werden. Die folgende Sequenz demonstriert, wie eine Race Condition, auf einem Single-core-System, während der Programmausführung auftritt:

T[1]: Lädt Zustand A vom gemeinsamen Speicher in CPU-Register

T[1]: Modifiziert gemeinsamen Zustand (z. B. Addition eines Werts)

T[2]: Lädt Zustand A vom gemeinsamen Speicher in CPU-Register

T[2]: Modifiziert gemeinsamen Zustand

T[1]: Schreibt veränderten Zustand zurück in den gemeinsamen Speicher

T[2]: Schreibt veränderten Zustand zurück in den gemeinsamen Speicher

Nachdem der Thread 1 (T[1]) den Wert aus dem gemeinsamen Speicherbereich gelesen und modifiziert hat, wird dieser unterbrochen und Thread 2 (T[2]) wird aktiv. Auch dieser Thread liest den gleichen Wert aus dem gemeinsamen Speicher, modifiziert diesen und schreibt den geänderten Wert allerdings erst zurück, nachdem T[1] den veränderten Wert seiner Berechnung zurückgeschrieben hat. Somit ist die Änderung von T[1] überschrieben worden und es scheint so, als hätte die Berechnung von T[1] nie stattgefunden. Ein ähnliches Bild ergibt sich, wenn der gleiche Code auf einem Multi-core-System ausgeführt wird. Auf einem Multi-core-System kommt es zwar zu keiner Unterbrechung von Threads, aber durch die zeitgleiche Ausführung entsteht ein gleiches Problem. Die nachfolgende Sequenz (Tabelle 2.1) verdeutlicht das Verhalten auf einem Multi-core-System:

Time	Prozessor 1	Prozessor 2
1	Lädt Zustand A vom gemeinsamen Speicher in CPU-Register	
2	Modifiziert gemeinsamen Zustand (z. B. Addition eines Werts)	Lädt Zustand A vom gemeinsamen Speicher in CPU-Register
3	Schreibt veränderten Zustand zurück in den gemeinsamen Speicher	Modifiziert gemeinsamen Zustand
4		Schreibt veränderten Zustand zurück in den gemeinsamen Speicher

Tabelle 2.1: Ausführungssequenz auf einer Dual-core-Maschine

Im optimalen Fall führen solche Race Conditions zu einer Ausnahme (Exception) und die Programmausführung bricht ab oder aber die Fehler werden erst durch spätere manuelle Nachberechnungen gefunden. Das kann nicht nur bei finanzorientierten Systemen fatale Folgen haben.

Um solche Race Conditions zu vermeiden, muss der Zugriff auf gemeinsame Daten synchronisiert werden. Es muss sichergestellt werden, dass zu einem Zeitpunkt gemeinsame Daten nur durch einen Thread verändert werden. Zu diesem Zweck werden Zugriffe auf gemeinsame Daten mittels Sperren (Locks) synchronisiert. Eine Sperre (Lock) schützt den Zugriff auf gemeinsame Daten. Bevor ein Thread auf geschützte gemeinsame Daten zugreifen kann, muss dieser die Sperre besitzen. Eine Sperre kann dabei in der Regel nur von einem Thread in Besitz genommen werden. Abbildung 2.5 zeigt die Position einer Sperre.

Benötigt ein Thread Zugriff auf gemeinsame Daten, kann aber die Sperre nicht in Besitz nehmen, muss er warten, bis die Sperre verfügbar ist. Sperren sind ein effektives Mittel, um Race Conditions zu vermeiden, müssen aber mit Bedacht und Sorgfalt eingesetzt werden. Da Sperren für eine Synchronisierung des Zugriffs auf gemeinsame Daten sorgen, führen zu viele Sperren zu einer sequenziellen Lösung. Somit wären die Geschwindigkeitsvorteile einer parallelen Lösung aufgehoben.

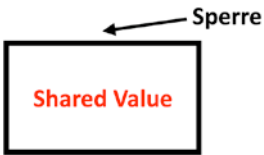


Abbildung 2.5: Geschützte Ressource

Um die Funktionsweise einer Sperre zu verdeutlichen, wird das vorherige Beispiel um eine Sperre auf den Zustand A erweitert. Durch den Einsatz einer Sperre ergibt sich die in Tabelle 2.2 aufgezeigte Ausführungssequenz.

Time	Prozessor 1	Prozessor 2
1	Nimmt Sperre A in Besitz	
2	Lädt Zustand A vom gemeinsamen Speicher in CPU-Register	Nimmt Sperre A in Besitz
3	Modifiziert gemeinsamen Zustand (z. B. Addition eines Werts)	Wartet auf Verfügbarkeit von Sperre A
4	Schreibt veränderten Zustand zurück in den gemeinsamen Speicher	Wartet auf Verfügbarkeit von Sperre A
5	Gibt Sperre A frei	Wartet auf Verfügbarkeit von Sperre A
6		Nimmt Sperre A in Besitz
7		Lädt Zustand A vom gemeinsamen Speicher in CPU-Register
8		Modifiziert gemeinsamen Zustand (z. B. Addition eines Werts)
9		Schreibt veränderten Zustand zurück in den gemeinsamen Speicher

Tabelle 2.2: Ausführungssequenz mit Sperre auf einer Dual-core-Maschine

Wie hier gut zu erkennen ist, wird nun mehr Zeit für die Ausführung benötigt. Das ist auch notwendig, da ansonsten ein falsches Ergebnis erzielt werden würde. Jede geschützte gemeinsame Ressource kann zu einer Ausführungsverzögerung führen, daher sollte genau überlegt werden, ob ein gemeinsamer Zustand wirklich benötigt wird. Je weniger Abhängigkeiten zwischen den einzelnen parallelen Aufgaben existieren, umso eine höhere Verarbeitungsgeschwindigkeit ist erzielbar.

Erkennung vom gemeinsamen Speicher

Wie an den vorherigen Beispielen erkennbar war, muss der Zugriff auf gemeinsame Zustände synchronisiert werden. Die vielleicht zunächst simple Fragestellung „Wie erkenne ich eine gemeinsame Ressource?“, ist nicht immer so leicht zu beantworten. Wird als Programmiersprache C++ eingesetzt, verschärft sich diese Fragestellung noch, da durch Zeigeroperationen auch zunächst nicht gemeinsame Speicherstellen plötzlich durch andere Threads verändert werden könnten. Auch die Nutzung einer als *static* deklarierten Variablen kann zu Problemen führen. Generell sollte der Zugriff auf eine gemeinsame Ressource nur über eine Eigenschaft (*get/set*-Eigenschaft in .NET) oder eine entsprechende Methode (z. B. *IncrementCounter()*) möglich sein. Innerhalb dieser Zugriffsmethoden muss dann der Zugriff mittels Sperren synchronisiert werden. Der freie Zugriff auf diese Ressourcen muss unterbunden werden, da ansonsten Sperren umgangen werden könnten.

2.1.3 Selbstblockade (Deadlock)

Wie weiter oben beschrieben wurde, muss der Zugriff auf gemeinsame Ressourcen synchronisiert und kontrolliert erfolgen. Um diesen Zugriffsschutz umzusetzen, werden Sperren (Locks) eingesetzt. Bevor ein Thread eine gemeinsame Ressource nutzen kann, muss er die dazugehörige Sperre (Lock) anfordern. Ist die Sperre bereits in Besitz eines anderen Threads, muss der Thread auf die Verfügbarkeit der Sperre warten. Genau an dieser Stelle kann es nun zu einem Deadlock (auch bezeichnet als: Verklemmung oder Selbstblockade) kommen. Abbildung 2.6 verdeutlicht die Entstehung einer Deadlock-Situation.

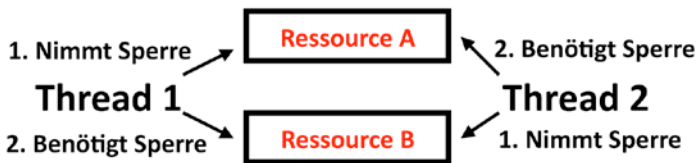


Abbildung 2.6: Wartegraf (Deadlock)

Thread 1 (T1) fordert Zugriff auf die gemeinsame Ressource A und nimmt die zugehörige Sperre in Besitz. Gleichzeitig ist Thread 2 (T2) aktiv und benötigt zunächst Zugriff auf die gemeinsame Ressource B. T2 nimmt dazu die freie Sperre für die Ressource B in Besitz. Im weiteren Verlauf von T1 benötigt dieser ebenfalls Zugriff auf die gemeinsame Ressource B. Diese befindet sich aber momentan im Besitz von T2 und kann daher nicht erlangt werden. T1 wartet nun solange, bis die Ressource B frei wird. T2 benötigt aber selbst, um weiter fortfahren zu können, Zugriff auf die gemeinsame Ressource A. Da T2 nicht sofort die Sperre erlangen kann, wartet T2 nun, bis die Sperre verfügbar ist. Nun ist eine Deadlock-Situation eingetreten. T1 kann nicht weiterarbeiten, solange T2 im Besitz der Sperre für die gemeinsame Ressource B ist. T2 kann aber erst die Sperre freigeben, wenn dieser zuvor Zugriff auf die Sperre für die gemeinsame Ressource A hatte.

Da Deadlock Situationen nicht deterministisch eintreten, ist das Aufspüren keine leichte Angelegenheit. Um ein endloses Warten auf eine Sperre zu vermeiden, kann optional eine maximale Zeitperiode definiert werden. Konnte nach Ablauf der Zeitperiode die Sperre nicht in Besitz genommen werden, kann dieser Systemzustand protokolliert werden. Umfangreiche und detaillierte Log-Informationen können oft helfen, eine Deadlock-Situation zu entdecken und zu lösen.

Deadlock-Erkennung mittels Wartegrafen

Mögliche Deadlock-Situationen können im Vorfeld oder mithilfe der Log-Informationen durch einen Wartegrafen (auch bekannt als Resource Dependency Graph) visualisiert und gelöst werden. Wartegrafen wer-

den oft bei der Verwendung von verteilten Datenbanktransaktionen eingesetzt, eignen sich aber auch um Threading-Sperrkonflikte zu visualisieren.

Ein Wartegraf stellt die Beziehung zwischen einzelnen Threads und abhängigen Objekten dar. Wartegrafen stellen allerdings nur eine statische Sicht auf den aktuellen Zustand der Sperren dar und eignen sich daher oft nicht zur Vorherbestimmung auftretender Deadlock-Situationen. Abbildung 2.7 zeigt einen typischen Wartegrafen.

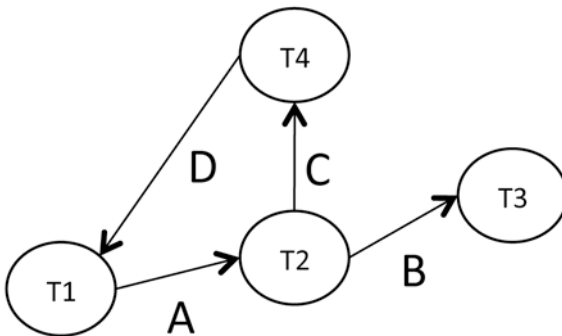


Abbildung 2.7: Wartegraf (Deadlock)

An den Kanten wird jeweils die Sperre (oder das gemeinsame Objekt), auf die gewartet wird, vermerkt. Entsteht bei dem Grafen ein zyklischer Verlauf, so wurde eine Deadlock-Situation aufgedeckt. In dem Beispiel aus Abbildung 2.7 besteht der Deadlock aus $T1 > T2 > T4$. T1 benötigt, um fortfahren zu können, die Ressource A, die sich gerade im Besitz von T2 befindet. T2 hingegen benötigt die Ressourcen B und C. Die Ressource B ist im Besitz von T3. T3 benötigt keine weiteren Ressourcen und kann seine Arbeit beenden. Danach gibt T3 die Ressource B frei, die dann durch T2 in Besitz genommen werden kann. Allerdings benötigt T2 noch die Ressource C, diese ist im Besitz von T4. T4 kann aber erst die Ressource freigeben, wenn T4 die Ressource D nutzen kann. Diese kann nicht genutzt werden, da diese von T1 gesperrt ist. Ergebnis: Deadlock.

Vermeidung von Deadlock-Situationen

Eine Deadlock-Situation im Nachhinein zu entdecken und zu beheben, ist nicht einfach. Schon beim Design der Anwendung sollten solche Zustände beachtet und vermieden werden. Eine Regel, die dabei beachtet werden sollte, ist die Reihenfolge der Sperranforderungen. Die Reihenfolge, wie Sperren angefordert und in Besitz genommen werden, sollte stets die gleiche sein. Hätte man diese Regel im ersten Beispiel (Abb. 2.6) angewendet, würde es zu keiner Deadlock-Situation kommen. In dem Beispiel hatte Thread 1 die Sperrsequenz $A > B$ verwendet und Thread 2 $B > A$. Würde generell die Sequenz $A > B$ verwendet, könnte der Deadlock vermieden werden. Bei dieser Sequenz muss T2 erst warten, bis die Sperre auf A verfügbar ist, bevor er die Sperre auf B erlangen kann. Somit kann T1 seine Arbeit erledigen und danach die Sperren wieder freigeben.

Wendet man die gleiche Regel auf dem Wartegraf aus Abbildung 2.7 an, kann ebenfalls die Deadlock-Situation aufgelöst werden. Zunächst ist in Tabelle 2.3 die originale Reihenfolge der Sperrsequenzen dargestellt.

Thread 1 (T1)	Thread 2 (T2)	Thread 3 (T3)	Thread 4 (T4)
D	A	B	C
A	B		D
	C		

Tabelle 2.3: Reihenfolge der Sperranforderungen

Legt man als generelle Reihenfolge $A > B > C > D$ fest, fordert nur Thread 1 die Sperren in einer falschen Reihenfolge an. Korrigiert man dieses, ergibt sich der unter Abbildung 2.8 Deadlock-freie Wartegraf. Thread 1 fordert nun zunächst die Sperre A und danach die Sperre D an. Da Thread 1 in Besitz beider Sperren ist, kann er die Arbeit beenden und die Sperren freigeben. Danach (oder auch parallel) kann Thread 3 seine Arbeit beenden und die Sperre B freigeben. Thread 4 war schon im Besitz der Sperre C und kann nun auch die Sperre D in Besitz nehmen. Nachdem Thread 4 die Arbeit beendet hat, gibt dieser die Sperren wieder frei und Thread 2 kann die fehlenden Sperren beziehen und seine Arbeit ebenfalls beenden.

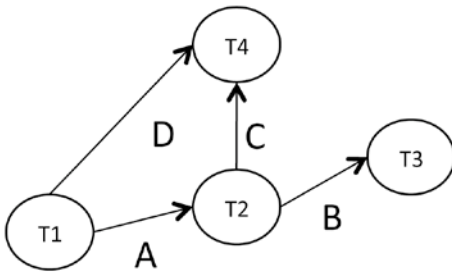


Abbildung 2.8: Auflösung eines Deadlocks

Livelock

Beim Livelock handelt es sich ebenfalls um eine Art Deadlock. Hierbei kommt allerdings der Prozess nicht gänzlich zum stehen, sondern er wechselt periodisch seinen Zustand. Ein wirklicher Arbeitsfortschritt wird dabei nicht mehr erzielt. Rechenzeit wird lediglich durch den ständigen Wechsel zwischen 2 oder mehreren Zuständen verbraucht.

Folgende Situation beschreibt das Auftreten eines Livelocks. In einem System existieren 2 Arbeit-Threads (Thread A und B). Diese greifen auf eine gemeinsame Queue zu, in der sich Arbeitsaufträge befinden. Einträge in die Queue werden von einem dritten Thread (Thread W) übernommen. Zu einem Zeitpunkt entnimmt Thread A den letzten Eintrag aus der Queue. Thread W hat einen neuen Eintrag, der in die Queue geschrieben werden soll. Allerdings kann Thread W zu diesem Zeitpunkt nicht auf die Queue zugreifen, da diese von Thread A gesperrt wurde. Nachdem Thread A den letzten Eintrag entnommen hat, gibt er die Sperre auf die Queue wieder frei. Als Nächstes erlangt dann Thread B die Sperre auf die Queue. Thread B findet keine Einträge vor und gibt die Sperre wieder frei. Danach wird Thread A wieder aktiv und holt sich die Sperre auf die Queue. Das heißt, die Sperre auf die Queue wechselt zwischen Thread A und B. In diesem Fall ist Thread W nie in der Lage, einen neuen Eintrag in die Queue zu schreiben (Kapitel 2.1.4). In diesem Fall kann ein Livelock durch unterschiedliche Prioritäten gelöst werden. Eine höhere Priorität kann Thread W zugewiesen werden, da dieser die Queue mit Aufgaben

füllt. Ebenfalls ist eine temporäre Erhöhung der Priorität möglich. Dabei werden die Versuche gezählt, eine Sperre auf die Queue zu erlangen. Wurde ein bestimmter Grenzwert erreicht, kann die Priorität von Thread W heraufgesetzt werden.

2.1.4 Starvation

Innerhalb eines Computersystems konkurrieren Prozesse und deren Threads um begrenzte Ressourcen. Beim zuvor beschriebenen Deadlock kann es dabei zum Stillstand kommen. Ebenfalls kritisch ist der Vorgang des Verhungerns (Starvation) einer Aufgabe. In diesem Zustand ist die Aufgabe theoretisch in der Lage, verarbeitet zu werden, bekommt aber meist aufgrund einer zugewiesenen Priorität nicht genügend Rechenzeit.

Als Beispiel für Starvation kann ein prioritätsorientierter Verarbeitungsprozess genannt werden. Innerhalb eines Systems bearbeitet ein Thread Arbeitspakete nach dem Consumer/Producer-Prinzip. Mehrere Producer produzieren Arbeitspakete und stellen diese in eine Warteschlange. Der Consumer arbeitet in diesem Fall die Warteschlange nicht nach dem FiFo (First in First out) Prinzip ab, sondern fragt die Prioritäten der Arbeitspakete ab. Ist nun ein Producer im System aktiv, der fortlaufend Arbeitspakete produziert, deren Priorität hoch ist, kann das dazu führen, dass Arbeitspakete geringerer Priorität nicht verarbeitet werden.

Starvation kann verhindert werden, indem nicht nur ein Merkmal zur Prioritätsbestimmung herangezogen wird. Besser ist eine Kombination von festgelegter Priorität und verstrichener Wartezeit. Wurde eine bestimmte Wartezeit überschritten, sollte automatisch die Priorität der Aufgabe erhöht werden.

2.1.5 Fehlerbehandlung

Neben der Beachtung von Deadlock-Situationen ist eine korrekte Fehlerbehandlung sehr wichtig. Die Fehlerbehandlung bei parallelen Lösungen ist aufwendiger und schwieriger als bei sequenziellen Programmen. Bei einem sequenziellen Programm kann zu einem Zeitpunkt nur ein Fehler ausgelöst werden. Da bei einem parallelen Programm gleichzeitig mehre-

re Anweisungen auf unterschiedlichen Prozessoren ausgeführt werden, können auch gleichzeitig mehrere Fehler produziert werden. Tritt ein Fehler innerhalb eines Threads auf, müssen auf jeden Fall die aktuell gehaltenen Sperren auf gemeinsame Ressourcen freigegeben werden. Ansonsten können wartende Threads nicht fortfahren, da Sperren nie freigegeben werden. Somit würde der gesamte Prozess zum Stillstand kommen.

Wurden bereits Zustände/Werte von gemeinsamen Ressourcen verändert, ist zu überlegen, ob diese zurückgesetzt werden müssen. Werden diese nicht zurückgesetzt, kann es unter Umständen zu ungünstigen Systemzuständen kommen, da die Verarbeitung unkontrolliert durch einen Fehler unterbrochen wurde. Außerdem darf auch im Fehlerfall nicht vergessen werden, reservierte Systemressourcen, die nicht durch die Garbage Collection verwaltet werden, ordnungsgemäß wieder freizugeben. Die bisher aufgeführten Maßnahmen betrafen nur den Thread, der die Ausnahme ausgelöst hat. In vielen Fällen bearbeiten einzelne Threads aber Teile einer großen Aufgabe. Was geschieht nun mit den anderen Threads, die bisher fehlerfrei ausgeführt wurden? Oder anders gefragt: Kann die Gesamtaufgabe auch ohne das Ergebnis des fehlerhaften Threads erfolgreich gelöst werden? Kann auf das Ergebnis nicht verzichtet werden, können zwei Maßnahmen eingeleitet werden:

- Alle weiteren Threads werden ebenfalls terminiert
- Der fehlerhafte Thread wird noch einmal gestartet

Bei der ersten Option werden alle aktiven und noch anstehenden Threads abgebrochen. Die Verarbeitung der Gesamtaufgabe muss somit noch einmal gestartet werden. Die zweite Variante ist effektiver, da hier nur der fehlerhafte Thread neu gestartet wird. Das ist allerdings nur möglich, wenn der Thread im Fehlerfall einen sauberen Systemzustand hinterlässt. In beiden Fällen muss zuvor sichergestellt werden, dass der Fehler bei einer erneuten Ausführung nicht mehr auftritt.

Da es sich bei einer parallelen Lösung um ein komplexes Implementierungskonstrukt handelt, ist es wichtig, dass Fehlerbehandlungen konsequent und durchgängig umgesetzt werden. Ansonsten kann eine nicht richtig behandelte Ausnahme ein gesamtes System zum Absturz bringen.

3.4 Atomare Operationen

Atomare Operationen werden wahrscheinlich die meisten Leser an Datenbanktransaktionen und das ACID-Prinzip erinnern. ACID steht dabei für:

- Atomicity (Atomarität)
- Consistency (Konsistenzerhaltung)
- Isolation (Isolation)
- Durability (Dauerhaftigkeit)

Es beschreibt die Anforderung an eine Datenbanktransaktion. Im Datenbankkontext bedeutet das für eine Transaktion:

Die ausgeführte Operation muss atomar sein, d. h. alle betroffenen Daten werden verändert oder keine. Konsistenz bezieht sich auf Datenintegrität und definierte Schlüssel- und Fremdschlüsselbeziehungen. Diese muss nach Abschluss der Transaktion gewährleistet sein. Isolation verhindert, dass sich gleichzeitig ausgeführte Transaktionen beeinflussen oder behindern. Dauerhaftigkeit garantiert, dass durchgeführte Änderungen auch nach einem Systemabsturz bestehen bleiben.

Ein ähnliches Prinzip gilt auch für Anwendungen, die mehrere Threads einsetzen. Sobald ein Thread eine nicht atomare Anweisung ausführt, die einen Wert einer Variablen ändert, muss diese Operation atomar und isoliert durchgeführt werden. Atomare Operationen zeichnen sich dadurch aus, dass sie nicht unterbrochen werden können. Die kleinste atomare Operation ist demnach die Ausführung eines Maschinenbefehls. Nun

stellt sich die Frage: Was ist eine atomare Operation in einer Hochsprache wie C#? Diese Fragestellung soll anhand eines einfachen Beispiels (Listing 3.12) geklärt werden.

```
class ACIDThreading
{
    public const int ITERATIONS = 500;
    private int counter = 0;
    public int Counter
    {
        get
        {
            return counter;
        }
    }
    public void ThreadA()
    {
        for (int i = 0; i < ITERATIONS; i++)
            counter++;
    }
    public void ThreadB()
    {
        for (int i = 0; i < ITERATIONS; i++)
            counter++;
    }
    public void Reset()
    {
        counter = 0;
    }
    static void Main(string[] args)
    {
        int tries = 0;
        ACIDThreading acid = new ACIDThreading();
        do
        {
            tries++;
            acid.Reset();
            Thread a = new Thread
                (new ThreadStart(acid.ThreadA));
            Thread b = new Thread
                (new ThreadStart(acid.ThreadB));
```

```
        a.Start();
        b.Start();
        a.Join();
        b.Join();
    }
    while (ACIDThreading.ITERATIONS * 2 == acid.
Counter);
    Console.WriteLine("Expected result: {0}.
Calculated result: {1}.
Tries: {2}", ACIDThreading.ITERATIONS * 2,
acid.Counter, tries);
    Console.ReadLine();
}
}
```

Listing 3.12: Änderung einer gemeinsamen Variablen

Das Beispiel in Listing 3.12 startet zwei Threads, beide Threads durchlaufen eine identisch lange Schleife, festgelegt durch die Konstante *ITERATIONS*, und inkrementieren bei jedem Durchlauf die gemeinsame Variable *counter*. Gemäß der Programmlogik kann der Wert der Variablen *counter* wie folgt berechnet werden:

```
counter = ITERATIONS * 2
```

In diesem Fall ist die Konstante *ITERATIONS* auf 500 festgelegt worden, daher wird nach Beendigung der beiden Threads ein Wert von 1 000 erwartet. Diese Erwartung wird auch oft bestätigt, teilweise liegt der *counter*-Wert aber unter 1 000. Schaut man sich den Programmcode an, um die Fehlerursache zu finden, ist der Fehler nicht unbedingt offensichtlich zu erkennen. Das umgesetzte Programm kann an jeder Stelle unterbrochen und fortgesetzt werden, die Anweisungen innerhalb der Thread-Methoden (*ThreadA* und *ThreadB*) sind augenscheinlich atomar. Dem ist aber nicht so. Jedes Hochsprachenprogramm wird kompiliert und früher oder später in Maschinensprache umgesetzt. Um atomare Operationen zu erkennen, ist, wie bereits angedeutet, der Maschinencode von Interesse. In dem kleinen Beispielprogramm verursacht die harmlos wirkende Anweisung *counter++* die beobachteten Probleme. Um nicht in die Tiefen der

Atomare Operationen

Assembler-Anweisungen einzutauchen, reicht schon ein Blick auf den erzeugten Intermediate-Language-Code (IL-Code), um das Problem zu erkennen. Listing 3.13 zeigt nur die IL-Anweisungen für die ++-Operation.

```
IL_0003: ldfld      int32 ThreadSimple.  
ACIDThreading::counter  
IL_0008: ldc.i4.1  
IL_0009: add  
IL_000a: stfld      int32 ThreadSimple.  
ACIDThreading::counter
```

Listing 3.13: Der IL-Code für die Anweisung ++

Wie anhand der IL-Instruktionen zu erkennen ist, wird die ++-Anweisung in mehrere Befehle gesplittet. Zunächst wird der Wert der *counter*-Variablen in ein Register geladen. Im Folgenden wird der Wert 1 addiert und zum Schluss wird der neue Wert wieder in die Variable *counter* zurückgeschrieben. Der hier gezeigte IL-Code ist nicht Thread-safe, da die IL-Anweisungen an jeder Stelle unterbrochen werden könnten, um einen anderen Thread auszuführen. Tabelle 3.3 zeigt einen möglichen Verlauf, wie ein falscher Wert entstehen kann.

T	Thread A	Thread B
1	ldfld int32 ACIDThreading::counter	
2	ldc.i4.1	
3		ldfld int32 ACIDThreading::counter
4		ldc.i4.1
5		add
6		stfld int32 ACIDThreading::counter
7	add	
8	stfld int32 ACIDThreading::counter	

Tabelle 3.3: Unterbrechung nichtatomarer Operationen

Zunächst startet Thread A und liest den Wert der Variablen *counter* ein. Dieser ist zu Beginn 0. Später wird Thread A unterbrochen und Thread B bekommt Rechenzeit zugewiesen. Dieser liest ebenfalls den Wert der Variablen *counter* ein, der zu diesem Zeitpunkt ebenfalls 0 ist. Danach inkrementiert Thread B den ermittelten Wert um 1 und schreibt den neuen Wert zurück. Danach fährt Thread A mit der Berechnung fort. Er hat allerdings immer noch den zuvor gelesenen Wert 0 der Variablen *counter* im Register. Thread A hat zu diesem Zeitpunkt keinerlei Informationen darüber, dass der Wert zwischenzeitlich verändert wurde. Also inkrementiert er ebenfalls den gespeicherten Wert um 1 und schreibt ihn zurück. Am Ende hat die Variable *counter* den Wert 1 anstatt 2. Gleiches Verhalten ist auch auf einem Mehrkernprozessorsystem zu beobachten. Dort kann es vorkommen, dass zeitgleich der Wert einer Variablen in ein Register geschrieben und zur Berechnung verwendet wird.

Um dieses Problem zu umgehen, existieren verschiedene Möglichkeiten. Im Folgenden wird die Verwendung der *Interlocked*-Klasse vorgestellt. In Kapitel 4 werden erweiterte Techniken demonstriert, die einen stabilen Systemzustand garantieren.

3.4.1 Die Methoden der *Interlocked*-Klasse

Wie zuvor gezeigt, handelt es sich nicht bei jeder augenscheinlich atomaren Operation wirklich um eine solche. Solange mehrere Threads nur jeweils auf wenige gemeinsame Variablen zugreifen, die keine Wechselwirkung untereinander haben, können die Methoden der *Interlocked*-Klasse verwendet werden. Tabelle 3.4 gibt einen Überblick über die angebotenen Methoden und deren Funktionsweise.

Methoden	Beschreibung
Add	Addiert zwei übergebene Zahlen innerhalb einer atomaren Operation. Die Summe wird in dem als Referenz übergebenen ersten Parameter gespeichert. Der Rückgabewert ist ebenfalls die berechnete Summe. Als Summanden können <i>int</i> - und <i>long</i> -Parametertypen übergeben werden.

Atomare Operationen

Methode	Beschreibung
Compare-Exchange	Ermöglicht den Vergleich zweier Variablen und ersetzt bei Gleichheit den Wert einer Variablen durch einen neuen Wert.
Decrement	Dekrementiert den Wert der per Referenz übergebenen Variablen. Ebenfalls wird der neue Wert der Variablen zurückgegeben. Es wird keine Ausnahme ausgelöst, wenn der übergebene Wert gleich <i>int.MinValue</i> bzw. <i>long.MinValue</i> ist. Die Methode liefert in diesem Fall <i>int.MaxValue</i> bzw. <i>long.MaxValue</i> zurück.
Exchange	Ermöglicht die Zuweisung eines Werts an eine Variable innerhalb einer atomaren Operation. Der Rückgabewert ist der Wert der Variablen vor der Zuweisung des neuen Werts.
Increment	Inkrementiert den Wert der per Referenz übergebenen Variablen. Ebenfalls wird der neue Wert der Variablen zurückgegeben. Es wird keine Ausnahme ausgelöst, wenn der übergebene Wert gleich <i>int.MaxValue</i> bzw. <i>long.MaxValue</i> ist. Die Methode liefert in diesem Fall <i>int.MinValue</i> bzw. <i>long.MinValue</i> zurück.
Read	Ermöglicht das atomare Lesen von 64-Bit-Werten auf einem 32-Bit-System.

Tabelle 3.4: Methoden der *Interlocked*-Klasse

Für eine vollständige Thread-Synchronisation reichen diese Methoden allerdings noch nicht aus. In Kapitel 4 werden daher erweiterte Konzepte und Techniken vorgestellt.

Mithilfe der verfügbaren statischen Methoden der *Interlocked*-Klasse kann aber das dargestellte Problem aus dem Beispiel 3.12 gelöst werden. Dort verursacht die nicht atomare ++-Operation Probleme, wenn mehrere Threads diese Operation auf einer gemeinsamen Variablen zeitgleich ausführen. In diesem Fall müssen nur die beiden Thread-Methoden (*ThreadA* und *ThreadB*) wie in Listing 3.14 umgeschrieben werden.

```
public void ThreadA()
{
    for (int i = 0; i < ITERATIONS; i++)
        Interlocked.Increment(ref counter);
}
public void ThreadB()
{
    for (int i = 0; i < ITERATIONS; i++)
        Interlocked.Increment(ref counter);
}
```

Listing 3.14: Verwendung der `Interlocked.Increment`-Methode

Die so durchgeführte Änderung sorgt dafür, dass der Thread während der Durchführung der `Interlocked.Increment`-Anweisung nicht unterbrochen wird. Somit ist sichergestellt, dass kein weiterer Thread den Wert der betroffenen Variablen – in diesem Fall `counter` – zwischengespeichert im Zugriff hat. Intern verwendet die `Interlocked`-Klasse atomare Win32-API-Funktionen. Die `Interlocked`-Methoden sollten immer verwendet werden, wenn eine gemeinsame Variable von mehreren Threads verwendet wird. Werden erweiterte Synchronisierungsmöglichkeiten (Kapitel 4) verwendet, kann teilweise auf die Verwendung verzichtet werden.

HINWEIS: Neben den Methoden der `Interlocked`-Klasse existiert noch das Schlüsselwort `volatile` sowie einige `Thread.VolatileXXX`-Methoden. Allein durch die Verwendung des Schlüsselworts `volatile` kann das Problem aus dem Beispiel 3.12 nicht gelöst werden. Es garantiert zwar immer aktuelle Speicherwerte und verhindert die Codeoptimierung durch den Compiler, kann aber die Unterbrechung eines Threads an einer kritischen Stelle nicht verhindern. Die Funktion des Schlüsselworts `volatile` wird in Kapitel 3.5.1 behandelt.

6

Task Parallel Library

Die ersten beiden Kapitel haben sich mit den theoretischen Grundlagen beschäftigt und erläutern, welche Faktoren bei der parallelen Verarbeitung beachtet werden sollten. Ein wesentliches Problem besteht in der korrekten Aufteilung der Last auf die verfügbaren CPU-Ressourcen. Wie gezeigt, kann mithilfe des Threadpools eine Verbesserung erreicht werden. Jedoch ist die umgesetzte Lösung nicht trivial und lenkt zu sehr den Fokus auf technische Details. Ein Anwendungsentwickler sollte sich jedoch mehr auf fachliche Probleme konzentrieren, anstatt sich mit Thread-spezifischen Problemen auseinandersetzen zu müssen. Ein erster, wenn auch nicht vollständiger Lösungsansatz wurde mit .NET 4.0 eingeführt. Die unter dem Namen „Task Parallel Library“ – kurz TPL – eingeführte Concurrent-API-Erweiterung löst einige der typischen Probleme, die im Umgang mit „reinen“ Threads auftreten. Die neuen APIs, die im Namensraum (Namespace) `System.Threading` abgelegt sind, erleichtern die Nutzung von Parallelität unter .NET. Allerdings muss der Entwickler auch bei deren Verwendung Zugriffe auf gemeinsame Variablen bzw. Ressourcen selbst steuern. Er muss daher genau prüfen, sobald er die Möglichkeiten der TPL nutzt, ob nicht zusätzliche Sperren auf gemeinsame Ressourcen nötig sind. Das folgende Kapitel beschreibt die Möglichkeiten der neuen APIs und zeigt deren Funktionsweise.

HINWEIS: Alle hier erklärten und gezeigten APIs beziehen sich auf die .NET-Framework-Version 4.0. Aus Übersichtsgründen verdeutlichen die nachfolgenden Beispiele im Schwerpunkt die Verwendung des API. Auf eine durchgängige Auflistung aller Methoden und Eigenschaften der einzelnen Klassen wurde daher verzichtet. Die aktuellen Informationen hierzu können der Online-MSDN-Hilfe entnommen werden.

Vermeidung von Threads

Das Task-Konzept und die Task-Klasse erinnern stark an die Thread-Klasse. Auf den ersten Blick scheint die Task-Klasse lediglich eine Erweiterung der Thread-Klasse zu sein. Dem ist aber nicht so. Thread- und Task-Klasse haben zunächst nichts miteinander zu tun. Erst später, bei der konkreten Ausführung des Task-Objekts, kommen Threads zum Einsatz. Die Verwendung der Klasse *Task* bringt in vielen Situationen erhebliche Vorteile gegenüber der Nutzung der Thread-Klasse. Falsch verwendet, verschlechtern Threads die Leistung eines Programms, anstatt sie zu verbessern. Die folgenden Abschnitte erläutern die typischen Probleme, die bei der Verwendung von Threads auftreten können.

Zu viele Threads

Die Verwendung eines Threads kann zu Geschwindigkeitsverbesserungen führen. Das kann allerdings nur erreicht werden, wenn durch den zusätzlichen Thread Rechenleistung verwendet werden kann, die ansonsten nicht nutzbar wäre. Ein klassisches Anwendungssystem, das nur einen Thread verwendet und auf einem Mehrkernprozessor ausgeführt wird, kann niemals die Rechenleistung von mehreren Kernen nutzen. Ein solches Anwendungssystem ist nur in der Lage, einen Kern effektiv zu nutzen. Hier kann durch einen zusätzlichen Thread eine Verbesserung erreicht werden, da der Thread parallel die Rechenleistung eines anderen Kernels nutzen kann. Oft werden aber Threads falsch eingesetzt und es werden zu viele Threads erzeugt. Das führt dann zu einer Verschlechterung der Systemleistung. Werden z. B. auf einer Dual-core-Maschine innerhalb eines Anwendungssystems 40 Threads zur Bearbeitung einer Aufgabe erzeugt (z. B. Sortierung von 40 Listen, wobei jeder Sortiervorgang in einem separaten Thread ausgelagert wird), kämpfen alle diese Threads um die verfügbaren CPU-Kerne. Somit ist der Betriebssystem-Scheduler damit beschäftigt, den 40 wartenden Threads abwechselnd Rechenleistung zuzuweisen. Jeder Wechsel bedeutet aber einen teuren Kontextwechsel, der zusätzliche Rechenleistung für Verwaltungsaufgaben verbraucht. Es kann dabei sogar der Fall eintreten, dass eine entsprechende sequentielle Variante schneller ist als das parallele Gegenstück.

Daher sollten auf einem System jeweils nur so viele Threads erzeugt werden, wie CPU-Kerne zur Verfügung stehen. Das bedeutet, auf einer Dual-core-Maschine sind 2 und auf einer Quad-core-Maschine vier Threads das Optimum. Die anstehende Arbeit ist dann auf den Threads gleichmäßig aufzuteilen. Bezogen auf das Beispiel bedeutet dies, dass lediglich 2 Threads erzeugt werden sollten. Jedem Thread werden dann jeweils 20 Listen zur Sortierung zugeteilt.

Die obere Aussage ist jedoch nur dann gültig, wenn die Threads wirklich zeitgleich aktiv sind. Sind zu einem Zeitpunkt immer nur einige der erstellten Threads aktiv, können auch mehrere Threads erstellt und verwaltet werden. Das trifft zum Beispiel auf einen serverseitigen Prozess zu, der mehrere Clients verwaltet. Jeder Client spiegelt im Serverprozess eine Session wider und jede Session wird in einem separaten Thread ausgeführt. Das ist zwingend notwendig, da es ansonsten zu Verzögerungen bei clientseitigen Anfragen kommen würde. In diesem Fall ist allerdings davon auszugehen, dass mehr Threads vorhanden sind als logische Kerne zur Verfügung stehen. Das ist aber akzeptabel, da wahrscheinlich niemals alle Clients zeitgleich eine Anfrage stellen werden. Zu einem bestimmten Zeitpunkt sind also immer nur einige der gesamten vorhandenen Threads im System aktiv. Zudem können in der Regel die Anfragen relativ zügig verarbeitet werden und die Threads sind nur kurzfristig aktiv. Problematisch wird es hingegen wieder, wenn die Anzahl der gleichzeitigen Clients steigt und die Verarbeitung einzelner Anfragen länger dauert. Dann muss u. U. darüber nachgedacht werden, den Serverprozess auf mehreren Servern zu starten und die Anfragen mittels Lastverteilung (Load Balance) aufzuteilen. Kurzfristig würde es auch helfen, die maximale Anzahl aktiver Threads manuell zu beschränken. Hierbei muss allerdings berücksichtigt werden, dass das Antwortverhalten dadurch negativ beeinflusst wird.

MEINUNG: Die maximale Anzahl von Threads ist abhängig vom konkreten Verwendungsszenario. Handelt es sich um gleichzeitige Verarbeitungen innerhalb eines Anwendungssystems, sollte die maximale Anzahl der Threads gleich der Anzahl logischer Kerne sein. Bei serverseitigen Prozessen, bei denen die Anzahl aktiver Threads abhängig ist von externen Anfragen, kann die vorhandene Thread-Anzahl größer sein. Entscheidend ist hier die Anzahl der Threads, die tatsächlich gleichzeitig aktiv sind.

Speicher

Jeder zusätzliche Thread benötigt System Speicher. So wird unter .NET für jeden Thread standardmäßig 1 MB Speicher reserviert. Findet keine vernünftige Aufteilung der Arbeitslast statt und es werden zu viele Threads gleichzeitig erzeugt, kann dies zu einem Speicherüberlauf führen. Ebenfalls bedeutet jede Speicherallokation Zeit, die für die Lösung des jeweiligen Problems verloren geht.

Die neue Task-Klasse ist dagegen eher speicherschonend und es findet nicht sofort eine Speicherreservierung in der oben genannten Größe statt. Zunächst handelt es sich bei der Task-Klasse um eine normale .NET-Klasse, die Speicheranforderungen sind also eher gering. Somit stellt es auch kein Problem dar, sehr viele Task-Instanzen gleichzeitig zu erstellen. Auch wenn die spätere tatsächliche Ausführung durch einen Thread erfolgt, ist dies unproblematisch, da nicht jede Task-Instanz zwingend einen eigenen Thread zugewiesen bekommt. Die Zuweisung und Steuerung der vorhandenen Threads übernimmt die Concurrent Runtime. Sie überwacht und verwaltet die Anzahl aktiver Threads. Dabei achtet sie darauf, nicht das System mit zu vielen gleichzeitig aktiven Threads zu überlasten. Aktuell nicht benötigte Threads werden nicht sofort entfernt, sondern später wiederverwendet. Dieses Vorgehen verbessert die Systemperformance, da nicht immer neue Thread-Instanzen erstellt werden müssen. Die Concurrent Runtime ist daher die alleinige Instanz, die darüber entscheidet, wie viele Threads vorhanden und aktiv sind. Die Anzahl der Task-Instanzen spielt dabei daher zunächst keine Rolle.

HINWEIS: Da die Concurrent Runtime die optimale Anzahl an Threads verwaltet, sollten keine eigenen Thread-Instanzen erstellt werden. Wird dies nicht beachtet, binden die manuell erstellten Threads ebenfalls Ressourcen. Diese gebunden Ressourcen sind der Concurrent Runtime nicht bekannt und können somit bei der Ressourcenplanung nicht berücksichtigt werden.

Bindung von Rechenzeit

Werden durch einen Prozess gleichzeitig sehr viele Threads erzeugt, bindet das Programm unverhältnismäßig viel Rechenzeit an sich. Somit stehen anderen Programmen weniger CPU-Ressourcen zur Verfügung. Das ist vor allem unfair, da sehr viel Rechenzeit verschwendet wird, um die Threads durch den Betriebssystem-Scheduler ein- und auszulagern. Aus diesem Grund sollte, wie im oberen Abschnitt bereits erläutert wurde, auf die manuelle Thread-Erstellung verzichtet werden.

Die drei aufgeführten Punkte müssen nicht zwingend auftreten, wenn Threads korrekt und kontrolliert eingesetzt werden. Allerdings bedarf es eines relativ hohen Aufwands, wenn all die Punkte optimal gelöst werden sollen. Gerade die optimale Verwaltung der Systemressourcen stellt eine erhebliche Herausforderung dar und darf nicht unterschätzt werden. In Zukunft kann und sollte daher immer auf die Klasse *Task* zurückgegriffen werden. Diese löst – in Zusammenspiel mit der Concurrent Runtime – die genannten Probleme und sorgt für eine optimale Ausnutzung der vorhandenen CPU-Ressourcen.

8.2 Benutzerdefinierte Aufgabenplaner

In Abschnitt 8.1 wurde die Arbeitsweise und Aufgabe des Task Schedulers innerhalb der Task Parallel Library beschrieben und erläutert, welche Einflussmöglichkeiten – z. B. über die *ParallelOptions* – möglich sind. Reichen diese vorhandenen Änderungsmöglichkeiten nicht aus, besteht die letzte Möglichkeit darin, einen eigenen Task Scheduler umzusetzen. Das ist nicht trivial und sollte auch nur geschehen, wenn keine anderen Lösungsalternativen vorhanden sind. Im Grunde besteht ein benutzerdefinierter – wie auch die eingebauten – Aufgabenplaner lediglich aus einer Klasse, die von der abstrakten Klasse *TaskScheduler* – aus dem Namensraum *System.Threading.Tasks* – ableiten muss. Die Klasse enthält alle wesentlichen Methoden in abstrakter Form, die mit eigener Logik überschrieben werden müssen. Konkret handelt es sich um die folgenden Methoden:

- *GetScheduledTasks*
- *QueueTask*
- *TryExecuteTaskInline*

Der Methode *QueueTask* wird ein neuer Task zu Ausführung übergeben, der zunächst einer Warteschlange hinzugefügt und verarbeitet wird, sobald Ressourcen verfügbar werden. Die Methode *GetScheduledTasks* gibt eine Liste der aktuell eingeplanten Task-Objekte zurück. Die Methode *TryExecuteTaskInline* versucht, einen Task auf dem aktuellen Thread auszuführen.

Die wesentliche Schwierigkeit bei der Umsetzung eines eigenen Aufgabenplaners liegt aber nicht in der Implementierung der oben genannten

Methoden, sondern darin, die eigenen notwendigen Algorithmen effizient und geschwindigkeitsoptimiert zu realisieren. Da der Aufgabenplaner innerhalb der Task Parallel Library einen zentralen Verkehrsknoten darstellt, wirken sich eventuelle Engpässe hier sofort negativ auf die gesamte Performance aus. Die überschriebenen Methoden und zugehörigen (Steuerungs-)Algorithmen müssen daher so effizient wie möglich realisiert werden. Innerhalb des Aufgabenplaners sollten daher auch jegliche I/O-Operationen und sonstige Zugriffe auf ausgelagerte Systeme unterbunden werden.

8.2.1 Task mit Priorität

Anhand eines konkreten Beispiels wird im Folgenden die Umsetzung und Realisierung eines eigenen Aufgabenplaners demonstriert. Der neue *Task Scheduler* soll die Möglichkeit bieten, auf die *Priority*-Eigenschaften der Threads Einfluss zu nehmen, die zur Ausführung der eingeplanten Task-Instanzen verwendet werden. Bei der Verwendung der eingebauten Aufgabenplaner ist das nicht möglich. Der erste Schritt zum eigenen Aufgabenplaner besteht darin, eine neue Klasse mit entsprechendem Namen anzulegen und sie von der abstrakten Klasse *TaskScheduler* ableiten zu lassen. Danach ist, wie in Listing 8.6 zu sehen, die Grundstruktur des Task Schedulers angelegt.

```
public class TaskThreadPriority : TaskScheduler, IDisposable
{
    protected override IEnumerable<Task> GetScheduledTasks()
    {
        throw new NotImplementedException();
    }
    protected override void QueueTask(Task task)
    {
        throw new NotImplementedException();
    }
    protected override bool TryExecuteTaskInline
        (Task task, bool taskWasPreviouslyQueued)
    {
        throw new NotImplementedException();
    }
}
```

```
}  
}
```

Listing 8.6: Aufgabenplaner in der Grundstruktur

Ausgehend von der erstellten Klasse können nun die notwendigen Implementierungen durchgeführt werden. Als Erstes müssen die benötigten Variablen und Initialisierungsmethoden definiert werden. Diese Erweiterungen sind in Listing 8.7 abgebildet.

```
public class TaskThreadPriority : TaskScheduler, IDisposable  
{  
    private ThreadPriority priority = ThreadPriority.Normal;  
    private ApartmentState apState = ApartmentState.MTA;  
    private Lazy<Thread[]> availableThreads = null;  
    private BlockingCollection<Task> queuedTasks;  
  
    [ThreadStatic]  
    private static bool currentThreadIsActive;  
    private int degreeOfParallelism = 1;  
  
    public TaskThreadPriority(ThreadPriority priority,  
                             int degreeOfParallelism)  
    {  
        this.degreeOfParallelism = degreeOfParallelism;  
        this.priority = priority;  
        queuedTasks = new BlockingCollection<Task>();  
  
        InitializeScheduler(this.degreeOfParallelism, this.  
                           priority);  
    }  
  
    private void InitializeScheduler(int parallelismLevel,  
                                     ThreadPriority priority)  
    {  
        availableThreads = new Lazy<Thread[]>(() =>  
        {  
            Thread[] threads = new Thread[parallelismLevel];  
            for (int i = 0; i < threads.Length; i++)  
            {
```

```
        threads[i] = new Thread(ProcessingRequests);
        threads[i].IsBackground = true;
        threads[i].Priority = priority;
        threads[i].SetApartmentState(apState);
        threads[i].Start();
    }
    return threads;
});
}
...
}
```

Listing 8.7: Variablen und Initialisierung

Wie in Listing 8.7 zu erkennen, können dem Konstruktor die wesentlichen Informationen, die zur Anlage des eigenen Task Schedulers notwendig sind, übergeben werden. Der Parameter *priority* legt die gewünschte Priorität für die späteren Threads fest, und über den Parameter *degreeOfParallelism* kann die maximale Anzahl aktiver Threads eingestellt werden. Gemäß dem übergebenen Parallelisierungslevel werden innerhalb des Konstruktors Thread-Instanzen erstellt und innerhalb eines Arrays gespeichert. Die gesamte Anlage der Threads wird innerhalb eines Lazy-Konstrukts durchgeführt. Somit erfolgt die Anlage erst bei dem ersten Zugriff auf das Array, d. h. die Threads werden nur angelegt, wenn der Aufgabenplaner tatsächlich zum Einsatz kommt. Bei der Anlage des Threads wird dann die definierte Priority-Einstellung übernommen. Neben anderen Einstellungen wird den Threads ein Thread Delegate übergeben, der auf die Methode *ProcessRequests* verweist. Hierbei handelt es sich um die Methode, die durch den Thread ausgeführt werden soll. Die Implementierung der Methode ist übersichtlich und in Listing 8.8 zu sehen.

```
private void ProcessingRequests()
{
    foreach (var task in queuedTasks.
        GetConsumingEnumerable())
    {
        currentThreadIsActive = true;
    }
}
```

```
        base.TryExecuteTask(task);
        currentThreadIsActive = false;
    }
}
```

Listing 8.8: Die Methode „ProcessRequests“

Die Methode besteht im Grunde nur aus einer *foreach*-Schleife, die kontinuierlich die Task-Warteschlange auf neue Task-Instanzen hin überprüft. Als Warteschlange wird eine *BlockingCollection* verwendet. Diese ist zu finden in dem Namensraum *System.Collections.Concurrent* und gehört zu den neuen, unter .NET 4.0 eingeführten thread-safe-Auflistungsklassen. Ein separater Schutz gegen gleichzeitige/konkurrierende Zugriffe ist also bereits gewährleistet. Der eigene Aufgabenplaner setzt mithilfe der *BlockingCollection* das typische Entwurfsmuster Producer/Consumer um. Der Producer ist in diesem Fall die Anwendung, die den Scheduler verwendet, die Consumer sind die einzelnen Threads, die anstehende Tasks verarbeiten. Ob ein Thread aktuell aktiv ist, wird über die *bool*-Variable *currentThreadIsActive* signalisiert, eine Variable, die mittels *ThreadStatic*-Attribut für jeden Thread neu initialisiert und unabhängig verwendbar ist (siehe dazu auch Kapitel 3.6, Listing 3.16). Die eigentliche Ausführung der Task-Instanz wird an die geerbte Methode *TryExecuteTask* weitergegeben. Damit überhaupt Tasks in die Warteschlange geschrieben werden können, muss auch die Methode *QueueTask* entsprechend implementiert werden. Auch diese Methode ist nicht sonderlich umfangreich, wie in Listing 8.9 erkennbar.

```
protected override void QueueTask(Task task)
{
    var forceIgnore = availableThreads.Value;
    queuedTasks.Add(task);
}
```

Listing 8.9: Die Methode „Queue Task“

Bevor ein Task in die Warteschlange geschrieben und für die Ausführung vorgesehen werden kann, muss sichergestellt werden, dass die Threads

bereit zur Ausführung sind. Dazu wird auf die Eigenschaft `Value` des `Lazy`-Konstrukts zugegriffen (vgl. dazu auch den `Lazy`-Datentyp in Kapitel 6.10.1). Nach diesem erstmaligen Zugriff sind die Threads angelegt und warten auf neue Aufgaben. Anschließend wird der übergebene Task der Warteschlange über `Add` hinzugefügt. Eine separate Sperre (`lock`) ist für den Zugriff auf die Warteschlange nicht notwendig, da die verwendete `BlockingCollection` für alle öffentlichen und geschützten Mitglieder `Thread Safety` garantiert. Die noch verbleibende Methode `TryExecuteTaskInline` prüft, ob ein Task auf dem gleichen Thread ausgeführt werden kann. Die dazu notwendige Prüflogik ist in Listing 8.10 zu sehen.

```
protected override bool TryExecuteTaskInline(Task task,
bool pQueued)
{
    if (currentThreadIsActive)
    {
        if (pQueued)
            TryDequeue(task);
        return base.TryExecuteTask(task);
    }
    return false;
}
```

Listing 8.10: Die Prüfmethode „`TryExecuteTaskInline`“

Innerhalb der Methode wird zunächst geprüft, ob der aktuelle Thread aktiv ist. Wenn ja, wird der eventuell zuvor geplante Task wieder aus der Warteschlange entfernt. Danach wird der Task mithilfe der Methode `TryExecuteTask` aus der übergeordneten Klasse zur Ausführung gebracht. Ist der zugehörige Thread nicht aktiv, kann der Task nicht eingeplant werden und die Methode gibt `false` zurück.

Nun fehlt nur noch die Methode `GetScheduledTasks`, die lediglich die aktuell eingeplanten Aufgaben (Tasks) zurückgibt. Diese Methode dient lediglich für Debug-Zwecke und wird nicht für den logischen Ablauf des Aufgabenplaners verwendet. Listing 8.11 zeigt die umgesetzte Methode. Wie ersichtlich ist, gibt die Methode lediglich die Liste der Aufgaben als `Array` zurück.

```
protected override IEnumerable<Task> GetScheduledTasks()
{
    return queuedTasks.ToArray();
}
```

Listing 8.11: Die Methode „GetScheduledTasks“

Somit ist die Implementierung eines eigenen Aufgabenplaners abgeschlossen, und der neue Aufgabenplaner kann verwendet werden.

8.2.2 Verwendung eigener Aufgabenplaner

In diesem Kapitel geht es darum, die Task Parallel Library mithilfe eines eigenen Aufgabenplaners zu verwenden. Dazu bietet die Task Parallel Library verschiedene Einstiegspunkte. Jedem erzeugten Task kann bei der Erstellung ein Aufgabenplaner mitgegeben werden. Am einfachsten kann das über die Fabrik-Methode *StartNew* realisiert werden, wie in Listing 8.12 zu sehen.

```
public void UsingTasksCustomScheduler()
{
    TaskThreadPriority taskPrio = new TaskThreadPriority
        (ThreadPriority.Lowest, 8);
    Task[] tasks = new Task[WORKLOAD];
    int i = 0;
    foreach (List<int> list in allLists)
    {
        tasks[i] = Task.Factory.StartNew(() =>
            { list.Sort(); },
            CancellationToken.None, TaskCreationOptions.None,
            taskPrio);
        i++;
    }
    Task.WaitAll(tasks);
}
```

Listing 8.12: Verwendung eines bestimmten Aufgabenplaners

Zunächst wird eine neue Instanz des eigenen Aufgabenplaners erstellt, die dann bei jedem Aufruf von *StartNew* an die neu zu erstellende Task-Instanz übergeben wird. Auch andere TPL-Objekte erlauben die Definition eines Aufgabenplaners, der zur Ausführung verwendet werden soll. Die *Parallel.XXX*-Schleifen erlauben ebenfalls über die Eigenschaft *ParallelOptions* die Spezifikation eines speziellen Aufgabenplaners für die Ausführung der Schleife.

8.2.3 Weitere Aufgabenplaner

Da die Umsetzung eigener Aufgabenplaner nicht trivial ist, lohnt sich ein Blick auf die „Samples for Parallel Programming with the .NET Framework“. Dort finden sich zahlreiche Beispiele, wie die TPL eingesetzt werden kann, darunter auch einige Beispielimplementierungen für Aufgabenplaner: <http://code.msdn.microsoft.com/ParExtSamples>.

8.3 TPL Dataflow (TDF)

Die Task Parallel Library hat in Bezug auf die Parallele Programmierung schon jetzt erhebliche Verbesserungen in das .NET Framework eingeführt. Das reichhaltige API erleichtert die Umsetzung und Steuerung paralleler Abläufe, wie in den vergangenen Kapiteln beschrieben. Dennoch kann die jetzige TPL natürlich noch nicht alle Anforderungen abdecken. Typische Probleme paralleler Abläufe, wie Race Condition, Deadlocks und Shared States, um nur einige zu nennen, müssen weiterhin manuell vom Entwickler berücksichtigt werden. Die TPL stellt allerdings dafür bessere Mechanismen bereit und bietet dem Entwickler eine bessere Unterstützung. Neben diesen technischen Einschränkungen beinhaltet die TPL bisher noch keine eingebauten Konzepte bzw. APIs, die eine agentenbasierte Umsetzung paralleler Abläufe ermöglichen. Auch Problemfelder, die optimal mit einem Nachrichtenaustauschmodell (Message-Passing Paradigm) gelöst werden können, sind mit dem aktuellen TPL API schwer umsetzbar. Aber gerade die beiden zuletzt aufgeführten