entwickler.press

Vollständig aktualisierte und erweiterte 2. Auflage

Parallel Computing in .NET

Multicore-Programmierung von .NET 2.0 bis 4.0

Marc André Zhou

Marc André Zhou

Parallel Computing in .NET

Multicore-Programmierung von .NET 2.0 bis 4.0

entwickler.press

Marc André Zhou
Parallel Computing in .NET
ISBN: 978-3-86802-072-4

Ein Imprint der Software & Support Media GmbH

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Ihr Kontakt zum Verlag und Lektorat:
Software & Support Media GmbH
entwickler.press
Geleitsstr. 14
60599 Frankfurt am Main
Tel.: +49 (0)69 630089-0
Fax: +49 (0)69 930089-89
lektorat@entwickler-press.de
http://www.entwickler-press.de

Lektorat: Sebastian Burkart
Korrektorat: Katharina Klassen und Lisa Pychlau
Satz: Pobporn Fischer
Belichtung, Druck & Bindung: M.P. Media-Print Informationstechnologie GmbH, Paderborn

Inhaltsverzeichnis

1 Einleitung

Das Fachgebiet der Informatik ist vergleichbar mit der schillernden Modewelt. In regelmäßigen Zeitperioden kommen neue so genannte „Hypethemen" auf. Einige davon entwickeln sich zu wirklichen Themen, die im Enterprise-Einsatz verwendet werden können, andere wiederum verschwinden so schnell von der Bildfläche wie sie gekommen sind. Seit einiger Zeit trifft man auch immer öfter auf den eigentlich alten Begriff „Parallel Computing". Auf zahlreichen IT-Konferenzen und in einigen IT-Zeitschriften ist dieses Thema wieder aktuell. Nun stellt sich natürlich die Frage: Handelt es sich dabei auch um ein kurzfristiges Hypethema oder lohnt sich eine nähere Beschäftigung mit diesem Thema? Da Sie bereits dieses Buch in den Händen halten, sind Sie wahrscheinlich schon davon überzeugt, dass dieses Themenfeld zukünftig noch an Bedeutung gewinnen wird. Um diese Einstellung zu untermauern, werden in den folgenden Absätzen die Gründe erläutert, warum Parallel Computing in der Zukunft immer wichtiger wird.

1.1 Für wen ist dieses Buch?

In erster Linie richtet sich das Buch an .NET-Softwareentwickler und Softwarearchitekten. Dabei spielt es keine Rolle, ob ASP.NET, Silverlight oder Desktopanwendungen realisiert werden – Parallel Computing kann in allen Anwendungstypen angewendet werden. Aber auch für IT-Projektleiter lohnt sich eine Auseinandersetzung mit diesem komplexen Themenfeld, da durch den Einsatz von Parallel Computing Systeme teilweise komplizierter werden und daher eine längere Entwicklungszeit benötigen. Softwareentwickler sollten ausreichend Kenntnisse über das .NET Framework besitzen und bereits mehrere (sequenzielle) Systeme realisiert haben. Für Softwaretester und Qualitätsmanager ist es eben-

falls wichtig zu verstehen, wie Parallel Computing angewendet wird und welche Auswirkung das auf das Laufzeitverhalten eines Programms hat. Als Programmiersprache wird hauptsächlich C# verwendet, in einigen Ausnahmenfällen wird auf C++ zurückgegriffen, wenn keine entsprechende .NET API (Managed API) verfügbar ist. Als Entwicklungsumgebung wird Visual Studio 2010 Professional Beta 2 unter Windows 7 RC (Release Candidate) 1 verwendet.

1.2 Ziele des Buchs

Das Buch soll in erster Linie einen Überblick über vorhandene Techniken unter .NET geben um Parallel Computing umzusetzen. Im zweiten Kapitel wird zunächst das nötige Theoriewissen zusammengefasst. In den weiteren Kapiteln werden die unter .NET verfügbaren APIs beschrieben, um Parallel Computing umzusetzen. Ausgehend von der .NET-Version 2.0 werden alle notwendigen Anweisungen, Konstrukte und Schlüsselwörter erläutert. Neben den reinen technischen Betrachtungen kommen auch Architekturüberlegungen nicht zu kurz. Ein eigenes Kapitel beschreibt detailliert die Neuerungen unter .NET 4.0 und geht vertiefend auf die Task Parallel Library (TPL) sowie PLINQ ein. Abschließend werden die Neuerungen erläutert, die Visual Studio 2010 im Bezug zur Fehleranalyse paralleler Systeme bietet. Kurzum: Ziel des Buchs ist es, die Grundlagen von Parallel Computing zu vermitteln und aufzuzeigen, welche Hilfsmittel das .NET Framework zur erfolgreichen Realisierung zur Seite stellt.

Am Ende widmet sich ein Kapitel den Problemen heutiger Programmiersprachen und geht auf die spezifischen Probleme ein, die auch nicht mit zukünftigen CLR-Versionen gelöst werden können. Abschließend werden Lösungsansätze vorgestellt, wie die speziellen Probleme nebenläufiger Ausführungen gelöst werden können.

1.3 Parallel Computing: Warum?

Warum Parallel Computing in Zukunft auch für Desktop bzw. typische Webapplikationen von Bedeutung sein wird, ist die zentrale Frage, die nun ausführlich behandelt und beantwortet werden soll. Der Begriff und die Verwendung von Parallel Computing ist nicht wirklich neu. Schon zu Beginn des Computerzeitalters wurde bereits Parallel Computing angewendet. So standen z. B. in Universitäten Großrechner zur Verfügung. Die Rechenleistung dieser Maschinen wurde priorisiert und auf bestimmte Personengruppen aufgeteilt. So erhielten Professoren mehr Recheneinheiten (oder auch Zeitscheiben) als Studenten. Dabei handelte es sich noch nicht um echtes Parallel Computing, vielmehr kann es als eine Vorstufe betrachtet werden. Wichtig an dieser Stelle ist nur zu verstehen, welches Ziel dieser Ansatz verfolgte. Dieser bestand darin, die Prozessorleistung optimal zu nutzen. Erfolgten zunächst Einteilungen und Zuweisungen der Zeitscheiben manuell, floss diese Idee in spätere Hard- sowie Softwarearchitekturen ein. Mithilfe von Multitasking wurde eine automatische Aufteilung der Prozessorzeit realisiert. Multitasking ist hardwareseitig auf so genannte Interrupts angewiesen. Wartet z. B. ein Programm auf eine Tastatureingabe, kann in der Zwischenzeit die Rechenleistung an einen anderen Prozess abgegeben werden. Sobald der Benutzer eine Taste drückt, wird das dem Betriebssystem durch das Auslösen eines Interrupts mitgeteilt. Der unterbrochene Prozess wird daraufhin wieder aktiviert und verarbeitet die Tastatureingabe. Ausführlichere Informationen zu verschiedenen Multitaskingversionen sind in Kapitel 2 zu finden. Multitaskingsysteme haben eine erhebliche Leistungssteigerung erzielt, von denen alle Softwareentwickler in der Vergangenheit profitiert haben. Um Multitasking in eigenen Programmen verwenden zu können, muss der Entwickler keinerlei Vorkehrungen während der Implementierung treffen. Ein Anwendungssystem wird synchron realisiert, alles Weitere wird automatisch vom Betriebssystem koordiniert. Dabei ist die Performanz eines multitaskingbasierten Systems von der Taktfrequenz der zentralen Recheneinheit (Central Processing Unit – CPU) abhängig. Je höher die Taktfrequenz ist, umso schneller ist die Verarbeitung der Befehle, und somit steigt die Gesamtleistung des Systems.

1.3.1 CPU: Umkehrung der Entwicklung

Alle modernen Betriebssysteme setzen Multitasking ein, um eine optimale Ausnutzung der Prozessorzeit zu erreichen. Wie bereits beschrieben, ist die Geschwindigkeit und die Systemleistung von der Taktfrequenz des Prozessors abhängig. Nach dem bekannten Gesetz von Gordon Moore (Moores Law), verdoppelt sich die Leistung integrierter Schaltkreise alle 18 bis 24 Monate. Abbildung 1.1 verdeutlicht die Entwicklung der vergangenen Jahre.

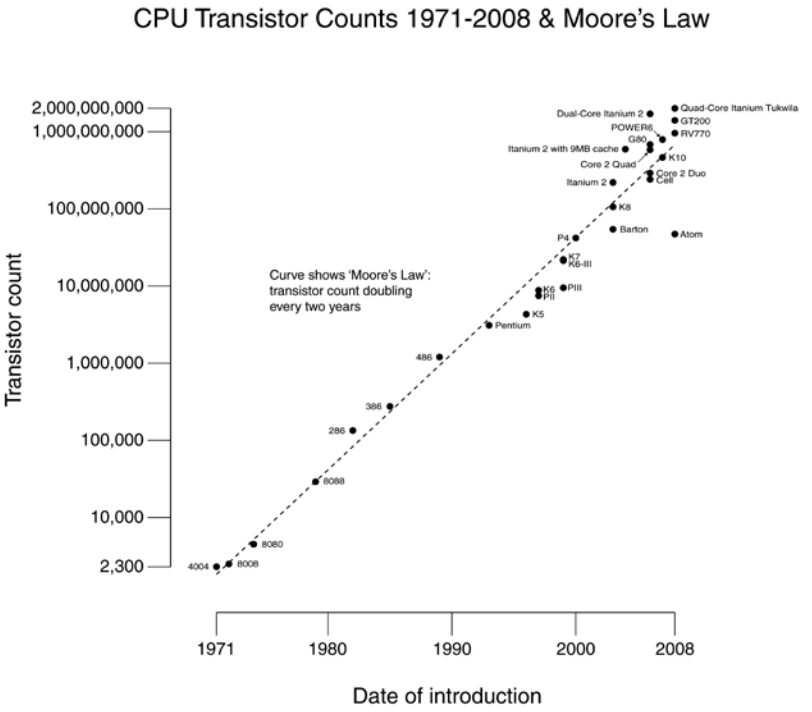

Abbildung 1.1: CPU-Entwicklung der vergangenen Jahre (Quelle: wikimedia.org)

Von dieser verbesserten Geschwindigkeit profitierten automatisch alle Anwendungssysteme, die unter einer neuen CPU-Generation betrieben wurden. Anwendungssysteme, die zur Zeitsteuerung bzw. zur Berechnung von Verzögerungen die Taktfrequenz nutzten, kamen auf neuer Hardware (CPU) aus dem Schritt. Dieses Problem betraf z. B. Spiele, die aufgrund der Ablaufgeschwindigkeit nicht mehr steuerbar waren. Dieses Problem wurde frühzeitig erkannt. Mittlerweile wird zur Steuerung nicht mehr die Taktfrequenz verwendet, sondern es findet eine reale Zeitsteuerung statt. Von all diesen Geschwindigkeitsverbesserungen haben Softwareentwickler ohne eigenes Mitwirken automatisch profitiert. Nun hat sich allerdings gezeigt, dass eine Erhöhung der Taktfrequenz aus technischer und wirtschaftlicher Sicht nicht mehr leicht möglich ist. Wirtschaftlich sprechen die Kosten gegen eine Erhöhung der Taktfrequenz. Technisch betrachtet können folgende Aspekte genannt werden, die gegen eine reine Erhöhung der Taktfrequenz sprechen:

- Physikalische Übertragungsgeschwindigkeit von Daten
- Signallaufzeiten
- Verkleinerungsgrenzen technischer Bauteile
- Wärmeentwicklung
- Baugröße

Innerhalb einer CPU werden Daten über einen internen Datenbus transportiert. Die Geschwindigkeit einer CPU ist direkt von der Transportgeschwindigkeit der Daten abhängig. Das Maximum an Geschwindigkeit ist durch die Lichtgeschwindigkeit (30cm/ns) definiert. Innerhalb integrierter Schaltkreise kommen in der Regel Kupferleitbahnen zum Einsatz, dessen maximale Datentransportgeschwindigkeit 3cm/ns beträgt. Außerdem müssen Daten teilweise in Speicherregister abgelegt werden, dazu müssen diese zum Speicher transportiert werden. Ist der Speicher nicht auf kurzem Weg erreichbar, führt das ebenfalls zu ungewollten Verzögerungen. Ebenso stößt die minimale Verkleinerung von technischen Bauteilen, z. B. Transistoren, an technische Grenzen. Ein weiteres Problem ist die entstehende Wärme, die durch mehr Bauteile auf einer kleineren

Fläche erzeugt wird. Um eine bessere Leistungsfähigkeit zu erreichen, sollte die Baugröße einer CPU nicht vergrößert werden. Innerhalb eines Desktop-PCs könnten die beiden letzten Punkte gelöst werden. Durch genügend große Kühlkörper kann die Wärme optimal abgeleitet werden. Probleme sind eher in mobilen Geräten zu sehen, z. B. Laptops oder Mobilfunkgeräte. Mobile Geräte werden immer leistungsfähiger und benötigen daher leistungsstarke Prozessoren. Da die Erhöhung der Taktfrequenz problematisch ist, gehen die Chiphersteller einen anderen Weg. Um die Leistungsfähigkeit einer CPU zu verbessern, können mehrere CPU-Kerne innerhalb einer CPU integriert werden. Die so entstehenden Multi-core CPUs, z. B. Intels Dual-core- oder Quad-core-Prozessor, haben eine höhere Leistungsfähigkeit. Die Fehler! Verweisquelle konnte nicht gefunden werden. verdeutlicht die beiden Möglichkeiten der Leistungsverbesserung: Zum einen kann die Taktfrequenz beschleunigt werden, das führt zu einer verbesserten Leistungskennzahl oder aber die Anzahl separater Kerne wird erhöht. Wie an der aktuellen Prozessorgeneration zu erkennen ist, hat man sich für Multi-core CPUs entschieden. Schon heute beinhalten die meisten Desktop-PCs einen Dual-core-(Zwei-Kerne-CPU) oder sogar einen Quad-core-(Vier-Kern-CPU-)Prozessor. In Zukunft werden CPUs mit noch mehr Kernen in Standard-PCs verbaut.

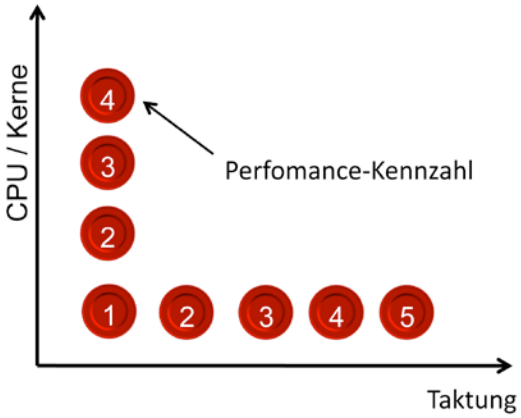

Abbildung 1.2: Leistungssteigerung einer CPU

1.3.2 Multi-core-Prozessoren

Die veränderte CPU-Architektur, die durch mehrere separate Kerne geprägt ist, stellt Softwareentwickler vor eine neue Herausforderung. In der Vergangenheit profitierte jedes sequenzielle Programm automatisch von einer CPU mit erhöhter Taktfrequenz. Neue Hardware versprach somit gleichzeitig eine verbesserte Leistungsfähigkeit der betriebenen Anwendungssoftware. Im Multi-core-CPU-Zeitalter profitiert nicht jedes Programm automatisch von einer neuen CPU mit mehr Kernen. Teilweise kann der Einsatz einer neuen CPU sogar das Gegenteil bewirken und Anwendungssysteme werden langsamer. Jetzt muss die Frage erlaubt sein: Wie ist das möglich? Trotz mehrerer Kerne keine Leistungsverbesserung? Die Frage ist je nach Kontext mit Ja oder Nein zu beantworten. Aus der Sicht der Hardware und den dort gemessenen Leistungsmerkmalen ist das System durch den Einsatz eines Multi-core-Prozessors performanter geworden. Leider überträgt sich diese Leistung nicht automatisch auf die betriebenen Anwendungssysteme. Die meisten heute betriebenen Desktopanwendungen sind für eine Single-core CPU (CPU mit einem Kern) ausgelegt. Diese Systeme sind sequenziell programmiert und besitzen meist nur einen Thread (Programmfaden). Somit nutzen diese Systeme auch auf einer Multi-core CPU nur einen Kern, alle anderen Kerne bleiben unbenutzt. An dieser Stelle ist der Anwendungsentwickler gefragt. Dieser muss explizit dafür sorgen, dass alle verfügbaren Kerne mit Arbeit versorgt werden. Die Lösung dafür lautet: Parallel Computing.

1.3.3 Nutzung von Multi-core-Prozessoren

Die letzten Abschnitte haben die Entwicklung der Prozessoren und die zukünftige Entwicklung aufgezeigt. Heutige Anwendungssysteme sind jedoch meist sequenziell umgesetzt und können daher nur ein CPU-Kern nutzen. Das bedeutet, dass jeder zusätzliche Kern nicht verwendet wird. Anhand eines einfachen sequenziellen Sortieralgorithmus kann das demonstriert werden. Listing 1.1 zeigt einen einfachen Bubblesort: Dieser Algorithmus wird ausgeführt um eine Liste von 1000 Zufallszahlen zu sortieren.

```
private static int[] BubbleSort(int[] toSort)
{
    int temp = 0;
    for (int i = 0; i < toSort.Length-1; i++)
    {
        for (int j = i + 1; j < toSort.Length; j++)
        {
            if (toSort[i] > toSort[j])
            {
                temp = toSort[i];
                toSort[i] = toSort[j];
                toSort[j] = temp;
            }
        }
    }
    return toSort;
}
```

Listing 1.1: Einfacher Bubblesort

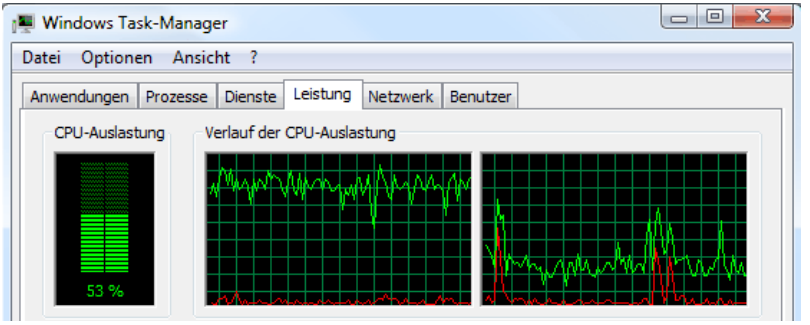

Abbildung 1.3: Task-Manager

Abbildung 1.3 zeigt – innerhalb des Task-Managers – die Auslastung des Systems während der Sortierung. Wie anhand der Abbildung gut zu erkennen ist, wird der erste Kern nicht zu 100 % genutzt und der zweite Kern nimmt kaum an der Verarbeitung teil. Wünschenswert an dieser Stelle wäre es, wenn beide Kerne gemeinsam zu 100 % die Sortierauf-

gabe erledigen. Auf einer Vier-Kerne-Maschine (Quad-core-Maschine) kommt es zu einem ähnlichen Verlauf, dort wird ebenfalls nur ein Kern beansprucht und die verbleibenden Kerne tragen nichts zum Arbeitsfortschritt bei.

Ziele des Parallel Computing

Hauptziel des Parallel Computings sollte es sein, die möglichen CPU-Ressourcen eines Systems optimal auszunutzen. So sollten auf einer Dual-core Maschine alle verfügbaren CPU-Kerne bei der Abarbeitung einer Aufgabe mit einbezogen werden. Erfolgreich eingesetztes und implementiertes Parallel Computing kann direkt gemessen werden. Ein parallelisiertes System oder Algorithmus muss schneller sein als das entsprechend synchrone Gegenstück.

1.4 Grundlagen

Nachdem nun die Wichtigkeit von Parallel Computing beschrieben und erläutert wurde, geht es nun darum, die notwendigen Grundlagen zu erarbeiten und den Begriff „Parallel Computing“ zu definieren.

Da in der Vergangenheit im Desktopbereich eher Single-core-Prozessoren zum Einsatz kamen, sind alle darauf basierenden Systeme für diese Prozessorarchitektur optimiert. Meistens wird also die gesamte Arbeit innerhalb eines Prozesses in einem Haupt-Thread (Programmfaden) ausgeführt. Somit werden alle Anweisungen sequenziell der Reihe nach abgearbeitet. Der Übergang zu Parallel Computing ist theoretisch relativ simpel. Anstatt das Anweisungen nacheinander ausgeführt werden, muss es möglich werden, gleichzeitig z. B. zwei Anweisungen auszuführen. Die in Abbildung 1.4 dargestellten Diagramme verdeutlichen den Übergang von einer sequenziellen zu einer parallelen Verarbeitung.

Im ersten Diagramm wird pro Zeitzyklus nur eine Anweisung ausgeführt, im zweiten Diagramm werden pro Zeitzyklus zwei Anweisungen parallel ausgeführt. Um das zu erreichen, muss die entsprechende

Hardwarebasis vorhanden sein. Eine echte parallele Verarbeitung ist nur auf einem System mit mehreren unabhängigen CPU-Kernen möglich. Besteht das Zielsystem nur aus einem CPU-Kern, ist keine parallele Verarbeitung möglich. Parallele Systeme sind im Großrechnerbereich nichts Neues und werden schon lange eingesetzt. Damit die Prozessorleistung optimal ausgenutzt wird, existieren spezielle Betriebssysteme für diese Maschinen. Hauptsächlich werden diese Maschinen zur Berechnung komplexer Szenarien eingesetzt, z. B. in der Wetterprognose. Eine weitere Art von Parallel Computing besteht durch die Kopplung von Maschinen über ein Netzwerk bzw. über das Internet. Ein bekanntes Projekt ist SETI (Search for Extraterrestrial Intelligence). Die Arbeitsweise solcher Rechnerverbunde ist relativ simpel: Auf den Clientrechnern wird ein kleines Programm installiert, das Arbeitsaufgaben vom Host-Rechner zugewiesen bekommt. Befindet sich der Clientrechner im *Idle* Mode und hat somit Rechenzeit verfügbar, wird die Aufgabe verarbeitet. Wurde die Aufgabe komplett gelöst, wird das Ergebnis zurück an den Host gesendet. Dieses Buch behandelt die Möglichkeiten der Parallelisierung innerhalb eines Anwendungssystems, das auf einem Mehrkernprozessorsystem betrieben wird. Dabei treten spezielle Probleme auf, die im weiteren Verlauf erläutert werden.

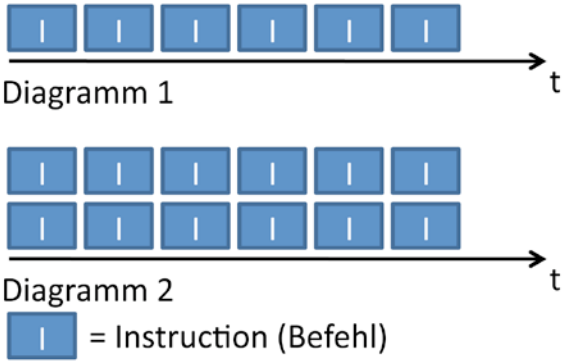

Abbildung 1.4: Sequenzielle vs. parallele Ausführung

1.4.1 Definition: Parallel Computing

Parallel Computing ist nichts Neues und es existieren unterschiedliche Ausprägungen und Umsetzungsvarianten. Hier soll zunächst eine Definition erfolgen, welche Art von Parallel Computing in diesem Buch behandelt wird. Befragt man die Onlineenzyklopädie Wikipedia zu dem Begriff „Parallel Computing“, erhält man Auszugsweise folgende Informationen:

> „Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (“in parallel”). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.
>
> Parallel computers can be roughly classified according to the level at which the hardware supports parallelism – with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.
>
> Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically one of the greatest obstacles to getting good parallel program performance. The speed-up of a program as a result of parallelization is governed by Amdahl's law.”
>
> [Quelle: http://en.wikipedia.org/wiki/Parallel_computing]

Die Wikipedia-Definition ist sehr umfassend und beschreibt alle Arten von Parallel Computing. Der Schwerpunkt dieses Buchs sind die folgenden Aspekte des Parallel Computings:

- Nutzung von Mehrkernprozessoren
- Daten- und Task-Parallelismus
- Probleme paralleler Systeme
- Erfolgsmessung paralleler Lösungen

Aus diesen Punkten kann folgende Arbeitsdefinition für Parallel Computing festgelegt werden:

> Parallel Computing beschäftigt sich unter .NET mit der Verteilung von großen Datenmengen auf vorhandene CPU-Ressourcen sowie der optimalen Arbeitsverteilung (Tasks) auf die vorhandenen Ressourcen. Dabei geht es darum, vorhandene Probleme parallel und nebenläufig abzuarbeiten. Typische Probleme (Race Conditions, Deadlocks usw.), die dabei auftreten, werden erläutert und es werden Lösungsansätze erarbeitet. Die theoretische Betrachtung liefert Hinweise, um eine Erfolgsmessung durchzuführen.

Nicht behandelt werden parallele Ansätze, die auf einem Rechnerverbund (Cluster) basieren. Ebenfalls findet keine Betrachtung von Parallelisierungsmöglichkeiten auf Instruktions- sowie Bit-Level-Ebene statt.

1.4.2 Rechnerarchitekturen

Echte parallele Verarbeitung von Instruktionen kann nur auf einem System mit mindestens zwei unabhängigen CPU-Kernen umgesetzt werden. Das kann b durch die Architektur heutiger Rechneregründet werden. Die heute üblichen Rechner basieren auf der von-Neumann-Architektur. Benannt wurde die Architektur nach dem österreichisch-ungarischen Mathematiker John von Neumann. Von Neumann definierte im Jahr 1946 folgende sieben Prinzipien der Rechnerorganisation, die heute noch gültig und nur in leicht abgewandelter Form in aktuellen CPUs verwendet werden:

1. Der Rechner besteht aus vier Werken:
   - Haupt- bzw. Arbeitsspeicher für Programme und Daten
   - Leitwerk, interpretiert das Programm
   - Rechenwerk, das die arithmetischen Operationen ausführt
   - Ein-/Ausgabenwerk, das mit der Umwelt kommuniziert, und Sekundärspeicher, der als Langzeitspeicher fungiert
2. Die Struktur des Rechners ist vom bearbeiteten Problem unabhängig: Die Anpassung erfolgt durch Programmsteuerung.
3. Programme und Daten befinden sich im gleichen Hauptspeicher und können durch die Maschine verändert werden.
4. Der Hauptspeicher ist in Zellen gleicher Größe geteilt, die eindeutig durch Adressen angesprochen werden können.
5. Ein Programm besteht aus einer Folge von Befehlen (Instruktionen), die nacheinander also sequenziell ausgeführt werden.
6. Die Abfolge der Instruktionen kann durch Bedingungen verändert werden. So sind Sprünge zu anderen Adressen zulässig.
7. Die Codierung basiert auf Binärcodes, Zahlen werden dual dargestellt.

Der ausschlaggebende Punkt ist Nummer 5: Instruktionen werden nacheinander abgearbeitet. Dieses Verhalten ist hardwareseitig umgesetzt und kann auch mit softwaretechnischen Implementierungen nicht umgangen werden. Um Instruktionen echt parallel auszuführen, muss die Maschine mindestens zwei unabhängige CPU-Kerne besitzen. Außerdem muss auch das Betriebssystem in der Lage sein, die verfügbaren CPUs zu verwalten. Das erste Betriebssystem aus dem Hause Microsoft, das mehrere CPUs verwalten konnte, war Windows NT. Hardwareseitig wurden Mainboards mit mehreren CPU-Sockeln verwendet. Auf jeden Sockel wurde eine vollständige CPU verbaut. Abbildung 1.5 zeigt ein typisches Dual-Sockel-Mainboard aus einem Dell-PowerEdge-System. Wurde diese Hardware hauptsächlich in Serversysteme verbaut, so be-

sitzen auch schon heutige Büroarbeitsplatzrechner Mehrkernsysteme. Hierbei werden allerdings keine Mainboards mit zwei oder mehreren Sockeln eingesetzt, vielmehr besitzen moderne CPUs in der Regel schon zwei oder vier unabhängige Kerne (vgl. Intel Dual-core bzw. Intel Quad-core CPUs).

Abbildung 1.5: Dual-Sockel-Mainboard

Die aktuellen Desktop-CPUs der Firma Intel gehören zu der Klasse der Mehrkernprozessoren. Innerhalb dieser Prozessoren sind alle Bauteile, bis auf wenige Ausnahmen, doppelt – bei einem Dual-core-Prozessor – oder vierfach – auf einem Quad-core-Prozessor – vorhanden. In Abbildung 1.6 ist das Blockdiagramm eines Intel-Conroe-Dual-core-Prozessors dargestellt.

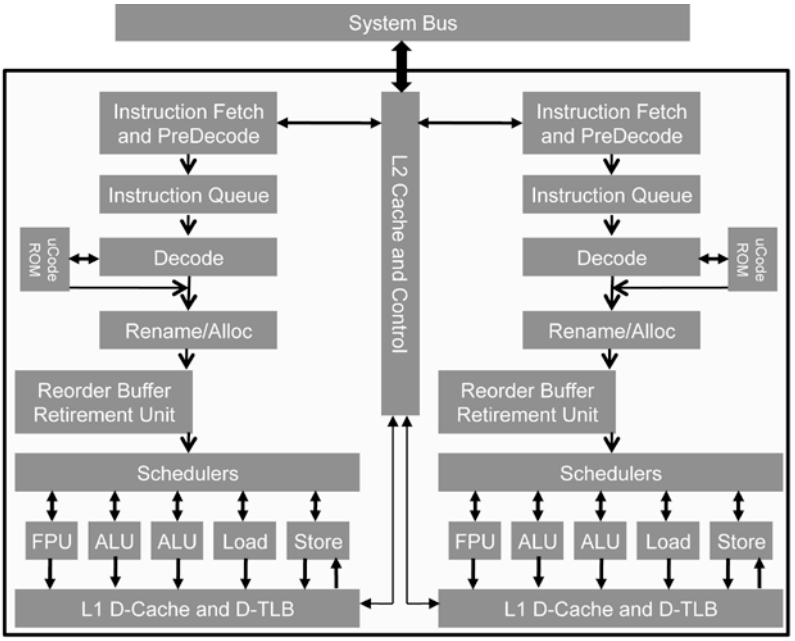

Abbildung 1.6: Intels Conroe-Dual-core-Prozessor (Blockdiagramm)

Wie dort gut erkennbar ist, befinden sich jeweils links und rechts vom L2-Cache separate Kerne. Als Vorstufe zu „echten" Multi-core-Prozessoren kann der Intel Pentium 4 mit Hyper-Threading-Technologie genannt werden. Mithilfe der Hyper-Threading-Technologie kann ein zweiter Kern innerhalb einer CPU simuliert werden. Bei der Hyper-Threading-Technologie werden hardwareseitig nicht komplette CPU-Kerne dupliziert, sondern nur einige Teile. Dazu gehören z. B. Registersätze und der Interrupt Controller. Single-core-Prozessoren mit Hyper-Threading-Technologie sind daher in der Lage, Multithreading hardwareseitig auszuführen und werden als Simultaneous-Multithreading-(SMT-) Prozessoren bezeichnet. Das Betriebssystem erkennt, bei einem Single-core-Prozessor mit Hyper-Threading zwei unabhängige Prozessoren. Zukünftige Multi-core-Prozessoren werden ebenfalls wieder die Hyper-

Threading-Technologie nutzen, um eine bessere Ausnutzung der Ressourcen zu erzielen. Eingesetzt bei ein einem Vier-Kern-Prozessor, führt das zu acht logischen Kernen.

Speichermodelle

Durch die erhöhte Anzahl an Kernen innerhalb einer CPU, muss zukünftig eine Änderung des Speicherzugriffs erfolgen. Derzeitige Intel-Multi-core-Prozessoren verwenden weiterhin das so genannte Uniform-Memory-Access-(UMA-)Verfahren, um auf den gemeinsamen Speicher zuzugreifen. Bei dieser Architektur verwaltet ein Memory Controller Hub (MCH) den Zugriff auf den Speicher. Der MCH ist dabei nicht Bestandteil des Prozessors, sondern dieser ist auf dem Mainboard untergebracht. Bei einem Dual-Sockel-Mainboard werden beide Prozessoren an diesen MCH angeschlossen. Durch diese Architektur sollte sichergestellt werden, dass jeder Prozessor jederzeit alle anstehenden Aufgaben durchführen kann. Dazu benötigen die Prozessoren zu jeder Zeit Zugriff auf alle aktuellen Daten des gemeinsamen Speichers. Da somit jeder Prozessor über Datenänderungen eines anderen Prozessors automatisch über den gemeinsamen Speicher informiert wird, spricht man auch von Cachekohärenz (CC-UMA: Cache Coherent). Ein erheblicher Nachteil dieser Architektur liegt in der Geschwindigkeit des Speicherzugriffs. Da alle Prozessoren über einen gemeinsamen MCH auf den Speicher zugreifen, bildet dieser einen Engpass im System. Um den Zugriff auf lokale und prozessnahe Daten zu verbessern, besitzt jeder Prozessor einen eigenen Cache, indem er Daten ablegen kann. Mit den Prozessoren, die auf der „Intel-Nehalem-Mikroarchitektur“ basieren, führt Intel die Non-Uniform-Memory-Access-(NUMA-)Technologie ein. Dabei ist der MCH im Prozessor integriert, jeder Kern besitzt einen eigenen MCH. Idealerweise benötigen die einzelnen Kerne zur Abarbeitung der Aufgabe nur Daten, die von anderen Kernen nicht benötigt werden. Ist das nicht der Fall, können diese via interner Prozessorkommunikation angefordert werden. Da dieser Informationsaustausch sehr ressourcenintensiv ist, sollte dieses wenn möglich vermieden werden. Eine optimale Speicherzuweisung kann jedoch nur in Kooperation mit dem Betriebssystem erfolgen. Intel setzt dazu bei Open Solaris einen Memory-Placement-Optimiza-