Inhalt

Inhalt I

Abbildungsverzeichnis VI

Tabellenverzeichnis VII

Abkürzungsverzeichnis VIII

1 Einleitung 11

1.1 Motivation 11

2 Grundlagen 12

2.1 Paarweise, ungeordnete Berechnungen 12

2.2 Begriffe 13

2.2.1 Berechnungselemente 13

2.2.2 Globale Daten 13

2.2.3 Core 13

2.2.4 Dummy 13

2.2.5 Latenz 13

3 Entwicklungsumgebung 14

4 Testumgebung 15

4.1 Betriebssystem 15

4.2 Prozessor 15

5 Gegenüberstellung der Algorithmen 16

6 Probleme bei der Verteilung von Berechnungen 17

6.1 Ressourcenzugriff 17

6.2 Synchronisation 17

6.3 Overhead 17

6.4 Mehrfachberechnung 17

7 Lösungsansätze 18

7.1 Validierung der Algorithmen 18

7.2 Locked Resource 19

7.2.1 Vorgehensweise 19

7.2.2 Deadlock-Gefahr 19

7.2.3 Diskussion 19

7.3 Round Robin Tournament Algorithmus 21

7.3.1 Vorgehensweise 21

7.3.2 Diskussion 22

7.4 Deepcopy 22

7.4.1 Diskussion 22

7.5 Divide and Conquer 23

8 Parallelisierungs-Systeme 24

8.1 Threadspawning 24

8.2 Actors 24

8.3 .NET Threadpool 25

9 Testsetup 26

9.1 Überblick 26

9.1.1 Initialisierung 26

9.1.2 Ablauf 26

9.2 Grundstruktur 28

9.3 Output-Validierung 29

9.3.1 Validierungs-Input 29

9.3.1.1 Struktur 29

9.3.1.2 Code 32

9.3.2 Test-Ablauf 33

9.3.2.1 Struktur 33

9.3.2.2 Code 34

9.4 Effizienz-Messung 35

9.4.1 Struktur 35

9.4.2 Code 36

9.4.3 Overhead-Messung 37

9.4.4 Zeitmessung mit fixierter Rechenzeit 37

9.5 Zeitmessung mit zufälliger Rechenzeit 38

9.5.1 Zeitmessung mit Auslastung 38

10 Systemabstraktion 39

10.1 Core-Pool 40

10.1.1 Berechnungsanweisung 40

10.1.2 Synchronisation 41

11 Verteilungsstrukturen 42

11.1 Grundstruktur 42

11.1.1 Struktur 42

11.1.2 Code 42

11.2 Core-Pool-Verteilung 44

11.2.1 Struktur 44

11.2.2 Code 44

11.3 Eigenständige Verteilung 45

11.3.1 Struktur 45

11.3.2 Code 46

11.4 Lock-Verteilung 47

11.4.1 Struktur 47

11.4.2 Code 47

11.5 Round Robin Tournament Matrix 50

11.5.1 Basis-Array 50

11.5.2 Array-Verschiebung 51

11.5.3 Matrix-Generierung 52

12 Algorithmus-Optimierung 54

12.1 Testroutinen 54

12.1.1 Validierung 54

12.1.2 Effizienz 57

12.1.2.1 Overhead 1 57

12.1.2.2 Overhead 2 57

12.1.2.3 Overhead 3 57

12.1.2.4 Fixierte Rechenzeit 1 59

12.1.2.5 Fixierte Rechenzeit 2 59

12.1.2.6 Zufällige Rechenzeit 1 59

12.1.2.7 Zufällige Rechenzeit 2 59

12.1.2.8 Auslastung 1 60

12.1.2.9 Auslastung 2 60

12.1.2.10 Auslastung 3 60

12.2 Single-Thread Referenz Algorithmus 61

12.2.1 Implementierung 61

12.2.2 Messergebnisse 62

12.2.2.1 Messung 62

12.2.2.2 Diskussion 62

12.2.2.3 Optimierung 62

12.3 Parallelisierung durch Locked Resource 63

12.3.1 Implementierung 63

12.3.2 Messergebnisse 64

12.3.2.1 Messung 64

12.3.2.2 Diskussion 64

12.3.2.3 Optimierung 65

12.4 Gleichmäßige Verteilung der Berechnungen 66

12.4.1 Implementierung 66

12.4.2 Messergebnisse 67

12.4.2.1 Messung 67

12.4.2.2 Diskussion 67

12.4.2.3 Optimierung 67

12.5 Round Robin Tournament Verteilung mit Locked Resource 68

12.5.1 Implementierung 68

12.5.2 Messergebnisse 69

12.5.2.1 Erwartung 69

12.5.2.2 Messung 69

12.5.2.3 Diskussion 69

12.5.2.4 Optimierung 70

12.6 Synchronisierte Round Robin Tournament Verteilung 71

12.6.1 Grundstruktur 71

12.6.2 Ansteuerung der Cores 72

12.6.3 Stapeleigenschaften 72

12.6.4 Stapelbildung 73

12.6.4.1 Struktur 73

12.6.4.2 Code 74

12.6.5 Verteilung der Stapel mit RRTA 75

12.6.6 Implementierung 76

12.6.7 Messergebnisse 77

12.6.7.1 Messung 77

12.6.7.2 Diskussion 77

Index 78

Quellenverzeichnis 79

Anlagen 80

Anlagen, Teil 1 I

Anlagen, Teil 2 III

Anlagen, Teil 3 VI

Selbstständigkeitserklärung 7

Abbildungsverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

Tabellenverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

Abkürzungsverzeichnis

**RRT** - Round Robin Tournament

**RRTA** - Round Robin Tournament Algorithmus

Einleitung

Dieser Abschnitt wird sich damit befassen, Grundlagen zu dem Thema zu erläutern und die Motivation hinter dieser Arbeit zu erklären.   
Anschließend werden zu erwartende Probleme aufgezeigt und sowohl logische als auch strukturelle Herangehensweisen angeboten.   
Die Vorschläge werden nicht im Detail ausgearbeitet, sondern sollen ein besseres Verständnis für das Problem schaffen, auf dem im Laufe der Arbeit aufgebaut wird.  
Strukturelle Konzepte werden oberflächlich gehalten und sollen klären, auf welcher Abstraktionsebene die Lösungen angewendet werden. Es soll ersichtlich werden, wo die Grenzen zum ober- und unterliegenden System liegen.

Motivation

In der Prozessorindustrie zeichnet sich ein starker Trend ab. Anstelle von höheren Frequenzen, setzen Prozessorhersteller auf mehrere Rechenkerne innerhalb ihrer Prozessoren [1]. Dies zwingt auch die Softwareentwicklung zu einem Umdenken.  
Parallelisierung von Standardproblemen sollte nicht mehr die Ausnahme, sondern die Regel sein. Dies ist jedoch oft aufwendig und Parallelisierung von Software kann, bei schlechter Umsetzung, mehr Probleme bereiten, als dadurch gelöst werden.   
Deshalb braucht es generische, skalierbare Lösungen, welche in dieser Arbeit untersucht werden sollen.

Vielen Problemen liegt diese Art der Berechnung zugrunde. Beispielsweise bei Physiksimulationen, können Wechselwirkungen zwischen Elementen ein solches Berechnungsmuster aufweisen. Eine Optimierung bzw. Beschleunigung solcher Vorgänge kann eine Zeitersparnis von Sekunden, Stunden oder sogar Wochen bedeuten.  
Wenn die Simulation in Echtzeit berechnet wird, kann ein Leistungsgewinn durch Parallelisierung, dabei helfen, stabile Frameraten zu generieren.

Zielsetzung

Es sollen mögliche Parallelisierungsmethoden auf ihre Kompatibilität mit dieser Problemstellung untersucht werden.  
Diese sollen, durch empirische Tests, in möglichst verschiedenen und extremen Szenarien geprüft und validiert werden.

Das Ziel ist nicht, dem Leser dieser Arbeit, eine vollständige Implementierung vorzugeben, sondern stattdessen Optionen aufzuzeigen, welche sich unter realen Bedingungen bewährt haben.

Grundlagen

Paarweise, ungeordnete Berechnungen

An dieser Stelle muss festgehalten werden, dass es sich um einen, vom Autor, gewählten Ausdruck handelt.

Die Art der Berechnung ergibt sich aus der abzählenden Kombinatorik, bei der für eine Anzahl n an Elementen einmalig Berechnungen zwischen den Elementen angestellt werden müssen.  
„Paarweise“ sagt hierbei aus, dass die Berechnung immer für zwei Elementen ausgeführt wird.  
„Ungeordnet“ steht für eine beliebige Reihenfolge der Berechnungen.

Beispiel: **Berechnungen zwischen 3 Elementen { A, B, C }**

Mögliche paarweise, ungeordnete Kombinationen und deren Berechnungen:

AB – Berechnung 1 zwischen den Elementen A und B  
AC – Berechnung 2 zwischen den Elementen A und C   
BC – Berechnung 3 zwischen den Elementen B und C  
  
Es ergeben sich somit **drei** paarweise, ungeordnete Berechnungen.

Für die ungeordnete Kombinatorik gilt folgende Formel für die Anzahl der Berechnungen

Für die paarweise Variante ist k immer 2

Begriffe

### Berechnungselemente

Alle Elemente zwischen denen ungeordnete, paarweise Berechnungen durchzuführen sind.

### Globale Daten

Wenn, innerhalb der Berechnungen, Daten geteilt werden, werden diese als Globale Daten angesehen.

### Core

Als Cores werden jene Teile des Rechnersystems beschrieben, auf welche die Berechnungen im Endeffekt verteilt werden. Mehrere Cores bilden zusammen den Prozessor.

### Dummy

Dummies ersetzen Teile des Systems, um spezielle Testroutinen durchführen zu können. Ein Dummy-Objekt kann entweder zusätzliche Funktionen bereitstellen, um zusätzliche Informationen bereitzustellen, oder besonders geringen Berechnungsaufwand benötigen, um Effizienztests für andere Module zu ermöglichen.

### Latenz

Die Verzögerungszeit zwischen den Cores. Dauert die Kommunikation, und damit auch die Synchronisation, zwischen den Cores lange, bedeutet dies eine hohe Latenz.

Entwicklungsumgebung

Entwickelt wird mit der Programmiersprache C# mit dem .NET-Framework von Microsoft. Die Arbeit selbst soll jedoch eine allgemeine Lösung, unabhängig von der Entwicklungsumgebung, ins Auge fassen. Die Implementierung dient lediglich der Beweisbarkeit und zur Veranschaulichung.

Es soll an dieser Stelle jedoch erwähnt sein, dass eine funktionale Sprache viele Vorzüge bietet, um die entwickelten Verteilungssysteme vielseitig anwendbar zu machen.

Testumgebung

Betriebssystem

Getestet wird auf einem Windows 10 Betriebssystem. Dieses unterstützt nativ das verwendete .NET-Framework. Es handelt sich dabei um kein Echtzeitsystem, was die Varianz der Ergebnisse erhöht.

Prozessor

Es wird auf zwei verschiedenen Prozessoren getestet. Ein Intel i3 mit zwei echten Cores und aktiviertem Hyperthreading (4 Threads) und ein Intel i5 mit vier echten Cores ohne Hyperthreading (ebenfalls 4 Threads).

Gegenüberstellung der Algorithmen

Um einen möglichst guten Vergleich anstellen zu können, werden sowohl die Algorithmen, als auch die Systemabstraktionen selbst, möglichst modular implementiert.  
Dies erlaubt eine flexible Kombination der Algorithmus- und Systemmodule wodurch möglichst viele verschiedene Tests durchgeführt werden können.

Zeitmessungen innerhalb der Tests sind immer von Zufall behaftet. Bei Rechenvorgängen, kann zum Beispiel das Betriebssystem die Rechenzeit von Test zu Test unterschiedlich verteilen.  
Solche Zeitmessungen werden deshalb wiederholt durchgeführt werden, um stattdessen einen Mittelwert zu erhalten, welcher statistische Ausreißer leicht erkennbar macht und in diesen Messungen wesentlich aussagekräftiger ist.

Probleme bei der Verteilung von Berechnungen

## Ressourcenzugriff

Wie bei den meisten parallelen Systemen, ist der virtuell gleichzeitige Zugriff auf einzelne Ressourcen problematisch. Es muss davon ausgegangen werden, dass Ressourcen vor gleichzeitigem Zugriff geschützt werden müssen.

## Synchronisation

Bei jeder Verteilung muss das Ergebnis der Cores anschließend synchronisiert werden. Dies bedeutet, dass entweder Zwischenergebnisse zwischen den Cores ausgetauscht werden, oder das Endergebnis der Cores zurück in die Applikation fließen. Je nach Algorithmus kann dies mit nur einer oder aber mit sehr vielen Synchronisierungen passieren.   
Synchronisationen können, je nach System, unterschiedlich lang dauern und wirken sich dementsprechend stark aus, wenn viele Synchronisationen in einem System passieren, welches diese nur sehr langsam durchführen kann.

## Overhead

Auch der sogenannte „Overhead“ der Verteilung muss beachtet werden. Dauert die Verteilung der Berechnungen lange, bedeutet dies einen großen Overhead und muss durch eine dementsprechend verkürzte Rechenzeit gerechtfertigt sein.

## Mehrfachberechnung

Je nach Verteilungsalgorithmus kann es sein, dass manche Berechnungen mehrfach, auf unterschiedlichen Nodes, durchgeführt werden.  
Für diese Arbeit zählt ausschließlich die Geschwindigkeit als Kriterium, es soll an dieser Stelle jedoch erwähnt werden, dass mit steigender Anzahl der Berechnungen auch der Energieverbrauch steigt, was für Systeme mit beschränktem Energiehaushalt durchaus ein Grund sein kann, einen Verteilungsalgorithmus nicht anzuwenden.

Lösungsansätze

Validierung der Algorithmen

Um sämtliche Berechnungen zu erhalten, kann folgender Pseudocode verwendet werden:

n… Anzahl der Elemente  
elements... Array mit n Elementen

for (int i = 0; i < n - 1; i++)

{

for (int j = i + 1; j < n; j++)

{

firstElement = elements[i];

secondElement = elements[j];

Calculation(firstElement, secondElement);

}

}

Dies wird als Referenzwert verwendet, um sämtliche Algorithmen zu validieren. Sollte ein Algorithmus nicht die gleichen Berechnungen ausführen, muss er als fehlerhaftet betrachtet werden.

Locked Resource

Ein Standardverfahren, um Ressourcen vor gleichzeitigem Zugriff durch mehrere Threads zu schützen.

### Vorgehensweise

Jede Ressource wird einzeln vor Zugriffen geschützt. Wenn ein Core eine Berechnung durchführt, werden zuerst beide Elemente, welche für die Berechnung benötigt werden, für alle anderen Cores gesperrt. Sobald die Berechnung beendet ist, gibt der Core die Elemente wieder frei.



Core 1 sperrt Element 1 und Element 2 und führt die Berechnung zwischen ihnen durch.  
  
Core 2 will die Berechnung zwischen Element 2 und Element 4 ausführen, muss jedoch warten bis Core 1 das Element 2 wieder freigibt.

### Deadlock-Gefahr

Sobald mit zwei oder mehr Semaphoren gearbeitet wird, muss analysiert werden, ob das System in einem Deadlock-Zustand enden kann. Dies kann jedoch bei richtiger Implementierung niemals passieren, denn es werden unter keinen Umständen die gleichen Semaphore von zwei verschiedenen Threads belegt. Das liegt daran, dass dies nur passieren könnte, wenn zwei Threads die gleiche Berechnung zwischen den gleichen Elementen durchführen.  
Im weiteren Sinne bedeutet das, dass ein Deadlock nur auftreten kann, wenn der Algorithmus von vornherein falsch arbeitet.

### Diskussion

Diese Art der Verteilung führt je nach Anzahl der Cores zu vielen oder wenigen Zugriffskollisionen. Bei einer Zugriffskollision versucht ein Core auf eine Ressource zuzugreifen, welche derzeit von einem anderen Core verwendet wird. Der Core muss nun warten bis die Ressource freigegeben wird, was zu erheblichen Leistungseinbußen führt, wenn solche Kollisionen oft vorkommen.

Round Robin Tournament Algorithmus

Diese Art der Verteilung wurde eigentlich für das Zuweisen von Spielpartnern, bei zum Beispiel Schachturnieren, entwickelt. Die Besonderheit ist dabei, dass jeder Teilnehmer gegen jeden anderen Teilnehmer genau einmal spielt.   
Die Verteilung sorgt dafür, dass pro „Spielrunde“ die maximale Anzahl an Kombinationen gleichzeitig durchgeführt werden.   
Durch die paarweise Struktur lässt sich dieses System auch auf die Problemstellung dieser Arbeit anwenden.

### Vorgehensweise

Die Spieler werden durchnummeriert und „kreisförmig“ angeordnet.  
Jeweils der Spieler, der in der Tabelle oben steht, spielt in der ersten Runde gegen den Spieler, welcher unter ihm steht. Im Beispiel wären das Spieler 1 gegen 14, Spieler 2 gegen 13, usw.



Runde 1

In der zweiten Runde werden alle Spieler um eins weitergerückt, mit der Ausnahme von einem Spieler (im Beispiel Spieler 1).  
Nun spielt wieder die obere Reihe gegen die untere (Spieler 1 gegen 13, Spieler 14 gegen 12, …).

 Runde 2

Dieser Vorgang wird wiederholt bis sämtliche Kombinationen von Spielerpaaren miteinander gespielt haben.

 Letzte Runde (Runde 13)

Die Rundenzahl ist abhängig von der Anzahl der Teilnehmer.

Die Anzahl der benötigten Runden beträgt für **N = gerade** **Teilnehmerzahlen**:  
Benötigte Runden = N – 1

Für **N = ungerade** **Teilnehmeranzahl**:  
Benötigte Runden = N

### Diskussion

Es wird eine gleichmäßige Auslastung mit verhältnismäßig geringem Overhead erreicht. Der größte Teil der Verteilung kann im Vorfeld berechnet werden und wirkt sich dadurch nur sehr mild im Betrieb aus.  
Der Algorithmus skaliert mit der Anzahl der verfügbaren Cores, wenn die Anzahl der Elemente, welche zu berechnen sind, im gleichen Maß steigen.  
Ein Nachteil sind die vielen Synchronisationen, die zwischen den Berechnungsrunden durchgeführt werden müssen. Die Anzahl der Synchronisierungen steigt mit der Anzahl der Cores linear an. Dies wird bei Systemen mit hohen Latenzen problematisch.

Deepcopy

Es wird für jeden Core eine eigene Kopie der Daten erstellt. Auf jedem Core wird dann ein Teil der Berechnungen ausgeführt. Die Ergebnisse von allen Cores müssen zum Schluss synchronisiert werden.

### Diskussion

Je nach Menge der benötigten Daten, kann der initiale Overhead groß sein.  
Sind die Daten jedoch verteilt, kann jeder Core, ohne Synchronisation, seinen Teil der Berechnungen störungsfrei durchführen.  
Die Synchronisation und Kombination der Endergebnisse kann je nach Art der Berechnung aufwendig sein und läuft außerdem nicht parallel ab, was den Vorgang nochmals verlangsamt.

Divide and Conquer

Bei „Divide and Conquer“-Algorithmen handelt es sich um spezialisierte Vorgehensweisen, große Aufgaben oder Berechnungen, in kleinere aufzuteilen [2], bis diese vernünftig lösbar sind.

Im Bereich der Parallelisierung ist die Aufteilung vor allem so zu verstehen, dass die Teilaufgaben getrennt voneinander berechnet werden können [3].  
Dies hat zur Folge, dass sich die einzelnen Teilbereiche sehr gut auf mehrere Rechenkerne verteilen lassen.

Die Herausforderung dabei ist es, die schlussendliche Rechenarbeit auf den einzelnen Rechenkernen möglichst gleich zu halten. Wenn die Aufteilung hingegen in zu kleine Bereiche ausfällt, kann der Overhead unerwünscht groß werden.

Nicht jedes Problem lässt sich durch einen „Divide and Conquer“-Algorithmus günstig aufteilen, wodurch es bei einer unpassenden Lösung durchaus sein kann, dass kein Effizienzgewinn oder sogar ein Effizienzverlust die Folge sind.

Parallelisierungs-Systeme

In diesem Abschnitt werden verschiedene Methoden, zur Parallelisierung von Programmen unter Windows, vorgestellt. Eine Verteilung könnte auf jedem der Systeme arbeiten, jedoch werden die Tests auf einem gemeinsamen System durchgeführt, um die Messung ausschließlich von der Verteilung, und nicht von unterschiedlichen Parallelisierung Systemen abhängig zu machen.

Threadspawning

Bei jeder Arbeitsanweisung werden neue Threads erzeugt, welche dann die auf die Cores verteilt werden.  
Ist die Berechnung abgeschlossen, werden die Threads zerstört und die Ressourcen wieder freigegeben.  
Das Erzeugen von Threads ist nicht billig und es muss daher mit Leistungseinbußen gerechnet werden. Es könnte sich jedoch in einem System, mit sehr begrenztem Speicher, als die einzige Alternative herausstellen.

Actors

Die Threads (Actors) werden initial erstellt und laufen immer parallel mit dem Hauptprogramm mit.  
Sobald Berechnungen durchzuführen sind, werden diese als „Messages“ an die Actors gesendet und abgehandelt.  
Da die Threads permanent sind, sind die benötigten Ressourcen dauerhaft belegt. Dies ist jedoch nicht mit Prozessorlaufzeit gleichzusetzten, da die Actors im Sleep Modus sind, wenn sie nicht aktiv sind.

.NET Threadpool

Die .NET Implementierung des Threadpools vereint die Eigenschaften von Threadspawning und Actors [4].  
Threads werden nach Bedarf erzeugt und bleiben weiterhin bestehen, wenn sie nicht mehr benötigt werden. Beim erhalten einer „Message“ wird zuerst ein inaktiver Thread angesteuert, bevor neue generiert werden.   
Dies hält den Ressourcenverbrauch niedrig, ohne dabei große Leistungseinbußen hinzunehmen.  
Nachteil ist die statische Implementierung, welche Verteilungsspezifische Optimierungen verhindert.  
Die interne Implementierung dieses Systems übersteigt das Volumen dieser Arbeit, kann jedoch auf der offiziellen Website von Microsoft nachgeschlagen werden [5].

Testsetup

Überblick

### Initialisierung

Die Testsuite wählt zuerst eine Kombination aus Verteilungssystem und Berechnungssytem, welche getestet werden soll. Dabei kann das Berechnungssystem auch Teil der Testsuite sein, um Tests durchführen zu können, welche von außerhalb nicht möglich wären.  
Die Konfiguration des Berechnungssystem beinhaltet zum Beispiel die Anzahl der verwendeten Cores, Verzögerungszeiten oder die Art der Berechnung, welche auf den Input angewendet wird.



### Ablauf

Der Input ist abhängig von der Testroutine und kann entweder aus realen Daten bestehen, um Effizeinz zu testen, oder aus Dummyobjekten bestehen, welche zusätzliche Auswertungen ermöglichen.  
Über den Output wird validiert, ob die Berechnungen korrekt durchgeführt wurde.  
Der Status des Berechnungssystems liefert Detailinformationen, zum Beispiel welche Berechnung auf welchem Core ausgeführt wurde.

Grundstruktur

Alle Tests erben von dieser Grundstruktur.



Jeder Test wird durch Constructor-Injection (<http://philkildea.co.uk/james/books/Dependency.Injection.in.NET.pdf>) mit einem Verteilungsalgorithmus initialisiert.  
Die Daten „m\_elements“ und „m\_globalData“ müssen in der Vererbung gesetzt werden.  
Die Funktion „CalculationFunction“ ist abstrakt und muss ebenfalls in der erbenden Klasse implementiert werden. Die Funktion wird während des Testens auf alle Elementpaare angewandt.

Die Funktion „TestRoutine“ führt den eigentlichen Test durch und schreibt die Ergebnisse in den Parameter. Dies beinhaltet auch die benötigte Zeit für den Durchlauf.

Output-Validierung

Der Output wird mithilfe von Dummy-Objekten validiert. Dabei führt jedes Element eine Liste von allen Elementen, mit denen es bereits berechnet wurde.  
Die Berechnung auf den Cores, beinhaltet lediglich das Beschreiben dieser Liste.  
Nach dem Abschluss der Berechnungen, werden sämtliche Listen auf Fehler geprüft.

### Validierungs-Input

Beschreibt die Dummyobjekte, welche zur Validierung des verwendeten Verteilungsalgorithmus, benötigt werden.

#### Struktur



Zuerst eine Übersicht über die intern geführte Liste. Diese wird mit der Funktion „Valid“ geprüft und liefert „true“ zurück, wenn sich die Liste im validen Zustand befindet.

Initialer Zustand von m\_calculatedWith bei insgesamt vier Elementen:

|  |  |
| --- | --- |
| Index | Value |
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |

Valid() liefert „false“ zurück

Valider Zustand von m\_calculatedWith wenn ElementIndex gleich 2 ist. Die Berechnung wurde mit allen Elementen genau einmal durchgeführt und nicht mit sich selbst:

|  |  |
| --- | --- |
| Index | Value |
| 0 | 1 |
| 1 | 1 |
| 2 | 0 |
| 3 | 1 |

Valid() liefert „true“ zurück

Invalider Zustand von m\_calculatedWith. Die Berechnung wurde doppelt mit Element 0 ausgeführt:

|  |  |
| --- | --- |
| Index | Value |
| 0 | 2 |
| 1 | 1 |
| 2 | 0 |
| 3 | 1 |

Valid() liefert „false“ zurück

Über die Funktion „SetCalculatedWithElement“ wird in die Liste geschrieben, dass die Berechnung mit dem übergebenen Index durchgeführt wurde.

|  |  |
| --- | --- |
| Index | Value |
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |

|  |  |
| --- | --- |
| Index | Value |
| 0 | 0 |
| 1 | 0 |
| 2 | 1 |
| 3 | 0 |

SetCalculatedWithElement( 2 ); 🡪

#### Code

class ValidationDummy

{

int[] m\_calculatedWith;

public int ElementCount { get { return m\_calculatedWith.Length; } }

public int ElementIndex { get; private set; }

public ValidationDummy(int elementCount, int index)

{

ElementIndex = index;

m\_calculatedWith = new int[elementCount];

for (int i = 0; i < ElementCount; i++)

{

m\_calculatedWith[i] = 0;

}

}

public void SetCalculatedWithElement(int otherElementIndex)

{

m\_calculatedWith[otherElementIndex]++;

}

public bool Valid()

{

for (int i = 0; i < ElementCount; i++)

{

if (m\_calculatedWith[i] > 1)

return false;

if ((i == ElementIndex) == (m\_calculatedWith[i] == 1))

return false;

}

return true;

}

}

### Test-Ablauf

Behandelt die Umsetzung des Validierungstests. Grundstruktur zu finden auf Seite 18.

#### Struktur



Die Typen werden auf „ValidationDummy“ als Element-Typ und „int“ als globaler Daten-Typ festgelegt. Die globalen Daten sind in diesem Test jedoch irrelevant und wurden willkürlich gewählt.

Die Klasse erstellt selbstständig die Dummyobjekte und führt einen Berechnungsdurchlauf mit ihnen durch. Anschließend wird geprüft, ob tatsächlich sämtliche Berechnungen durchgeführt wurden.

Dazu wird die Berechnungsfunktion „CalculationFunction“ überschrieben, um die Listen in den Validierungs-Dummies zu beschreiben.

Die Validierungsfunktion „Valid“ prüft sämtliche Dummyobjekte und deren interne Listen.

#### Code

class OutputValidation : UniquePairTest<ValidationDummy, int>

{

public OutputValidation(

UniquePairDistribution<ValidationDummy, int> distribution,

int elementCount) :

base(distribution)

{

// initialize the validation objects

m\_elements = new ValidationDummy[elementCount];

for (int i = 0; i < elementCount; i++)

m\_elements[i] = new ValidationDummy(elementCount, i);

}

protected override void CalculationFunction(

ValidationDummy part1, ValidationDummy part2, int global)

{

part1.SetCalculatedWithElement(part2.ElementIndex);

part2.SetCalculatedWithElement(part1.ElementIndex);

}

protected override void TestRoutine(TestResult result)

{

// distribute calculations and execute them with the injected algorithm

Distribution.Calculate(m\_elements, m\_globalData);

result.Successful = Valid();

}

private bool Valid()

{

// check every validation object and return true if all are valid

for (int i = 0; i < m\_elements.Length; i++)

{

if (!m\_elements[i].Valid())

return false;

}

return true;

}

}

Effizienz-Messung

Die Effizienz-Messung wird ohne Output Validierung durchgeführt, da die Prüfung zusätzlichen Rechenaufwand verursacht, welcher die Zeitmessung verfälscht.  
Wie auf Seite 10 beschrieben, werden die Messungen mehrfach ausgeführt, um Mittelwert und statistische Ausreißer nach oben und unten hin, zu erfassen.  
Es werden verschiedene Testroutinen durchgeführt um möglichst viele Eigenschaften einer Verteilung messen zu können. Diese werden im folgenden Abschnitt beschrieben.

### Struktur

Die Struktur ist bei allen Effizienz-Messungen gleich. Die übergeordnete Struktur ist auf Seite 18 zu finden.



Unterschiede zur Basisklasse sind der vordefinierte Typ, welcher auf Integer festgelegt wurde. Dies dient entweder zur Identifikation des Elements oder ist schlichtweg irrelevant für manche Tests.

Im Konstruktor werden die einzelnen Elemente auf Integer-Werte initialisiert, welche den Index des Elementes darstellt.

### Code

abstract class TimeMeasurementTest : UniquePairTest<int, int>

{

public TimeMeasurementTest(int elementCount) : base()

{

// initialize elements

m\_elements = new int[elementCount];

for (int i = 0; i < elementCount; i++)

{

// set value to element index

m\_elements[i] = i;

}

}

protected abstract void CalculationFunction(int element1, int element2, int global);

}

### Overhead-Messung

Um den Overhead effektiv feststellen zu können, wird der Berechnungsanteil auf ein Minimum gesetzt. Die Zeitmessung liefert somit den Anteil zurück, welcher die Verteilung damit verbringt, die einzelnen Rechnungen aufzuteilen.

Der Berechnungsfunktion „CalculationFunction“, welche bei jedem Element-Paar durchgeführt wird, wird auf eine NOP (No operation) Funktion gesetzt, um dem System möglichst wenig Rechenzeit abzunehmen.

Außerdem sollte eine große Anzahl an Elementen (und damit Berechnungen) gewählt werden, um den Overhead-Anteil im Vergleich zu den restlichen Abläufen möglichst groß zu halten.

protected override void CalculationFunction(int part1, int part2, int global)

{

// don’t do anything to keep the load at a minimum

}

### Zeitmessung mit fixierter Rechenzeit

Input und Output spielen bei diesem Test keine Rolle. Es geht lediglich darum, durch die fixierte Rechenzeit, einen Effizienzwert für die Verteilung zu erhalten.

Bei dieser Messung, wird die Berechnungsfunktion überschrieben, damit sie eine fixe Laufzeit hat. Es ist dabei irrelevant ob sie aktiv vom Prozessor bearbeitet werden muss.

protected override void CalculationFunction(int element1, int element2, int global)

{

// fixed duration of the execution

Thread.Sleep(100);

}

Zeitmessung mit zufälliger Rechenzeit

Der Aufbau ist gleich wie bei der Zeitmessung mit fixierter Rechenzeit auf Seite 25**Error! Bookmark not defined.**. Der Unterschied ist lediglich, dass statt einer fixen Rechenzeit, eine zufällige benötigt wird, welche bei jedem Paar unterschiedlich sein kann.

Diese Messung ist wichtig, da sie längere Laufzeiten bei der Synchronisation aufzeigt, wenn die Cores unterschiedlich lange für die Berechnung benötigen.

protected override void CalculationFunction(int element1, int element2, int global)

{

// random duration of the execution

Thread.Sleep(Random(100));

}

### Zeitmessung mit Auslastung

Es wird keine Laufzeit vorgegeben, sondern eine aktive Rechnung zwischen den Elementen durchgeführt. Dies verursacht neben dem Overhead eine Belastung der Cores. Diese Messung stellt am besten eine reale Anwendung der Verteilung dar.

protected override void CalculationFunction(int element1, int element2, int global)

{

int sum = 0;

for (int i = 0; i < Difficulty; i++)

{

sum += element1 \* element2;

}  
}

Die Variable „Difficulty“ stellt die Anzahl der Schleifendurchläufe und damit die Laufzeit der Berechnung ein.

Systemabstraktion

Das Verteilungssystem soll ein austauschbarer Teil des Gesamtsystems sein und zwischen der eigentlichen Applikation und dem unterliegenden Rechensystem arbeiten.  
Dies soll die Applikation unabhängig davon machen, welches unterliegende System schlussendlich die Berechnungen, parallelisiert, durchführt.

|  |  |  |  |
| --- | --- | --- | --- |
| Applikation | | | |
|  |  |  |  |
| Verteilung | | | |
|  |  |  |  |
| Berechnungssystem | | | |
|  |  |  |  |
| Cores | | | |

Das Berechnungssystem soll über ein Interface an das Verteilungssystem gekoppelt werden.   
Dadurch kann das Berechnungssystem flexibel ausgetauscht werden und außerdem durch ein Dummysystem ersetzt werden, welches gewisse Eigenschaften des Berechnungssystems simulieren kann. Das Dummysystem kann außerdem dazu verwendet werden, um Tests durchzuführen, die von außerhalb nicht möglich sind.



Core-Pool

Diese Klasse dient als Basis für alle unterliegenden Berechnungssysteme. Sie stellt die notwendige Information über das System bereit, um die Verteilungen anzupassen, und bietet sowohl Berechnungs- als auch Synchronisationsfunktionalität, welche jedoch in den erbenden Klassen implementiert werden müssen.



### Berechnungsanweisung

Innerhalb des Core-Pools werden Anweisungen, für das Berechnen von Paaren, mittels „PairingData“-Objekten an die Funktion „DistributeCalculation“ übergeben.

public class PairingData<PartType, GlobalDataType>

{

public PartType[] Stack1 { get; set; }

public PartType[] Stack2 { get; set; }

public GlobalDataType GlobalData { get; set; }

public bool CalculateInternally { get; set; }

}

Mithilfe der „Stacks“ wird festgelegt, welche Elemente mit dem Aufruf kombiniert werden sollen.  
„CalculateInternally“ legt fest, ob die Berechnungen auch innerhalb der Stacks durchgeführt werden müssen.

Beispiel für einen Aufruf:

|  |  |
| --- | --- |
| Stack1 | Stack2 |
| A | C |
| B | D |

Für CalculateInternally = false müssen folgende Paare berechnet werden:  
AC, AD, BC, BD

Für CalculateInternally = true müssen folgende Paare berechnet werden:  
 AB, AC, AD, BC, BD, CD

### Synchronisation

Ein Aufruf der Synchronisations-Funktionen muss gewährleisten, dass der aufrufende Thread so lange wartet, bis die Berechnungen abgeschlossen sind.

Wenn ein Core-Index an die Funktion „Synchronize“ übergeben wird, muss der Thread warten, bis der jeweilige Core seine Berechnungen beendet hat.  
Ohne Argument muss gewartet werden, bis sämtliche Berechnungen auf allen Cores abgeschlossen wurde.

Verteilungsstrukturen

Dieser Abschnitt beschreibt die grundlegenden Verteilungsklassen.

## Grundstruktur

Die abstrakte Basisklasse, von welcher sämtliche Verteilungsalgorithmen erben.

### Struktur



“GetCoreCount” liefert die Anzahl der verwendeten Cores zurück. Diese muss nicht zwingend mit der tatsächlichen Anzahl des Systems übereinstimmen und sagt lediglich aus, wie viele die Verteilung selbst verwendet.

„SetCalculationFunction“ legt die Berechnungsfunktion fest, welche zwischen den Elementen angewandt wird.

„Calculate“ verteilt alle Berechnungen, abhängig von dem implementierten Algorithmus, auf die Cores. Alle Berechnungen müssen nach dem Aufruf abgeschlossen sein.

### Code

public abstract class UniquePairDistribution<PartType, GlobalDataType> : IDisposable

{

public delegate void PairCalculationFunction(

PartType element1, PartType element2, GlobalDataType globalData);

public abstract int CoreCount { get; }

public abstract void SetCalculationFunction(PairCalculationFunction function);

public abstract void Calculate(PartType[] elements, GlobalDataType globalData);

public virtual void Dispose()

{

}

}

In der tatsächlichen Implementierung wird zusätzlich das „Dispose“-Pattern [REFERENZ] implementiert, damit die Ressourcen durch den Aufruf wieder freigegeben werden können.

Core-Pool-Verteilung

Wenn eine Verteilung den flexiblen Austausch des unterliegenden Systems erlaubt, so erbt sie von dieser abstrakten Basisklasse. Grundstruktur zu finden auf Seite 29.

### Struktur



Die Verteilung wird mit einer Instanz der „CorePool“-Klasse initialisiert.  
Die Funktionen „SetCalculationFunction“ und „CoreCount“ werden implementiert.

Lediglich die Funktion „Calculate“ bleibt abstrakt und muss von der erbenden Klasse implementiert werden.

### Code

public abstract class CorePoolDistribution<PartType, GlobalDataType> : UniquePairDistribution<PartType, GlobalDataType>

{

protected CorePool<PartType, GlobalDataType> CorePool { get; private set; }

public sealed override int CoreCount { get { return CorePool.CoreCount; } }

public CorePoolDistribution(SharedMemoryCores<PartType, GlobalDataType> pool)

{

CorePool = pool;

}

public sealed override void SetCalculationFunction(

PairCalculationFunction function)

{

CorePool.SetCalculationFunction(function);

}

public override void Dispose()

{

CorePool.Dispose();

}

}

Eigenständige Verteilung

Diese abstrakte Basisklasse wird vor allem für Testszenarien verwendet. Es wird auf die Schnittstelle des Core-Pools verzichtet um die Implementierung insgesamt effizienter zu machen.  
Wird vor allem für Tests verwendet um die Messungen möglichst am optimalen Effizienzgrad durchzuführen. Grundstruktur zu finden auf Seite 29.

### Struktur



Wieder ist die einzige Funktion, die noch überschrieben werden muss, die „Calculate“-Funktion.  
„CoreCount“ wird mittels Konstruktor gesetzt und „SetCalculationFunction“ erhält ein Feld in der Klasse, in dem die übergebene Funktion gespeichert wird.

### Code

public abstract class StandaloneDistribution<PartType, GlobalDataType> :

UniquePairDistribution<PartType, GlobalDataType>

{

protected PairCalculationFunction CalculationFunction { get; private set; }

private int m\_coreCount;

public sealed override int CoreCount { get { return m\_coreCount; } }

public StandaloneDistribution(int coreCount)

{

m\_coreCount = coreCount;

}

public sealed override void SetCalculationFunction(

PairCalculationFunction function)

{

CalculationFunction = function;

}

}

Lock-Verteilung

Um Algorithmen mit gesperrten Ressourcen (Seite 13) zu realisieren, wird eine Basisklasse erstellt, welche die grundliegende Ressourcenverwaltung übernimmt. Die erbenden Klassen sollen nur noch die Verteilung selbst implementieren und keine Information über die Semaphore, welche zum Sperren der Ressourcen verwendet werden, haben.

Aus Effizienzgründen wird auf die Abstraktion durch die Core-Pool Klasse verzichtet.

Die Implementierung wird nur oberflächlich beschrieben und auf das Nötigste reduziert.

### Struktur



### Code

Die „Calculate“-Funktion initialisiert die Semaphore und Signalobjekte. Danach werden die Threads erzeugt, welche die „Distribute“-Funktion ausführen.  
Anschließend wird gewartet, bis sämtliche Threads die Berechnungen beendet haben bzw. die Signalobjekte „m\_waitHandles“ geschalten wurden.

public override void Calculate(PartType[] elements, GlobalDataType globalData)

{

ResetWaitHandles();

ResetLocks(elements.Length);

for (int i = 0; i < CoreCount; i++)

{

ThreadPool.QueueUserWorkItem(Distribute, i);

}

WaitHandle.WaitAll(m\_waitHandles);

}

protected abstract void Distribute(int coreID);

“CalculatePair” ist eine Hilfsfunktion für die erbenden Klassen, welche es erlaubt, die Berechnung für ein Paar an Elementen durchzuführen, ohne, dass ein anderer Thread währenddessen auf die Elemente zugreifen kann.

protected void CalculatePair(int id1, int id2)

{

m\_locks[id1].WaitOne();

m\_locks[id2].WaitOne();

CalculationFunction(m\_elements[id1], m\_elements[id2], m\_global);

m\_locks[id2].Release();

m\_locks[id1].Release();

}

Round Robin Tournament Matrix

Die grundsätzliche Funktionsweise wird auf Seite 14 beschrieben.

Um eine möglichst gleichmäßig und kollisionsfreie Anordnung der Berechnungen durch den Round Robin Tournament Algorithmus zu erhalten, wird eine Matrix erzeugt, welche für jeden Aufruf das jeweilige Elementpaar zurückliefert, welches berechnet werden soll..



### Basis-Array

Das Basis-Array bildet den RRTA im ersten Schritt (step 0) ab. Im Speicher wird es als Array von Integer-Werten dargestellt. Wird mit der Funktion „CreateBaseArray“ erzeugt.  
  
Als Beispiel der erste Schritt des RRTA mit 6 Elementen und das zugehörige Interger-Array.  
Element-Index-Paare sind farblich gekennzeichnet:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RRTA - erster Schritt | | |  | Speicheräquivalent als Integer-Array | | | | | |
| 0 | 1 | 2 |  | 0 | 1 | 2 | 3 | 4 | 5 |
| 5 | 4 | 3 |  |  |  |  |  |  |  |

### Array-Verschiebung

Schiebt den Zustand des Arrays in den nächsten RRTA-Schritt.   
Befindet sich das Array im Basiszustand (erster Schritt oder Step 0) dann wird es in den Zustand des zweiten Schrittes (Step 1) gebracht.

Mithilfe der Funktion „ShiftArray“ wird aus dem ersten Schritt der zweite generiert:

  
  
 🡪 ShiftArray() 🡪

### Matrix-Generierung

Die „GenerateMatrix“-Funktion generiert alle Daten, inklusive der RRTA Matrix, aus der Anzahl der Elemente.

Für eine Elementanzahl von 4 würden die RRTA-Schritte wie folgt aussehen.  
Step 0 ist das Basis-Array von 4 Elementen:



ShiftArray()🡪 ShiftArray()🡪

Aus diesen Schritten wiederum, wird eine Matrix generiert, welche für den jeweiligen Schritt und den jeweiligen Paarindex, beide Elementindexe zurückliefert.

In dem vorhergehenden Beispiel, müsste der Zugriff auf die Matrix mit Schritt 2 und Paarindex 0 die Elementnummer 0 und 2 zurückgeben.

* PairMatrix[step2][pair0] = { 0 , 2 }

Für die Verteilung würde dies nun bedeuten, dass im **3. Durchlauf** der **Core 0** die Berechnung zwischen **Element 0** und **Element 2** durchführen muss.

Die vollständige Matrix für 4 Elemente:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Paar | |
|  |  | 0 | 1 |
| Schritt | 0 | { 0 , 3 } | { 1 , 2 } |
| 1 | { 0 , 1 } | { 2 , 3 } |
| 2 | { 0 , 2 } | { 1 , 3 } |

In dieser Matrix sind immer sämtliche Paare, welche zwischen einer bestimmten Anzahl an Elementen möglich sind, abgedeckt.

Die Parameter „StepCount“ und „PairCount“ stellen die Höhe bzw. Breite der Matrix dar. In diesem Beispiel wäre der StepCount 3 und der PairCount 2.

Vom Programm generierte Matrix mit 8 Elementen auf die Konsole ausgegeben:



Algorithmus-Optimierung

Im folgenden Abschnitt werden Algorithmen beschrieben, implementiert, getestet und optimiert.  
Jeder, der Algorithmen, wurde durch den Validierungsprozess getestet und liefert eine gültige Verteilung.  
Außerdem werden Effizienztests durchgeführt, um zu bestimmen, ob eine Verteilung besser oder schlechter, im Vergleich zu anderen, abschneidet.

Die Parallelisierung wird einheitlich mit dem Threadpool von dem .NET-Framework (Seite 16) realisiert.

Testroutinen

Es werden die Bedingungen, von allen durchgeführten Tests festgelegt. Sämtliche Verteilungen werden mit diesen Parametern getestet und können somit untereinander verglichen werden.

### Validierung

Jede Verteilung muss mehrere Validierungs-Setups bestehen. Liefert eines der Setups ein invalides Ergebnis, so wird die Verteilung als fehlerhaft angesehen.

Validierung mit folgender Elementanzahl durchgeführt:

4 - Mindestanzahl um zumindest zwei Berechnungen parallel durchführen zu können  
16 - Gut auf die derzeit gängige Anzahl von Prozessorkernen (2, 4 und 8) aufteilbar  
1024 - Große Anzahl an Elementen  
59 - Primzahl im mittleren Bereich

Testdurchläufe:





### Effizienz

Für die Feststellung der Effizienz werden unterschiedliche Tests in Bezug auf Laufzeit oder Auslastung vorgenommen. Die Ergebnisse dürfen nur innerhalb des gleichen Tests verglichen werden.

Die Algorithmen wurden für die Tests optimiert, um die Messung unter optimalen Umständen durchzuführen.   
Dies beinhaltet teilweise auch das Entfernen der Core-Pool-Schnittstelle, was die Kompatibilität mit den unterliegenden Systemen zerstört. Der Core-Pool wird durch eine statische Implementierung des .NET-Threadpools ersetzt, was wiederum effizientere Implementierungen erlaubt.

Alle Tests werden 100-mal wiederholt, um aussagekräftige Mittelwerte zu erhalten und um statistische Ausreißer festzuhalten.

#### Overhead 1

Testbeschreibung zu finden auf Seite 25.

Cores: 4  
Elemente: 16  
Daraus resultierende Anzahl der Berechnungen: 120 (Ermittlung auf Seite 7)  
Laufzeit einer Berechnung: Je nach Compiler-Optimierung keine oder nur sehr geringe Laufzeit.

#### Overhead 2

Testbeschreibung zu finden auf Seite 25.

Cores: 4  
Elemente: 128  
Daraus resultierende Anzahl der Berechnungen: 8128 (Ermittlung auf Seite 7)  
Laufzeit einer Berechnung: Je nach Compiler-Optimierung keine oder nur sehr geringe Laufzeit.

#### Overhead 3

Testbeschreibung zu finden auf Seite 25.

Wiederholungen: 10  
Cores: 4  
Elemente: 1024  
Daraus resultierende Anzahl der Berechnungen: 499.500 (Ermittlung auf Seite 7)  
Laufzeit einer Berechnung: Je nach Compiler-Optimierung keine oder nur sehr geringe Laufzeit.

#### Fixierte Rechenzeit 1

Testbeschreibung zu finden auf Seite 25.

Wiederholungen: 10  
Cores: 4  
Elemente: 16  
Daraus resultierende Anzahl der Berechnungen: 120 (Ermittlung auf Seite 7)  
Laufzeit einer Berechnung: 2ms

#### Fixierte Rechenzeit 2

Testbeschreibung zu finden auf Seite 25.

Wiederholungen: 10  
Cores: 4  
Elemente: 128  
Daraus resultierende Anzahl der Berechnungen: 8128 (Ermittlung auf Seite 7)  
Laufzeit einer Berechnung: 2ms

#### Zufällige Rechenzeit 1

Testbeschreibung zu finden auf Seite 26.

Wiederholungen: 10  
Cores: 4  
Elemente: 16  
Daraus resultierende Anzahl der Berechnungen: 120 (Ermittlung auf Seite 7)  
Durchschnittliche Laufzeit einer Berechnung: 10ms  
Varianz: +-5ms

#### Zufällige Rechenzeit 2

Testbeschreibung zu finden auf Seite 26.

Wiederholungen: 10  
Cores: 4  
Elemente: 128  
Daraus resultierende Anzahl der Berechnungen: 8128 (Ermittlung auf Seite 7)  
Durchschnittliche Laufzeit einer Berechnung: 2ms  
Varianz: +-1ms

#### Auslastung 1

Testbeschreibung zu finden auf Seite 26.

Wiederholungen: 10  
Cores: 4

Elemente: 16  
Daraus resultierende Anzahl der Berechnungen: 120 (Ermittlung auf Seite 7)

Berechnete Summierungs-Schleifen: 1024  
Daraus resultierende Anzahl der Summenberechnungen: 122.880

#### Auslastung 2

Testbeschreibung zu finden auf Seite 26.

Wiederholungen: 10  
Cores: 4

Elemente: 128  
Daraus resultierende Anzahl der Berechnungen: 8128 (Ermittlung auf Seite 7)

Berechnete Summierungs-Schleifen: 1024  
Daraus resultierende Anzahl der Summenberechnungen: 8.323.072

#### Auslastung 3

Testbeschreibung zu finden auf Seite 26.

Wiederholungen: 10  
Cores: 4

Elemente: 1024  
Daraus resultierende Anzahl der Berechnungen: 523.776 (Ermittlung auf Seite 7)

Berechnete Summierungs-Schleifen: 1024  
Daraus resultierende Anzahl der Summenberechnungen: 536.346.624

Single-Thread Referenz Algorithmus

Dieser Algorithmus wurde auf Seite 12 beschrieben.

### Implementierung



In der auf Seite 31 beschriebenen Basisklasse wird lediglich die „Calculate“-Funktion überschrieben.  
Die Berechnungen werden sequentiell abgearbeitet und benötigen dafür nur einen Core.  
Der Algorithmus ist demnach nicht parallel und wird nur implementiert, um einen Referenzwert zu erhalten.   
Basierend auf diesem Referenzalgorithmus werden die anderen Algorithmus-Implementierungen bewertet.

public override void Calculate(PartType[] elements, GlobalDataType globalData)

{

for (int i = 0; i < elements.Length - 1; i++)

{

for (int j = i + 1; j < elements.Length; j++)

{

CalculationFunction(elements[i], elements[j], globalData);

}

}

}

### Messergebnisse

#### Messung



#### Diskussion

Der Overhead ist verhältnismäßig gering, was durch den unterbrechnungsfreien Durchlauf mit minimalem Verteilungsaufwand erklärt werden kann.

Bei fixierter und zufälliger Rechenzeit werden ähnliche Ergebnisse erzielt, da sich, über mehrere Wiederholungen hinweg, die zufällige Zeitdauer statistisch an die fixierte Zeitdauer der Berechnungen angleicht.

Beim Auslastungstest sind geringe Belastungen (mit kleiner Elementzahl) schneller, als mit anderen Verteilungen, was vor allem am geringen Overhead und der nicht blockierenden Berechnungen liegt.  
Bei großer Last und gleichzeitig verhältnismäßig kleinem Overhead-Anteil ist allerdings ein deutlicher abwärtstrend erkennbar sein, was die Effizienz betrifft, da nur einer der vier Cores ausgelastet wird.

#### Optimierung

Die Verteilung dient nur als Referenzwert. Die folgenden Verteilungen sollten alle verfügbaren Cores nutzen und somit, durch eine geringe Erhöhung des Overheads, eine stark verbesserte Effizienz, in den anderen Messungen, zeigen.

Parallelisierung durch Locked Resource

Um mehrere Cores verwenden zu können, werden Semaphore verwendet, um einzelne Elemente vor Mehrfachzugriff zu schützen. Dies wurde auf Seite 13 beschrieben.

### Implementierung

Für einen ersten Versuch wird dabei die innere Schleife auf die einzelnen Cores aufgeteilt.

// outer loop

for (int i = 0; i < elementCount - 1; i++)

{

// inner loop, distributing to multiple cores

for (int j = i + 1; j < elementCount; j++)  
 {

Calculation(elements[i], elements[j]);

}

}

Der erste Core berechnet alle Kombinationen mit dem ersten Element, der zweite Core übernimmt alle Kombinationen mit dem zweiten Element, usw.

Um herauszufinden, auf welchem Core die innere Schleife ausgeführt werden soll, wird

CoreIndex = i % Anzahl der Cores

berechnet.

Pseudocode – roter Teil wird auf die Cores verteilt:

// outer loop

for (int i = 0; I < elementCount - 1; i++)

{

int coreIndex = i % CoreCount;

ExecuteInCore(coreIndex)

{

// inner loop

for (int j = i + 1; j < elementCount; j++)

{

Calculation(elements[i], elements[j]);

}

}

}

### Messergebnisse

#### Messung





#### Diskussion

Die Overhead-Messung zeigt einen extremen Zeitbedarf im Vergleich zur Referenzverteilung. Dies liegt an den Zugriffskollisionen, welche statistisch mit der Anzahl der Elemente steigen.

Dieser extreme Anstieg des Overheads wird jedoch bei den Messungen von fixierten und zufälligen Rechenzeiten durch die Nutzung von mehreren Cores kompensiert. Im Vergleich, zur vorherigen Verteilung, werden die Berechnungen mehr als doppelt so schnell durchgeführt. Die Varianz der Messungen ist dabei jedoch hoch, da, je nach Scheduling des Betriebssystems, einmal bessere und einmal schlechtere Bedingungen für Zugriffskollisionen herrschen.

Beim Auslastungstest macht sich der extreme Overhead-Zuwachs bemerkbar. Dieser kann trotz Nutzung von mehreren Cores nicht ausgeglichen werden.

#### Optimierung

Die Verteilung belastet die Cores unterschiedlich stark. Dies liegt daran, dass die innere Schleife, je nach Index i, eine unterschiedliche Anzahl an Durchläufen hat.  
Ist der Index i gleich 0, wird die Schleife (Elementanzahl – 1) mal durchlaufen.  
Bei Index i gleich der Elementanzahl – 2, wird die Schleife nur einmal durchlaufen.

Roter Teil sorgt für die ungleichmäßige Verteilung auf die einzelnen Cores:

for (int i = 0; i < elementCount - 1; i++)  
{  
 for (int j = i + 1; j < elementCount; j++)

{

Calculation(elements[i], elements[j]);

}

}

Bei 16 Elementen und 120 Berechnungen werden nun die Berechnungen wie folgt verteilt:

Core 0: 36  
Core 1: 32  
Core 2: 28  
Core 3: 24

Optimierungspotential liegt in der gleichmäßigen Aufteilung der Berechnungen auf die Cores und der generellen Reduzierung von Zugriffskollisionen.

Gleichmäßige Verteilung der Berechnungen

### Implementierung

Um die Berechnungen möglichst gleichmäßig zu verteilen, wird bei jedem Schleifendurchlauf ein Zähler inkrementiert.

Abhängig von diesem Zähler, wird die Berechnung auf die Cores verteilt.  
Dies sorgt dafür, dass jeder Core maximal eine Berechnung weniger ausführt, als jeder andere.

int coreSelect = 0;

for (int i = 0; i < elementCount - 1; i++)

{

for (int j = i + 1; j < elementCount; j++)

{

coreSelect++; // increment counter

// calculate index of selected core

int coreIndex = coreSelect % CoreCount;

// execute calulation on selected core

CalculatePair(coreIndex, i, j);

}

}

### Messergebnisse

#### Messung





#### Diskussion

Die Verteilung schneidet, trotz gleichmäßiger Verteilung der Berechnungen, in sämtlichen Kategorien schlechter ab als die „Locked Resource“-Verteilung.

Dies lässt den Schluss zu, dass eine gleichmäßige Verteilung nur dann sinnvoll ist, wenn damit nicht gleichzeitig die Zugriffskollisionen erhöht werden.

#### Optimierung

Der logische nächste Schritt ist die Reduzierung der Zugriffskollisionen, bei gleichzeitigem Beibehalt der gleichmäßigen Verteilung der Berechnungen.

Round Robin Tournament Verteilung mit Locked Resource

Die Verteilung arbeitet, wie die „Locked Resource“-Verteilung, beschrieben auf Seite 35, mit Semaphoren, um die Elemente vor zeitgleichem Zugriff zu schützen. Allerdings wird die auf Seite 39 beschriebene RRT-Matrix verwendet, um die Berechnungspaare möglichst kollisionsfrei zu verteilen.

### Implementierung

for (int step = 0; step < PairLogic.StepCount; step++)

{

for (int pair = 0; pair < PairLogic.PairCount; pair++)

{

// calculate index of selected core

int coreIndex = pair % CoreCount;

// get element index from RRT matrix

int id1 = PairLogic.PairMatrix[step][pair].ID1;

int id2 = PairLogic.PairMatrix[step][pair].ID2;

// execute calculation on selected core

CalculatePair(coreIndex, id1, id2);

}

}

### Messergebnisse

#### Erwartung

Die Verteilung kombiniert eine gleichmäßige Aufspaltung der Berechnung auf die Cores mit einer möglichst kollisionsfreien Ausführung der Berechnungen.  
Dies bedeutet jedoch nicht, dass Kollisionen ausgeschlossen sind.  
In diesem Sinne wird erwartet, dass der Algorithmus schneller als alle bisher getesteten agieren wird.

#### Messung





#### Diskussion

Bei ähnlichem Overhead kann die Verteilung deutlich bessere Ergebnisse für die Messungen mit fixierter und zufälliger Rechenzeit erzielen.

Bei den Auslastungstests führt jedoch die kurze Laufzeit der Berechnungen, gepaart mit hoher Anzahl der Elemente, zu einer Messung, bei der der Overhead den größten Anteil ausmacht und sie damit schlechter, als bei der Referenzverteilung, macht.

#### Optimierung

Sämtliche Vorgehensweisen, welche das Sperren von Ressourcen als Synchronisationsmodel verwenden, liefern nicht zufriedenstellende Ergebnisse bei sowohl Overhead- als auch Auslastungstests.   
Durch Umstellung des Synchronisationsmodels, sollen die Zugriffskollisionen drastisch reduziert werden, ohne dabei Einbußen im Hinblick auf die gleichmäßige Verteilung der Berechnungen hinnehmen zu müssen.

Synchronisierte Round Robin Tournament Verteilung

Die Synchronisierung wird durch das „Divide and Conquer“-Prinzip abgelöst.  
Anstatt einzelne Elemente zu sperren, werden, pro Core, zwei Stapel (Stacks) von Elementen ohne Unterbrechung abgearbeitet.

### Grundstruktur



### Ansteuerung der Cores

Jedes Element des ersten Stapels, mit sämtlichen Elementen des zweiten Stapels berechnet.   
Jeder Core arbeitet sein Stapelpaar ab und wartet dann, bis alle anderen Cores ihre Stapel ebenfalls berechnet haben.

Beispiel, Stapelpaar für einen Core:

Der Core muss, je nach Berechnungsrunde, zuerst die stapelinternen Paare berechnen.  
Anschließend werden die Stapel untereinander verknüpft.

Folgende Paare müssen von dem Core berechnet werden, welcher das Stapelpaar erhält:

|  |  |  |  |
| --- | --- | --- | --- |
| interne Paare | | verknüpfte Paare | |
| 1, 2 |  | 1, 3 |  |
| 3, 4 |  | 1, 4 |  |
|  |  | 2, 3 |  |
|  |  | 2, 4 |  |

### Stapeleigenschaften

Die Elemente in einem Stapel müssen dabei einzigartig Auftreten und dürfen nicht im Stapel von einem anderen Core auftauchen, um einen gleichzeitigen Zugriff zu verhindern.

Da die Anzahl der Berechnungen mit der Fakultät der Elemente steigt und die Elemente in den Stapeln nur einmal auftreten dürfen, können die Berechnungen niemals in nur einem Schritt abgearbeitet werden. Es müssen mehrere Schritte durchgeführt werden, bei denen die Stapel jedes Mal in unterschiedlichen Paarkombinationen auf die Cores aufgeteilt werden. Diese Paarkombinationen werden mit dem RRTA erstellt.  
Nach jedem Schritt werden die Cores synchronisiert.

Beispiel, Stapel für zwei Cores mit insgesamt 8 Elementen:

Valide Stapel, jedes Element kommt nur einmal vor



Invalide Stapel, Element 2 kommt in beiden Stapeln vor, Element 3 fehlt in den Stapeln

### Stapelbildung

Ein Nachteil gegenüber dem Sperren von Elementen ist, dass immer auf alle Cores gewartet werden muss. Um diese Wartezeiten möglichst gering zu halten, werden Stapel-Paare gebildet, welche möglichst gleich schnell abgearbeitet werden können.

#### Struktur

Für die Stapelbildung bedeutet das, dass die RRTA-Matrix nicht mit der Elementanzahl, sondern mit der doppelten Anzahl an Cores erstellt wird.  
Die Anzahl der Stapel ist deshalb die doppelte Anzahl der Cores.

Die Verteilung der Elemente auf die Stapel erfolgt reihum, wobei es auch Stapel geben kann, welche um ein Element größer sind, als andere.

Beispiel, Verteilung von 7 Elementen auf 4 Stapel (2 Cores):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack1 |  | Stack2 |  | Stack3 |  | Stack4 |
| elem1 |  | elem2 |  | elem3 |  | elem4 |
| elem5 |  | elem6 |  | elem7 |  |  |

#### Code

int ElementCount { get; set; }

int UsableCoreCount { get; set; }

ElementType[][] Stacks { get; set; }

private void CreateStacks(ElementType[] parts)

{

int stackCount = UsableCoreCount \* 2;

Stacks = new ElementType[stackCount][];

int stackSize = ElementCount / stackCount;

int leftover = ElementCount % stackCount;

for (int i = 0; i < stackCount; i++)

{

// as there might be numbers of parts which are not divideable cleanly

// the leftovers get added one by one to the first few stacks

// e.g. 6 parts divided by 4 stacks would mean Stacks[0] and Stacks[1] would

// hold 2 values while Stacks[2] and Stacks[3] hold only one

if (i < leftover)

{

Stacks[i] = new ElementType[stackSize + 1];

Array.Copy(parts, i \* stackSize + i, Stacks[i], 0, stackSize + 1);

}

else

{

Stacks[i] = new ElementType[stackSize];

Array.Copy(parts, i \* stackSize + leftover, Stacks[i], 0, stackSize);

}

}

}

### Verteilung der Stapel mit RRTA

Anstatt einzelne Elemente, nach dem RRTA zu verteilen, werden die Stapel durchnummeriert und paarweise, nach dem Algorithmus, auf die Cores verteilt.

Beispiel für den ersten Schritt des RRTA mit drei Cores:

|  |  |  |
| --- | --- | --- |
| RRTA - erster Schritt | | |
| 0 | 1 | 2 |
| 5 | 4 | 3 |

Core 0 würde im ersten Schritt den Stapel 0 und den Stapel 5 berechnen.  
Core 1 würde im ersten Schritt den Stapel 1 und den Stapel 4 berechnen.  
Core 2 würde im ersten Schritt den Stapel 2 und den Stapel 3 berechnen.

### Implementierung

Die überschriebene „Calculate“-Funktion initialisiert die notwendigen Variablen und teilt die Elemente auf möglichst gleich große Stapel auf (Seite 49) und generiert die   
zugehörige RRTA-Matrix (Seite 34).  
Danach wird durch alle Schritte in der Matrix iteriert.

public override void Calculate(ElementType[] elements, GlobalDataType globalData)

{

ElementCount = elements.Length;

GlobalData = globalData;

CreateStacks(elements);

PairLogic.GenerateMatrix(UsableCoreCount \* 2);

for (int step = 0; step < PairLogic.StepCount; step++)

{

CalculateStep(step);

}

}

Ein Schritt besteht dabei aus dem Aufteilen der Stapelpaare auf die zugehörigen Cores und dem abschließenden Synchronisieren. Die CorePool-Klasse wird auf Seite 28 beschrieben.

private void CalculateStep(int step)

{

for (int i = 0; i < UsableCoreCount; i++)

{

CorePool.DistributeCalculation(i, CreateCalculationPair(i, step));

}

CorePool.Synchronize();

}

Die Anweisungen (Seite 28) mit den Stapelpaaren für die Cores werden dabei aus der Matrix generiert. Im ersten Schritt (step == 0) werden auch die internen Berechnungen eines jeden Stapels berechnet.

private PairingData<ElementType, GlobalDataType> CreateCalculationPair(

int coreID, int step)

{

return new PairingData<ElementType, GlobalDataType>(

Stacks[PairLogic.PairMatrix[step][coreID].ID1],

Stacks[PairLogic.PairMatrix[step][coreID].ID2],

GlobalData,

step == 0);

}

### Messergebnisse

#### Messung



#### Diskussion

Die Messung zeigt eine drastische Verbesserung bezüglich der Overhead-Messung. Bei einer großen Anzahl an Elementen ist der Overhead sogar kleiner als beim Referenzalgorithmus, da die statische RRTA-Matrix eine direkte Verteilung über Speicherzugriff und ohne Berechnungs-Operationen zulässt.

Den Einfluss der Generierung der RRTA-Matrix sieht man im Maximalwert der ersten Overhead-Messung mit 16 Elementen. Die folgenden Wiederholungen der Messung können auf Cache-Werte im Prozessor zugreifen und sind so nochmals um einiges schneller.

Bei der fixierten und zufälligen Berechnungsdauer kann kein besonderer Effizienzgewinn festgestellt werden. Es lässt sich jedoch festhalten, dass die Verteilung zumindest relativ ähnliche Ergebnisse wie die Vorgänger liefert und somit auch nicht schlechter geeignet ist.

Beim Auslastungstest fällt wiederum die drastische Overhead-Reduzierung ins Gewicht.  
Zusammen mit einer gleichmäßigen Aufteilung bei kleinstmöglicher Unterbrechung der Cores führt dies zu einer mindestens 3-fach so schnellen Ausführung, im Vergleich zu den anderen Algorithmen.

Fazit

Im allgemeinen Vergleich kann die Synchronisierte RRT-Verteilung vor allem beim Praxisnahen Auslastungstest als der beste Algorithmus zur Parallelisierung angesehen werden.

Es gibt jedoch Extremfälle in denen auch andere Verteilungen Anwendung finden können.  
Bei synchronisierten Verteilungen, führen drastische Ausreißer in Bezug auf die Laufzeit dazu, dass alle Cores lange blockiert werden, während bei einer sperrenden Lösung, die übrigen Cores meistens nicht blockiert werden.  
Es sei angemerkt, dass mit steigender Anzahl an Berechnungselementen auch eine gleichmäßigere statistische Verteilung der Berechnungslaufzeiten einstellt, was dieses Argument hauptsächlich für überschaubare Mengen an Elementen geltend macht.

Ein weiterer Fall ist eine starke Limitierung von Speicher. Um eine RRT-Matrix mit 1024 Elementen zu speichern, werden in einem System mit 32-Bit Integer-Variablen knapp 4Megabyte belegt (4 Byte pro Element \* 1024 Elemente pro Schritt \* 1023 Schritte). Dies kann speziell in eingebetteten Systemen ein Problem darstellen, welches sich nur durch die Wahl einer anderen Parallelisierungsmethode umgehen lässt.

Index

Webservice

Kernelemente 2

<https://web.archive.org/web/20150806181915/http://www.zdnet.com/article/intel-why-a-1000-core-chip-is-feasible/>

<https://www.ll.mit.edu/HPEC/agendas/proc08/Day1/16-Day1-Session2-Reilly-abstract.pdf>

<http://www.eetimes.com/document.asp?doc_id=1167932>

<https://www.chesskid.com/article/view/ask-coach-jessica-chess-tournaments>

<http://www.emba.uvm.edu/~jdinitz/preprints/design_tourney_talk.pdf>

<https://arxiv.org/abs/1205.2367>

<https://dspace.mit.edu/bitstream/handle/1721.1/6935/AITR-633.pdf?sequence=2> // Actors

# Quellenverzeichnis

|  |  |
| --- | --- |
| [1] | J. Clark, 25 12 2010. [Online]. Available: http://www.zdnet.com/article/intel-why-a-1000-core-chip-is-feasible/. |
| [2] | Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, Introduction to Algorithms, Cambridge, Massachusetts: The MIT Press, 2004. |
| [3] | Radu Rugina, Martin Rinard, „Recursion Unrolling for Divide and Conquer Programs,“ in *Languages and Compilers for Parallel Computing*, Cambridge, Massachusetts, Springer, 2001, pp. 34-48. |
| [4] | P. Rezaei, 5 8 2007. [Online]. Available: https://blogs.msdn.microsoft.com/pedram/2007/08/05/dedicated-thread-or-a-threadpool-thread/. |
| [5] | D. Carmona, 6 2002. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms973903.aspx?f=255&MSPPError=-2147217396. |

Anlagen

Teil 1 …………………………………………………………………………………… A-I

Teil 2 …………………………………………………………………………………… A-III

Teil 3 …………………………………………………………………………………… A-V

Anlagen, Teil 1

<inhalt anlage teil1>

Anlagen, Teil 2

<inhalt anlage teil2>

Anlagen, Teil 3

<inhalt anlage teil3>

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

<ort>, den <tag>.<monat>.<jahr>

<unterschrift>

<vorname> <name>