Diplomarbeit

Parallelisierung von ungeordneten, paarweisen Berechnungen

David Hofer, Technische Informatik, KT13wIA-F

# Kurzfassung

Diese Arbeit behandelt den empirischen Vergleich von Verteilungsmethoden für paarweise, ungeordnete Berechnungen. Durch die Verteilung der Berechnungen, soll eine möglichst gute Auslastung des unterliegenden Berechnungssystems erreicht werden.

Zuerst werden paarweise, ungeordnete Berechnungen im Detail beschrieben um dem Leser das notwendige Fundament zu liefern, auf dem die Arbeit aufbaut.

Inhaltsverzeichnis

[Kurzfassung 2](#_Toc492293150)

[Einleitung 5](#_Toc492293151)

[Motivation 5](#_Toc492293152)

[Grundlagen 6](#_Toc492293153)

[*Paarweise, ungeordnete Berechnungen* 6](#_Toc492293154)

[Begriffe 7](#_Toc492293155)

[Berechnungselemente 7](#_Toc492293156)

[Globale Daten 7](#_Toc492293157)

[Core 7](#_Toc492293158)

[Dummy 7](#_Toc492293159)

[Latenz 7](#_Toc492293160)

[Entwicklungsumgebung 8](#_Toc492293161)

[Testumgebung 8](#_Toc492293162)

[Betriebssystem 8](#_Toc492293163)

[Prozessor 8](#_Toc492293164)

[Gegenüberstellung der Algorithmen 9](#_Toc492293165)

[Abstraktion des Systems 10](#_Toc492293166)

[Probleme bei der Verteilung von Berechnungen 10](#_Toc492293167)

[Ressourcenzugriff 10](#_Toc492293168)

[Synchronisation 10](#_Toc492293169)

[Overhead 10](#_Toc492293170)

[Mehrfachberechnung 10](#_Toc492293171)

[Lösungsansätze 11](#_Toc492293172)

[Validierung der Algorithmen 11](#_Toc492293173)

[Resource Lock 12](#_Toc492293174)

[Deadlock-Gefahr 12](#_Toc492293175)

[Diskussion 12](#_Toc492293176)

[Round Robin Tournament Algorithmus 13](#_Toc492293177)

[Vorgehensweise 13](#_Toc492293178)

[Diskussion 14](#_Toc492293179)

[Deepcopy 14](#_Toc492293180)

[Diskussion 14](#_Toc492293181)

[Unterliegende Systeme 15](#_Toc492293182)

[Single Thread 15](#_Toc492293183)

[Threadspawning 15](#_Toc492293184)

[Actors 15](#_Toc492293185)

[.NET Threadpool 15](#_Toc492293186)

[Testsetup 16](#_Toc492293187)

[Überblick 16](#_Toc492293188)

[Initialisierung 16](#_Toc492293189)

[Ablauf 16](#_Toc492293190)

[Performance-Messung 16](#_Toc492293191)

[Overhead-Anteil 17](#_Toc492293192)

[Implementierung 17](#_Toc492293193)

[Grundstruktur 17](#_Toc492293194)

[Output-Validierung 18](#_Toc492293195)

[Zeitmessung mit fixierter Rechenzeit 22](#_Toc492293196)

[Systemabstraktion 23](#_Toc492293197)

[Core-Pool 24](#_Toc492293198)

[Berechnungsanweisung 24](#_Toc492293199)

[Synchronisation 24](#_Toc492293200)

[Grundstruktur der Verteilungen 25](#_Toc492293201)

[Algorithmus-Optimierung 26](#_Toc492293202)

[Validierungs-Test 26](#_Toc492293203)

[Effizienz-Test 26](#_Toc492293204)

[Single-Thread Referenz Algorithmus 27](#_Toc492293205)

[Implementierung 27](#_Toc492293206)

[Messergebnisse 28](#_Toc492293207)

[Parallelisierung durch Resource Lock 29](#_Toc492293208)

[Implementierung 29](#_Toc492293209)

[Messergebnisse 30](#_Toc492293210)

[Kollisionsreduktion durch Round Robin Tournament Algorithmus 31](#_Toc492293211)

[Links 32](#_Toc492293212)

[Selbstständigkeitserklärung 33](#_Toc492293213)

# 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. 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.

“… the rapid increase in single processor performance is likely a thing of the past. Future advancements in computing capability must come from harnessing more cores to a single solution.“

# 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 Performancetests 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.

## Abkürzungen

RRTA - Round Robin Tournament Algorithmus

# 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 und aussagekräftige 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.

# Abstraktion des Systems

# 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.

## Resource Lock

Ein Standardverfahren, um Prozesse zu Parallelisieren.   
Jede Ressource wird einzeln vor Zugriffen geschützt. Wenn ein Core eine Berechnung durchführt, werden zuerst beide Ressourcen, welche für die Berechnung benötigt werden, für alle anderen Cores gesperrt. Sobald die Berechnung beendet ist, gibt der Core die Ressourcen 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

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.

# Unterliegende Systeme

## Single Thread

Es wird nur ein einziger Thread für die Ausführung verwendet. Somit kann auch nur ein Core des Systems ausgelastet werden.  
Wird verwendet um einen Referenzwert zu erhalten.

## Threadspawning

Bei jedem Aufruf 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.  
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.

<https://blogs.msdn.microsoft.com/pedram/2007/08/05/dedicated-thread-or-a-threadpool-thread/>

<https://msdn.microsoft.com/en-us/library/ms973903.aspx>

# 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 Performance 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.

### Performance-Messung

Die Performance-Messung wird ohne Output Validierung durchgeführt, da die Prüfung zusätzlichen Rechenaufwand verursacht, welcher die Zeitmessung verfälscht.  
Wie auf Seite

### Overhead-Anteil

Um den Overhead effektiv feststellen zu können, wird den Berechnungen eine fixe Laufzeit gegeben.  
Es muss zwischen Overhead, der zusätzliche Last auf dem Prozessor verursacht, und Overhead, der den Prozessor nicht zusätzlich auslastet, sondern nur die Berechnungen verlangsamt, unterschieden werden.

## Implementierung

Im folgenden Abschnitt wird die Implementierung der Tests beschrieben.

### 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

##### 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;

}

}

#### Testszenario

Behandelt die Implementierung des Validierungstests.

##### 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;

}

}

### Zeitmessung mit fixierter Rechenzeit

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

#### Struktur



Unterschiede zur Basisklasse sind der vordefinierte Typ, welcher wieder willkürlich auf Integer festgelegt wurde, weil er nicht zum Test beiträgt.

Der wichtige Teil ist die überschriebene Berechnungsfunktion „CalculationFunction“, welche bei jedem Element-Paar durchgeführt wird. Diese muss eine fixe Laufzeit haben, um zum Beispiel den Overhead der Verteilung berechnen zu können.

#### Code

class FixedDurationTest : UniquePairTest<int, int>

{

public FixedDurationTest(

UniquePairDistribution<int, int> distribution,

int dummyCount) :

base(distribution)

{

// initialize elements

m\_elements = new int[dummyCount];

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

{

m\_elements[i] = i;

}

}

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

{

// fixed duration of the execution

Thread.Sleep(100);

}

}

# 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 | | | |
|
|  |  |  |  |
| Verteilungssystem | | | |
|
|  |  |  |  |
| Berechnungssystem | | | |
|

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.

## Grundstruktur der Verteilungen

Die Basisklasse, von welcher sämtliche Verteilungsalgorithmen erben. Sie beinhaltet den Core-Pool, auf dem die Berechnungen durchgeführt werden und eine abstrakte Funktion für das Starten der Verteilung.



In der tatsächlichen Implementierung wird der Core-Pool mittels Constructor-Injection übergeben und über die Getter-Funktion nach außen sichtbar gemacht. Dies verhindert, dass der Core-Pool nach dem Erstellen ausgetauscht werden kann.

public abstract class UniquePairDistribution<PartType, GlobalDataType>

{

protected CorePool<PartType, GlobalDataType> m\_corePool;

public CorePool<PartType, GlobalDataType> CorePool { get { return m\_corePool; } }

public UniquePairDistribution(CorePool<PartType, GlobalDataType> pool)

{

m\_corePool = pool;

}

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

}

„Calculate“ ist die einzige Funktion, die von der Applikation aufgerufen wird. Es werden alle Elemente, zwischen denen Berechnungen durchgeführt werden müssen, zusammen mit den globalen Daten übergeben.  
Es muss gewährleistet sein, dass nach dem Aufruf sämtliche Berechnungen vollständig abgeschlossen sind.

# 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.

## Validierungs-Test

Jede Verteilung muss mehrere Validierungs-Setups bestehen.

<insert conditions here>

## Effizienz-Test

Um einen gemeinsamen Nenner für die Effizienz der Verteilungen zu erhalten, werden sämtliche Tests mit den folgenden Bedingungen durchgeführt:

Anzahl Cores: 4  
Elemente: 16  
Daraus resultierende Anzahl der Berechnungen: 120 (Berechnung auf Seite 6  
Laufzeit einer Berechnung: 10ms

Der Test wird 10-mal wiederholt um Minimum-. Maximum- und Durchschnittswert für die Dauer der Testläufe zu erhalten.

## Single-Thread Referenz Algorithmus

Dieser Algorithmus wurde auf Seite 12 beschrieben.   
Es wird nur ein Core für die Berechnung verwendet.

### Implementierung



In der auf Seite 26 beschriebenen Grundstruktur wird lediglich die „Calculate“-Funktion überschrieben.

In der Implementierung muss der Algorithmus an die Schnittstellen des Core-Pools angepasst werden. Dabei wird der Vorgang jedoch nur syntaktisch abgewandelt, das Grundschema der Verteilung bleibt gleich.

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

{

// initialize arrays with just one entry

PartType[] element1 = new PartType[1];

PartType[] elememt2 = new PartType[1];

PairingData<PartType, GlobalDataType> pair;

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

{

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

{

// Set values in the arrays

element1[0] = elements[i];

elememt2[0] = elements[j];

// Create instruction for the core pool

pair = new PairingData<PartType, GlobalDataType>(

element1, elememt2, globalData, false);

// Calculate the instruction on Core 0

m\_corePool.DistributeCalculation(0, pair);

// Wait until Core 0 finished

m\_corePool.Synchronize(0);

}

}

}

### Messergebnisse

#### Erwartung

Bei einem perfekten Durchlauf ohne Overhead würden alle 120 Berechnungen in Serie durchgeführt und somit eine Dauer von 120 \* 10ms = 1200ms haben.

#### Messung

|  |  |
| --- | --- |
| Single Thread Reference | |
| Minimum[ms] | 1271.3972 |
| Maximum[ms] | 1293.3019 |
| Durchschnitt[ms] | 1282.452 |

#### Diskussion

Die Ergebnisse weichen im erwarteten Rahmen von der Vorhersage ab. Der Overhead beschränkt sich auf maximal ~100ms, was bei 120 Aufrufen der Berechnungsfunktion als auch der Synchronisationfunktion argumentierbar ist.

## 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 < n - 1; i++)

{

// inner loop, distributing to multiple cores

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

{

firstElement = elements[i];

secondElement = elements[j];

Calculation(firstElement, secondElement);

}

}

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

ausgeführt.

Pseudocode – roter Teil wird auf die Cores verteilt:

// outer loop

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

{

int coreIndex = i % CoreCount;

ExecuteInCore(coreIndex)

{

// inner loop

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

{

firstElement = elements[i];

secondElement = elements[j];

Calculation(firstElement, secondElement);

}

}

}

### Messergebnisse

#### Erwartung

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 < n - 1; i++)

{

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

{

firstElement = elements[i];

secondElement = elements[j];

Calculation(firstElement, secondElement);

}

}

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  
  
Der Idealfall ohne Overhead wäre also, dass die Berechnungen nach 36 \* 10ms = 360 ms abgeschlossen wären.

#### Messung

|  |  |
| --- | --- |
| Resource Lock | |
| Minimum[ms] | 568.3314 |
| Maximum[ms] | 995.8116 |
| Durchschnitt[ms] | 709.9585 |

#### Diskussion

Die Messung zeigt zwei Probleme auf.   
Die Maximaldauer ist fast doppelt so lang wie die Minimaldauer. Dies bedeutet, dass der Algorithmus eine hohe Varianz aufweist.  
Außerdem weicht auch der Durchschnittswert weit von dem Erwartungswert ab.

Ein Problem sind die Zugriffskollisionen, bei denen sich die Cores gegenseitig die benötigten Ressourcen sperren, und sich somit blockieren.

Das zweite Problem ist die ungleichmäßige Verteilung auf die Cores.

## Gleichmäßige Verteilung der Berechnungen

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 < n - 1; i++)

{

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

{

coreSelect++; // increment counter

// calculate index of selected core

int coreIndex = coreSelect % CoreCount;

// execute calulation on selected core

CalculatePair(coreIndex, i, j);

}

}

### Messergebnisse

#### Erwartung

Durch die gleichmäßige Verteilung, sollte die Durchschnittsdauer geringer sein. Die Varianz hingegen bleibt wahrscheinlich ähnlich drastisch, wie sie es schon im vorherigen Durchlauf mit „Resource Lock“ war.  
  
Bei 16 Elementen und 120 Berechnungen werden nun die Berechnungen wie folgt verteilt:

Core 0: 30  
Core 1: 30  
Core 2: 30  
Core 3: 30

Der Idealfall ohne Overhead wäre also, dass die Berechnungen nach 30 \* 10ms = 300 ms abgeschlossen wären.

#### Messung

|  |  |
| --- | --- |
| Resource Lock | |
| Minimum[ms] | 568.3314 |
| Maximum[ms] | 995.8116 |
| Durchschnitt[ms] | 709.9585 |

#### Diskussion

Der erwartete Durchschnittswert weicht wieder signifikant vom erwarteten Wert ab.  
Der Durchlauf dauert im Schnitt sogar etwas länger, als es bei der ungleichmäßigen Verteilung „Locked Resource“ auf Seite 30 der Fall war.

Eine gleichmäßige Aufteilung kann also sogar einen negativen Effekt bringen, wenn dadurch die Zugriffskollisionen vermehrt auftreten.

## Kollisionsreduktion durch Round Robin Tournament Algorithmus

Durch eine effizientere Verteilung können die Kollisionen reduziert werden. Der Round Robin Tournament Algorithmus bildet dabei das Fundament. Er ist darauf ausgelegt, dass möglichst viele Paare gleichzeitig gebildet werden können.

### Round Robin Tournament Implementierung

Die grundsätzliche Funktionsweise wird auf Seite 13 beschrieben.

Um die Verteilung erweitern zu können, wird eine Matrix erzeugt, welche für jeden Aufruf das jeweilige Elementpaar zurückliefert, welches berechnet werden soll. Dies bildet das Herzstück für alle Implementierungen, welche auf dem Round Robin Tournament Algorithmus beruhen.



Die Funktionen werden absichtlich nicht in ihrer Reihenfolge erläutert, um möglichst einfach die Erstellung der Matrix zu erklären.

#### Basis-Array

Das Basis-Array bildet den RRTA im ersten Schritt ab. Im Speicher wird es als Folge 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.

Verteilung

# Links

<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

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwen-dung 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>