Inhalt

Inhalt I

Abbildungsverzeichnis VI

Tabellenverzeichnis VII

Formelverzeichnis VIII

Abkürzungsverzeichnis IX

1 Einleitung 1

1.1 Motivation 1

1.2 Zielsetzung 1

1.3 Begriffe 2

1.3.1 Berechnungselemente 2

1.3.2 Globale Daten 2

1.3.3 Verteilung 2

1.3.4 Dummy 3

1.3.5 Latenz 3

1.3.6 Konkurrenz 3

1.3.7 Zugriffskollision 3

2 Parallelisierungsprobleme 4

2.1 Ressourcenzugriff 4

2.2 Synchronisation 4

2.3 Overhead 4

2.4 Mehrfachberechnung 5

2.5 Deadlock 5

Sequentieller Algorithmus 6

3 Ressourcensperrung 7

3.1 Vorgehensweise 7

3.2 Merkmale 7

4 Schleifen-Parallelisierung 8

4.1 Aufteilung in innere und äußere Schleife 8

4.2 Auflösen der Schleifen 9

5 Verteilung durch Laufvariablenmodulo 10

5.1.1 Verteilung durch Round Robin Tournament Algorithmus 11

6 Round Robin Tournament Algorithmus 12

6.1 Vorgehensweise 12

6.2 Round Robin Tournament Matrix 14

6.2.1 Basis-Array 14

6.2.2 Array-Verschiebung 14

6.2.3 Matrix-Generierung 15

6.3 Umsetzung auf die Problemstellung 16

6.4 Merkmale 16

7 Divide and Conquer 17

7.1 Realisierung durch RRTA 18

7.2 Stapelbildung 19

7.3 Merkmale 20

8 Implementierungskomponenten des Testsystems 21

8.1 Entwicklungsumgebung 21

8.2 Betriebssystem 21

8.3 Parallelisierung unter Windows 22

8.3.1 Threads 22

8.3.2 Actors 22

8.3.3 .NET Threadpool 22

8.4 Relevante Hardware-Spezifikationen 23

9 Testsetup 24

9.1 Überblick 24

9.1.1 Initialisierung 24

9.1.2 Ablauf 25

9.2 Grundstruktur 25

9.3 Output-Validierung 26

9.3.1 Validierungs-Input 26

9.3.1.1 Struktur 26

9.3.1.2 Code 28

9.3.2 Test-Ablauf 29

9.3.2.1 Struktur 29

9.3.2.2 Code 30

9.4 Effizienz-Messung 31

9.4.1 Struktur 31

9.4.2 Code 31

9.4.3 Overhead-Messung 32

9.4.4 Zeitmessung mit fixierter Rechenzeit 32

9.4.5 Zeitmessung mit zufälliger Rechenzeit 33

9.4.6 Zeitmessung mit Auslastung 33

10 Systemabstraktion 34

10.1 Prozessorkern-Pool 35

10.1.1 Berechnungsanweisung 35

10.1.2 Synchronisation 36

11 Verteilungsstrukturen 37

11.1 Grundstruktur 37

11.1.1 Struktur 37

11.1.2 Code 37

11.2 Prozessorkern-Pool-Verteilung 38

11.2.1 Struktur 38

11.2.2 Code 39

11.3 Eigenständige Verteilung 39

11.3.1 Struktur 40

11.3.2 Code 40

11.4 Lock-Verteilung 41

11.4.1 Struktur 41

11.4.2 Code 42

12 Algorithmus-Optimierung 43

12.1 Testroutinen 43

12.1.1 Validierung 43

12.1.2 Effizienz 44

12.1.2.1 Overhead 1 44

12.1.2.2 Overhead 2 44

12.1.2.3 Overhead 3 45

12.1.2.4 Fixierte Rechenzeit 1 46

12.1.2.5 Fixierte Rechenzeit 2 46

12.1.2.6 Zufällige Rechenzeit 1 46

12.1.2.7 Zufällige Rechenzeit 2 46

12.1.2.8 Auslastung 1 47

12.1.2.9 Auslastung 2 47

12.1.2.10 Auslastung 3 47

12.2 Single-Thread Referenz Algorithmus 48

12.2.1 Implementierung 48

12.2.2 Messergebnisse 49

12.2.2.1 Diskussion 49

12.2.2.2 Optimierung 49

12.3 Parallelisierung durch Locked Resource 50

12.3.1 Implementierung 50

12.3.2 Messergebnisse 51

12.3.2.1 Messung 51

12.3.2.2 Diskussion 51

12.3.2.3 Optimierung 52

12.4 Gleichmäßige Verteilung der Berechnungen 53

12.4.1 Implementierung 53

12.4.2 Messergebnisse 54

12.4.2.1 Messung 54

12.4.2.2 Diskussion 54

12.4.2.3 Optimierung 54

12.5 Round Robin Tournament Verteilung mit Locked Resource 55

12.5.1 Implementierung 55

12.5.2 Messergebnisse 56

12.5.2.1 Erwartung 56

12.5.2.2 Messung 56

12.5.2.3 Diskussion 56

12.5.2.4 Optimierung 56

12.6 Synchronisierte Round Robin Tournament Verteilung 57

12.6.1 Grundstruktur 57

12.6.2 Stapelbildung 57

12.6.3 Verteilung der Stapel mit RRTA 59

12.6.4 Implementierung 60

12.6.5 Messergebnisse 61

12.6.5.1 Messung 61

12.6.5.2 Diskussion 61

13 Ergebnis 62

13.1 Fazit 62

13.2 Ausblick und weiterführende Forschung 63

13.2.1 Testumfang 63

13.2.2 Deadlock-Analyse 63

13.2.3 Optimierung 63

13.2.4 Auslagerung auf externe Prozessoren 63

Quellenverzeichnis 64

Anlagen 66

Anlagen, Validierungsergebnisse 67

Selbstständigkeitserklärung 69

Abbildungsverzeichnis

[Abbildung 1: System im Deadlock 5](#_Toc497739410)

[Abbildung 2: Ressourcensperrung 7](file:///D:\dhdev\DA\Diplomarbeit-Textteil.docx#_Toc497739411)

[Abbildung 3: RRTA - erste Runde 12](#_Toc497739412)

[Abbildung 4: RRTA - zweite Runde 12](#_Toc497739413)

[Abbildung 5: RRTA - letzte Runde 12](#_Toc497739414)

[Abbildung 6: RRTA - Nullrunde bei ungerader Teilnehmerzahl 13](#_Toc497739415)

[Abbildung 7: RRTA – Basisarray und Speicherabbild 14](#_Toc497739416)

[Abbildung 8: RRTA - Schrittweise Verschiebung 14](file:///D:\dhdev\DA\Diplomarbeit-Textteil.docx#_Toc497739417)

[Abbildung 9: RRTA - Alle Schritte bei vier Spielern 15](file:///D:\dhdev\DA\Diplomarbeit-Textteil.docx#_Toc497739418)

[Abbildung 10: RRTA-Matrix für vier Spieler 15](#_Toc497739419)

[Abbildung 11: RRTA – Aufteilung auf Prozessorkerne 16](#_Toc497739420)

[Abbildung 12: Divide and Conquer mittels RTTA 19](#_Toc497739421)

[Abbildung 13: Valide Stapel für zwei Prozessorkerne mit insgesamt 8 Elementen 19](#_Toc497739422)

[Abbildung 14: Invalide Stapel für zwei Prozessorkerne mit insgesamt 8 Elementen 19](#_Toc497739423)

[Abbildung 15: Valide Stapel für zwei Prozessorkerne mit insgesamt 7 Elementen 20](#_Toc497739424)

[Abbildung 16: Testsuite 24](#_Toc497739425)

[Abbildung 17: Systemabstraktion 34](#_Toc497739426)

[Abbildung 18: Austauschbares Berechnungssystem 34](#_Toc497739427)

Tabellenverzeichnis

[Tabelle 1: Beispiel - Stapelpaar für einen Prozessorkern 18](#_Toc497739428)

[Tabelle 2: Beispiel - Elementpaare zwischen den Stapeln 18](#_Toc497739429)

[Tabelle 3: Validierungsarray - initialer Zustand 26](#_Toc497739430)

[Tabelle 4: Validierungsarray - Einzelvalidierung 27](#_Toc497739431)

[Tabelle 5: Validierungsarray - valider Zustand 27](#_Toc497739432)

[Tabelle 6: Validierungsarray - Doppelberechnung 27](#_Toc497739433)

[Tabelle 7: Berechnungsanweisung mit Stacks 36](#_Toc497739434)

[Tabelle 8: Messergebnisse - Referenz-Verteilung 49](#_Toc497739435)

[Tabelle 9: Messergebnisse - Ressourcensperrende Verteilung 51](#_Toc497739436)

[Tabelle 10: Messergebnisse - Gleichmäßig Sperrende Verteilung 54](#_Toc497739437)

[Tabelle 11: Messergebnisse - Sperrende RRT-Verteilung 56](#_Toc497739438)

[Tabelle 17: Stapelverteilung mit der RRT-Matrix 59](#_Toc497739439)

[Tabelle 18: Messergebnisse - Synchronisierte RRT-Verteilung 61](#_Toc497739440)

Formelverzeichnis

[Formel 1: Mögliche Kombinationen mit unbestimmten Teilmengen 6](#_Toc497739441)

[Formel 2: Mögliche Kombinationen mit Paaren 6](#_Toc497739442)

[Formel 3: Schleifenauflösung – Aufteilung mittels Laufvariablenmodulo 10](#_Toc497739443)

[Formel 4: Schleifenauflösung – Einzigartige Laufvariable 10](#_Toc497739444)

[Formel 5: RRTA Rundenanzahl - gerade Anzahl an Teilnehmer 13](#_Toc497739445)

[Formel 6: RRTA Rundenanzahl - ungerade Anzahl an Teilnehmer 13](#_Toc497739446)

Abkürzungsverzeichnis

**RRT** - Round Robin Tournament

**RRTA** - Round Robin Tournament Algorithmus

**DCA** - Divide and Conquer Algorithmus

Einleitung

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 Objekten ein solches Berechnungsmuster aufweisen. Eine Optimierung bzw. Beschleunigung solcher Vorgänge kann eine erhebliche Zeitersparnis bedeuten.   
Wenn die Simulation in Echtzeit berechnet wird, kann ein Leistungsgewinn, durch Parallelisierung, dabei helfen, stabile Frameraten zu generieren.

Zielsetzung

Paarweise, ungeordnete Berechnungen, ein vom Autor gewählter Ausdruck, ergeben 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.

Es sollen mögliche Parallelisierungsmethoden auf ihre Kompatibilität mit dieser Problemstellung untersucht werden. Es wird von einem lokalen Prozessorsystem ausgegangen, welches auf einen gemeinsamen Speicher Zugriff hat.  
Die Parallelisierungsmethoden sollen, durch empirische Tests, in möglichst individuellen 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.

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.

### Verteilung

Beschreibt einen Algorithmus, welcher die Berechnungen auf die Prozessorkerne aufteilt.

### 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 Prozessorkernen. Dauert die Kommunikation, und damit auch die Synchronisation, zwischen den Prozessorkernen lange, bedeutet dies eine hohe Latenz.

### Konkurrenz

Zwei oder mehrere Prozessorkerne befinden sich in Konkurrenz, wenn sie eine gemeinsame Ressource, welche nur ein Prozessorkern gleichzeitig verwenden darf, benötigen.

### Zugriffskollision

Wenn ein Konkurrenzzustand zwischen Prozessorkerne herrscht und trotzdem zeitgleich ein Zugriff auf eine Ressource von mehreren Prozessorkernen erfolgt, spricht man von einer Zugriffskollision.

Parallelisierungsprobleme

Dieses Kapitel, soll den Leser dafür sensibilisieren, welche grundlegenden Hürden, bei der Parallelisierung dieser Problemstellung, zu überwinden sind.

## 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 Parallelisierung, muss das Ergebnis der Prozessorkerne anschließend synchronisiert werden. Dies bedeutet, dass entweder Zwischenergebnisse zwischen den Prozessorkernen ausgetauscht werden, oder das Endergebnis der Prozessorkerne 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 Parallelisierung 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 Prozessorkernen, 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.

Deadlock

Da ein gleichzeitiger Zugriff auf die Ressourcen nicht erlaubt ist, die Berechnung jedoch mehr als eine Ressource benötigt, ist das System gefährdet, in einem unlösbaren, verklemmten Zustand zu enden (Deadlock).

Die folgenden vier Bedingungen müssen **alle** erfüllt sein, damit ein System in einem Deadlock enden kann [2]:

* Ressourcen können nur von einem Prozessorkern gleichzeitig verwendet werden
* Ressourcen können einem Prozessorkern nicht entzogen werden
* Prozessorkerne fordern Ressourcen, während sie schon eine Ressource belegen
* Kreisförmige Anordnung von Ressourcen-Zuteilung und gleichzeitigem Anfordern von zwei oder mehr Prozessorkerne



Abbildung : System im Deadlock

Sequentieller Algorithmus

Um sämtliche Berechnungen, in sequentieller Abfolge, durchzuführen, 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, oder nicht die gleiche Anzahl an Berechnungen, ausführen, muss er als fehlerhaft betrachtet werden.

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

n … Anzahl der Elemente  
k … Anzahl der Elemente welche für eine Berechnung benötigt werden

Formel : Mögliche Kombinationen mit unbestimmten Teilmengen

Für die paarweise Variante ist k immer 2

Formel : Mögliche Kombinationen mit Paaren

Der Algorithmus wird außerdem als Grundstein, für die folgenden Parallelisierungsmethoden verwendet.

Ressourcensperrung

Um Ressourcen, vor konkurrierenden Prozessorkernen, zu schützen, wird jede Ressource gesperrt, sobald ein Kern darauf zugreift. Dies verhindert, dass ein anderer Prozessorkern ebenfalls darauf zugreifen kann, solange die Ressource noch bearbeitet wird.  
Das Betriebssystem muss dafür die unterliegenden Strukturen bereitstellen (Mutex, Lock, Semaphor) [2].

Vorgehensweise

Wenn ein Prozessorkern eine Berechnung durchführt, werden zuerst beide Elemente, welche für die Berechnung benötigt werden, für alle anderen Prozessorkerne gesperrt. Sobald die Berechnung beendet ist, gibt der Prozessorkern die Elemente wieder frei.

Prozessorkern 1 sperrt Element 1 und Element 2 und führt die Berechnung zwischen ihnen durch.  
  
Prozessorkern 2 will die Berechnung zwischen Element 2 und Element 3 ausführen, muss jedoch warten bis Prozessorkern 1 das Element 2 wieder freigibt.

Abbildung : Ressourcensperrung

Merkmale

Diese Methode verhindert, dass mehrere Prozessorkerne zeitgleich auf eine Ressource zugreifen können. Passiert es jedoch, dass zwei oder mehrere Prozessorkerne die gleiche Ressource benötigen, wird von einer Zugriffskollision gesprochen.  
Diese Kollision beeinflusst zwar den logischen Ablauf nicht, kann aber für erhebliche Leistungseinbußen sorgen, was die Laufzeit der Berechnung betrifft.  
Mit steigender Anzahl von Berechnungselementen, nimmt die Anzahl der theoretisch möglichen Kollisionen exponentiell zu.

Schleifen-Parallelisierung

Da die Problemstellung in eine doppelte Schleife übersetzt werden kann, lässt sie sich mittels Schleifen-Parallelisierung angehen.  
Dabei werden einzelne Schleifendurchläufe aufgeteilt und Prozessorkernen zugeordnet [3].

Aufteilung in innere und äußere Schleife

Für einen einfachen, ersten Schritt, können die Schleifen in eine innere und eine äußere Schleife aufgeteilt 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++)

{

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

}

}

Die äußere Schleife wird anschließend auf die Prozessorkerne verteilt, sodass eine innere Schleife immer komplett von diesem abgearbeitet wird.

Prozessorkern 1:

for (int j = 0 + 1; j < n; j++)

{

Calculation(elements[0], elements[j]);

}

Prozessorkern 2:

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

{

Calculation(elements[1], elements[j]);

}

Usw.

Man beachte, dass in dem Beispiel, der Schleifenindex „i“ durch den tatsächlichen Index ersetzt wurde.   
Diese Zuordnung wird wiederholt bis sämtliche Durchläufe der äußeren Schleife abgearbeitet sind. Übersteigt die Anzahl der Schleifendurchläufe die Anzahl der Prozessorkerne, beginnt die Zuteilung wieder beim ersten Prozessorkern.

Problematisch ist die ungleichmäßige Aufteilung der Last. In dem Beispiel, würde der zweite Prozessorkern die Schleife weniger oft durchlaufen, als der erste (unterschiedlicher Start-Index „j“ bei gleichem End-Index „n“). Angenommen, die Berechnungen innerhalb einer Schleife haben die gleiche Laufzeit, bedeutet dies, dass der erste Prozessorkern mehr Zeit benötigt, um die Berechnung durchzuführen. Dies wirkt sich negativ auf die Gesamtlaufzeit der Berechnung aus, da der Vorgang erst zum Aufrufer zurückkehren darf, wenn wirklich alle Ergebnisse vorliegen.

Beachtet werden muss außerdem, dass bei dieser Art der Parallelisierung, nicht ausgeschlossen werden kann, dass Prozessorkerne gleichzeitig auf eine Ressource (Berechnungselement) zugreifen.   
Um dies zu verhindern, muss zum Beispiel mit Ressourcensperrung gearbeitet werden, was wiederum laufzeitenverlängernde Zugriffskollisionen ermöglicht.

Zusammenfassend lässt sich festhalten, dass eine Schleifen-Parallelisierung, mittels innerer und äußerer Schleife, Simplizität auf Kosten von Optimierung anbietet.

Auflösen der Schleifen

Um eine ungleichmäßige Verteilung auf die Prozessorkerne zu verhindern, werden die Schleifen noch granulierter aufgeteilt. Die Prozessorkerne bearbeiten dabei nicht mehr ganze Schleifen, sondern berechnen Einzelergebnisse.

Prozessorkern 1:

Calculation(elements[0], elements[1]);

Prozessorkern 2:

Calculation(elements[0], elements[2]);

Usw.

Die feinere Unterteilung bedeutet in weiterer Folge einen erhöhten Overhead, was die Verteilung auf die Prozessorkerne betrifft.

Es ist außerdem nicht garantiert, dass Prozessorkerne nicht gleichzeitig auf Ressourcen zugreifen, weshalb diese mittels Ressourcensperrung geschützt werden müssen.

Die Aufteilung selbst, kann auf unterschiedliche Weise erfolgen. Ziel ist es jedoch, eine gleichmäßige Verteilung der Last zu erreichen und Zugriffskollisionen zu minimieren.

Verteilung durch Laufvariablenmodulo

Um die Einzelberechnungen an Prozessorkerne zu verteilen, kann das Modulo über die Laufvariable gebildet werden.

Formel : Schleifenauflösung – Aufteilung mittels Laufvariablenmodulo

Für die doppelte Schleife der Problemstellung, muss für jede Kombination der Laufvariablen ein einzigartiges Ergebnis gebildet werden. Dafür kann, die Laufvariable der äußeren Schleife, mit der Elementanzahl multipliziert und anschließend, mit der Laufvariable der inneren Schleife, addiert werden.

Formel : Schleifenauflösung – Einzigartige Laufvariable

In Kombination, kann jeder Schleifendurchlauf, einzeln, auf die Prozessorkerne verteilt werden.

n… Anzahl der Elemente  
coreCount... Anzahl der Prozessorkerne

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

{

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

{

int uniqueIndex = i \* n + j;

int coreIndex = uniqueIndex % coreCount;

<Calculate elements with index i and j on core with index coreIndex>

}

}

Durch diese Art der Verteilung auf die Prozessorkerne, wird eine gleichmäßige Auslastung erreicht, wenn davon ausgegangen werden kann, dass die Einzelberechnungen selbst, ähnliche Laufzeiten aufweisen.  
Eine Reduzierung der Zugriffskollisionen, ist jedoch nicht angedacht und bietet Potential für Verbesserungen.

### Verteilung durch Round Robin Tournament Algorithmus

Der Round Robin Tournament Algorithmus wird im folgenden Abschnitt beschrieben und enthält unter dem Kapitel **6.3 Umsetzung auf die Problemstellung** eine Erläuterung zu diesem Thema.

Round Robin Tournament Algorithmus

Der Round Robin Tournament Algorithmus (**RRTA**) wurde ursprünglich für das Zuweisen von Spielpartnern, bei zum Beispiel Schachturnieren, entwickelt [4]. Die Besonderheit ist dabei, dass jeder Teilnehmer, gegen jeden anderen Teilnehmer, genau einmal spielt.   
Die Verteilung sorgt dafür, dass pro „Spielrunde“ die **maximal mögliche 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 0 gegen 7, Spieler 1 gegen 6, usw.



Abbildung : RRTA - erste Runde

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 0 gegen 1, Spieler 2 gegen 7, …).



Abbildung : RRTA - zweite Runde

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



Abbildung : RRTA - letzte Runde

Die Aufteilung kann nur mit einer geraden Anzahl an Spielern durchgeführt werden.  
Bei einer ungeraden Anzahl an Spielern, muss ein Spielerindex als Nullrunde angesehen werden. Alle Spieler, welche mit dem Nullrunden-Index ein Paar bilden, müssen in der Runde aussetzen. In der Praxis, bedeutet dies, dass in jeder Runde ein Spieler aussetzt.

In der folgenden Abbildung, könnte beispielsweise der Index 0 als Nullrunden-Index gewählt werden. Im ersten Schritt, des RRTA, würde somit der Spieler mit Index 7 aussetzen. Im zweiten Schritt, der Spieler mit Index 1 aussetzen.

 

Abbildung : RRTA - Nullrunde bei ungerader Teilnehmerzahl

Die Rundenzahl ist abhängig von der Anzahl der Teilnehmer.  
Die Anzahl der benötigten Runden beträgt für:

**N = gerade** **Teilnehmerzahlen**:  
Benötigte Runden = N – 1

Formel : RRTA Rundenanzahl - gerade Anzahl an Teilnehmer

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

Formel : RRTA Rundenanzahl - ungerade Anzahl an Teilnehmer

Round Robin Tournament Matrix

Um alle Schritte des RRTA in einer Struktur abbilden zu können, kann eine Matrix erzeugt werden, welche für jeden Aufruf das jeweilige Spielerpaar zurückliefert, welches gegeneinander spielen soll.  
Die Matrix sorgt dafür, dass der RRTA nur eimal durchlaufen werden muss und die Ergebnisse im Speicher abgelegt werden können. Dies führt inheränt dazu, dass auch der Cache des Prozessors besser genutzt werden kann.

### Basis-Array

Das Basis-Array bildet den RRTA im ersten Schritt (step 0) ab. Im Speicher kann es als Array von Integer-Werten dargestellt.  
  
Als Beispiel der erste Schritt des RRTA mit sechs Spielern und das zugehörige Interger-Array.  
Paare sind farblich gekennzeichnet.

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

Abbildung : RRTA – Basisarray und Speicherabbild

### Array-Verschiebung

Die Array-Verschiebung, bringt 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.

  
  
🡪Verschiebung 🡪

Abbildung : RRTA - Schrittweise Verschiebung

Diese Verschiebung wird so oft angewandt, bis alle Zustände der RRT-Matrix durchlaufen sind.

### Matrix-Generierung

Aus dem Basis-Array und dessen Verschiebungen, kann die gesamte RRTA-Matrix erzeugt werden.

Für eine Elementanzahl von vier würden die RRTA-Schritte wie folgt aussehen:



Verschiebung🡪 Verschiebung🡪

Abbildung : RRTA - Alle Schritte bei vier Spielern

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.

* RRTA-Matrix[step2][pair0] = { 0 , 2 }

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | pair | |
|  |  | 0 | 1 |
| step | 0 | { 0 , 3 } | { 1 , 2 } |
| 1 | { 0 , 1 } | { 2 , 3 } |
| 2 | { 0 , 2 } | { 1 , 3 } |

Die vollständige Matrix für vier Spieler:

Abbildung : RRTA-Matrix für vier Spieler

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

Umsetzung auf die Problemstellung

Die „Spieler“ werden durch die Berechnungselemente ersetzt. Anhand dieser wird die RRTA-Matrix gebildet. Die Berechnungspaare werden dann, wie im Kapitel **4.2 Auflösen der Schleifen** beschrieben, auf die Prozessorkerne aufgeteilt.  
Die RRTA-Schritte geben dabei die zeitliche Abfolge vor.



Abbildung : RRTA – Aufteilung auf Prozessorkerne

In einem Beispiel, mit zwei Prozessorkernen, würden acht Elemente laut **Abbildung 11: RRTA – Aufteilung auf Prozessorkerne** verteilt werden. Da es nicht für jeden Paarindex einen Prozessorkern gibt, werden sie gleichmäßig, auf die Kerne, aufgeteilt.  
Prozessorkern 0 muss zuerst alle zugeteilten Elementpaare vom   
ersten Schritt ( 0-7 und 2-5 ) berechnen, während Prozessorkern 1 ebenfalls den ersten Schritt abarbeitet ( 1-6 und 3-4 ).  
Nach diesem Schema wird durch alle RRTA-Schritte iteriert. In diesem Fall wäre die Berechnung abgeschlossen, wenn sämtliche Prozessorkerne den siebten Schritt (step 6) abgeschlossen hätten.

Merkmale

Eine Aufteilung durch den RRTA führt inhärent dazu, dass jeder Prozessorkern gleich viele Berechnungen ausführen muss. Abhängig davon, ob diese Berechnungen eine ähnliche Laufzeit haben, bedeutet dies auch eine gleichmäßige Aufteilung der Last.  
Wird mit Ressourcensperrung gearbeitet, kann durch den RRTA eine Reduktion der Kollisionen erreicht werden.

Zu Bedenken ist die Größe der Matrix, welche benötigt wird, um den Algorithmus abzuarbeiten - vor allem bei einer hohen Anzahl an Berechnungselementen. Dies kann, bei Systemen mit sehr kleinem Speicher, zu Problemen führen.

Divide and Conquer

Bei „Divide and Conquer“-Algorithmen (**DCA**) handelt es sich um spezialisierte Vorgehensweisen, große Aufgaben oder Berechnungen, in kleinere aufzuteilen [5], 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 [6].  
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 DCA günstig aufteilen, wodurch es bei einer unpassenden Lösung durchaus sein kann, dass kein Effizienzgewinn oder sogar ein Effizienzverlust die Folge ist.

Realisierung durch RRTA

Durch den RRTA, lässt sich die Problemstellung sinnvoll als DCA darstellen.  
Dafür werden die Berechnungselemente in Stapel aufgeteilt. Die Anzahl der Stapel, ist dabei die doppelte Anzahl der Prozessorkerne, um, in jedem Rechenschritt des RRTA, zwei Stapel an jeden Prozessorkern übergeben zu können.



Tabelle : Beispiel - Stapelpaar für einen Prozessorkern

Zwischen diesen Stapelpaaren muss jede Kombination an Berechnungselementen abgearbeitet werden.   
Außerdem muss, für einen Berechnungsdurchlauf einmalig, jede Elementkombination innerhalb eines Stapels berechnet werden.

|  |  |  |  |
| --- | --- | --- | --- |
| interne Paare: | | verknüpfte Paare: | |
| A, B |  | A, C |  |
| C, D |  | A, D |  |
|  |  | B, C |  |
|  |  | B, D |  |

Tabelle : Beispiel - Elementkombinationen zwischen den Stapeln

Als Berechnungsbasis für die RRTA-Matrix, wird die Anzahl der Stapel, also ebenfalls die doppelte Anzahl an Prozessorkernen verwendet.

Die Schritte des RRTA werden einzeln parallelisiert, jedoch werden die Prozessorkerne nach jedem Schritt synchronisiert. Da es innerhalb eines RRTA-Schrittes keine Überlappung von Berechnungselementen gibt, kann auf die Sperrung von Ressourcen gänzlich verzichtet werden.  
Das gegenseitige Warten der Prozessorkerne, nach jedem Schritt, eliminiert somit alle Zugriffskollisionen, allerdings mit dem Nachteil, dass immer die längste Laufzeit für die Gesamtlaufzeit ausschlaggebend ist. Es muss daher besonders darauf geachtet werden, die Last möglichst gleich auf die Prozessorkerne zu verteilen.



Abbildung : Divide and Conquer mittels RTTA

Stapelbildung

Jedes Berechnungselement muss in den Stapeln insgesamt genau einmal auftreten.   
Weiters, sollten die Berechnungselemente möglichst gleichmäßig auf die Stapel verteilt sein, um möglichst ähnliche Laufzeiten für die einzelnen RRTA-Schritte zu erhalten.

Valide Stapel, jedes Element kommt nur einmal vor:



Abbildung : Valide Stapel für zwei Prozessorkerne mit insgesamt 8 Elementen

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



Abbildung : Invalide Stapel für zwei Prozessorkerne mit insgesamt 8 Elementen

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

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

Abbildung : Valide Stapel für zwei Prozessorkerne mit insgesamt 7 Elementen

Merkmale

Die Erstellung eines DCA durch RRTA bringt erhebliche Vorteile, was die Skalierbarkeit bezüglich der Berechnungselemente betrifft.

Die RRTA-Matrix wird nur mehr anhand der Prozessorkernanzahl gebildet, welche ein konstanter und verhältnismäßig kleiner Wert ist. Außerdem muss die Matrix nicht mehr angepasst werden, wenn sich die Anzahl der Berechnungselemente ändert. Dies spart Laufzeit, wenn die Berechnungsanfragen von zyklischer Natur sind, da die Matrix wiederverwendet werden kann.

Mit steigender Anzahl von Berechnungselementen, steigt außerdem die potentielle Anzahl an Zugriffskollisionen, welche durch den DCA komplett eliminiert werden.   
Die Synchronisationszeit ist ausschließlich davon abhängig, wie gleichmäßig die Last auf die Prozessorkerne verteilt ist. Mit steigender Anzahl von Berechnungselementen und damit auch deren Berechnungen, können diese, statistisch gesehen, auch gleichmäßiger verteilt werden, was dazu führen kann, dass die Synchronisationszeit, mit zunehmender Anzahl von Berechnungselementen, nicht nur gleich bleibt, sondern sogar sinkt.

Die Dauer für die Stapelbildung steigt linear mit der Anzahl der Berechnungselemente. Für zyklische Prozesse, lässt sich dieser Vorgang jedoch, ähnlich wie bei der RRTA-Matrix, speichern und wiederverwenden. Dieses Datenrecycling ist aber von Änderungen der Berechnungselementanzahl abhängig und sollte deshalb von Fall zu Fall betrachtet werden.

# Implementierungskomponenten des Testsystems

Der folgende Abschnitt beschreibt die Hard- und Softwarekomponenten und Eigenschaften der Implementierung, welche zur Erstellung der Testergebnisse verwendet wurde.

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 Herangehensweise, und in weiterer Folge Programmiersprache, viele Vorzüge bietet, um die entwickelten Verteilungssysteme vielseitig anwendbar zu machen.

Betriebssystem

Getestet wird auf einem 64 Bit Windows 10 Betriebssystem. Dieses unterstützt nativ das verwendete .NET-Framework.   
Es handelt sich dabei um kein Echtzeitsystem, was die Varianz der Ergebnisse erhöht. Dies muss, zum Beispiel durch Mittelwertbildung über die Messergebnisse, kompensiert werden.

Parallelisierung unter Windows

In diesem Abschnitt werden verschiedene Methoden, zur Parallelisierung von Programmen unter Windows, beschrieben. 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 Parallelisierungsmethoden abhängig zu machen.

### Threads

Für jede parallel durchgeführte Arbeitsanweisung werden neue Threads erzeugt, welche dann die auf die Prozessorkerne 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 [7].  
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 Threads und Actors [8].  
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 [9].

Relevante Hardware-Spezifikationen

**Motherboard**:

Hersteller DELL  
Produktnummer DELL 0GY6Y8   
Chipsatz Intel Q77 (Panther Point DO)

**Prozessor**:

Hersteller Intel®  
Produktname Core™ i5-3470 CPU  
Prozessorkerne 4  
Taktfrequenz 3,2 GHz

**Arbeitsspeicher**:

Hersteller SK Hynix  
Nummer HMT351U6CFR8C-PB  
Speichertyp DDR3 1600  
Speichergröße 8 GB (2 x 4 GB)  
Modus Dual-Channel Mode  
Speichergeschwindigkeit 800 MHz  
Timings 11-11-11-28

Testsetup

Überblick

### Initialisierung

Die Testsuite wählt zuerst eine Kombination aus Verteilungssystem und Berechnungssystem, welche getestet werden soll.  
Die Konfiguration des Berechnungssystem beinhaltet zum Beispiel die Anzahl der verwendeten Prozessorkerne, Verzögerungszeiten oder die Art der Berechnung, welche auf den Input angewendet wird.



Abbildung : Testsuite

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.

### 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 Prozessorkern ausgeführt wurde.

Grundstruktur

Alle Tests erben von dieser Grundstruktur.



Jeder Test wird 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 ein Array mit allen Elementen, mit denen es bereits berechnet wurde.  
Die Berechnung auf den Prozessorkerne, beinhaltet lediglich das Beschreiben dieses Arrays.  
Nach dem Abschluss der Berechnungen, werden sämtliche Arrays auf Fehler geprüft.

### Validierungs-Input

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

#### Struktur



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

Initialer Zustand von m\_calculatedWith bei insgesamt vier Elementen:

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

Tabelle : Validierungsarray - initialer Zustand

Valid() liefert „false“ zurück

Über die Funktion „SetCalculatedWithElement“ wird in das Array 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 ); 🡪

Tabelle : Validierungsarray - Einzelvalidierung

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

Tabelle : Validierungsarray - valider Zustand

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

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

Tabelle : Validierungsarray - Doppelberechnung

Valid() liefert „false“ zurück

#### 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 25.

#### 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 Arrays in den Validierungs-Dummies zu beschreiben.

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

#### 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 **Error! Bookmark not defined.** 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 25 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, eine minimale Laufzeit für die Berechnungen vorzugeben.

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 9.4.4. 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 Prozessorkerne 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 Prozessorkerne. 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 | | | |
|
|  |  |  |  |
| Verteilungssystem | | | |
|
|  |  |  |  |
| Berechnungssystem | | | |
|
|  |  |  |  |
| Prozessorkerne | | | |
|

Abbildung : Systemabstraktion

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.



Abbildung : Austauschbares Berechnungssystem

Prozessorkern-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 Prozessorkern-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

Tabelle : Berechnungsanweisung mit Stacks

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 Prozessorkern-Index an die Funktion „Synchronize“ übergeben wird, muss der Aufrufer warten, bis der jeweilige Prozessorkern seine Berechnungen beendet hat.  
Ohne Argument muss gewartet werden, bis sämtliche Berechnungen auf allen Prozessorkernen 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 Prozessorkerne 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 Prozessorkerne. 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);

}

Prozessorkern-Pool-Verteilung

Wenn eine Verteilung den flexiblen Austausch des unterliegenden Systems erlaubt, so erbt sie von dieser abstrakten Basisklasse. Grundstruktur zu finden auf Seite 37.

### 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(SharedMemoryCoree<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 Prozessorkern-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 37.

### 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 **Error! Bookmark not defined.**) 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 Prozessorkern-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();

}

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 22) 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:

|  |  |
| --- | --- |
| Elementanzahl | Kommentar |
| 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 |

Ausgabe der Validierungsergebnisse im Anhang 1 zu finden.

### 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 Prozessorkern-Pool-Schnittstelle, was die Kompatibilität mit den unterliegenden Systemen zerstört. Der Prozessorkern-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.

Die Anzahl der Einzelberechnungen hängt von der Anzahl der Berechnungselemente ab. Die zugehörige Formel kann auf Seite 6 nachgelesen werden.

#### Overhead

Testbeschreibung zu finden auf Seite 32.

Je nach Compiler-Optimierung, hat eine Einzelberechnung, keine oder eine nur sehr geringe Laufzeit.

Messdurchläufe:

|  |  |
| --- | --- |
| **Overhead 1** | |
| Anzahl Elemente | 16 |
| Anzahl Berechnungen | 120 |

|  |  |
| --- | --- |
| **Overhead 2** | |
| Anzahl Elemente | 128 |
| Anzahl Berechnungen | 8.128 |

|  |  |
| --- | --- |
| **Overhead 3** | |
| Anzahl Elemente | 1024 |
| Anzahl Berechnungen | 499.500 |

#### Fixierte Rechenzeit

Testbeschreibung zu finden auf Seite 32.

Die Laufzeit einer Einzelberechnung ist auf 2ms fixiert.

Messdurchläufe:

|  |  |
| --- | --- |
| **Fixierte Rechenzeit 1** | |
| Anzahl Elemente | 16 |
| Anzahl Berechnungen | 120 |

|  |  |
| --- | --- |
| **Fixierte Rechenzeit 2** | |
| Anzahl Elemente | 128 |
| Anzahl Berechnungen | 8.128 |

#### Zufällige Rechenzeit

Testbeschreibung zu finden auf Seite 33.

Die durchschnittliche Laufzeit einer Einzelberechnung ist 2ms.   
Die mögliche Abweichung nach oben und unten beträgt 1ms.

Messdurchläufe:

|  |  |
| --- | --- |
| **Zufällige Rechenzeit 1** | |
| Anzahl Elemente | 16 |
| Anzahl Berechnungen | 120 |

|  |  |
| --- | --- |
| **Zufällige Rechenzeit 2** | |
| Anzahl Elemente | 128 |
| Anzahl Berechnungen | 8.128 |

#### Auslastung

Testbeschreibung zu finden auf Seite 33.

Jede Einzelberechnung beinhaltet 1024 Schleifendurchläufe.  
Multipliziert man die Schleifendurchläufe mit der Anzahl der Einzelberechnungen, so erhält man die Anzahl der Summen, welche gebildet werden müssen.

Messdurchläufe:

|  |  |
| --- | --- |
| **Auslastung 1** | |
| Anzahl Elemente | 16 |
| Anzahl Berechnungen | 120 |
| Anzahl Summenberechnungen | 122.880 |

|  |  |
| --- | --- |
| **Auslastung 2** | |
| Anzahl Elemente | 128 |
| Anzahl Berechnungen | 8128 |
| Anzahl Summenberechnungen | 8.323.072 |

|  |  |
| --- | --- |
| **Auslastung 1** | |
| Anzahl Elemente | 1024 |
| Anzahl Berechnungen | 523.776 |
| Anzahl Summenberechnungen | 536.346.624 |

Single-Thread Referenz Algorithmus

Dieser Algorithmus wurde auf Seite **Error! Bookmark not defined.** beschrieben.

### Implementierung



In der auf Seite 39 beschriebenen Basisklasse wird lediglich die „Calculate“-Funktion überschrieben.  
Die Berechnungen werden sequentiell abgearbeitet und benötigen dafür nur einen Prozessorkern.  
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



Tabelle : Messergebnisse - Referenz-Verteilung

#### 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 Prozessorkerne ausgelastet wird.

#### Optimierung

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

Parallelisierung durch Locked Resource

Um mehrere Prozessorkerne verwenden zu können, werden Semaphore verwendet, um einzelne Elemente vor Mehrfachzugriff zu schützen. Dies wurde auf Seite **Error! Bookmark not defined.Error! Bookmark not defined.** beschrieben.

### Implementierung

Für einen ersten Versuch wird dabei die innere Schleife auf die einzelnen Prozessorkerne 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 Prozessorkern berechnet alle Kombinationen mit dem ersten Element, der zweite Prozessorkern übernimmt alle Kombinationen mit dem zweiten Element, usw.

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

CoreIndex = i % Anzahl der Prozessorkerne

berechnet.

Pseudocode – roter Teil wird auf die Prozessorkerne 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



Tabelle : Messergebnisse - Ressourcensperrende Verteilung

#### 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 Prozessorkernen 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 Prozessorkernen nicht ausgeglichen werden.

#### Optimierung

Die Verteilung belastet die Prozessorkerne 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 Prozessorkerne:

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:

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

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

Gleichmäßige Verteilung der Berechnungen

Als Basis wird die „Locked Resource“-Verteilung auf Seite 50 verwendet.  
Sie wird so überschrieben, dass eine simple aber gleichmäßige Aufteilung der Last auf alle Prozessorkerne erfolgt.

### Implementierung

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

Abhängig von diesem Zähler, wird die Berechnung auf die Prozessorkerne verteilt.  
Dies sorgt dafür, dass jeder Prozessorkern 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



Tabelle : Messergebnisse - Gleichmäßig Sperrende Verteilung

#### 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 nächste logische 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 50, mit Semaphoren, um die Elemente vor zeitgleichem Zugriff zu schützen. Allerdings wird die auf Seite 14 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 Prozessorkerne 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



Tabelle : Messergebnisse - Sperrende RRT-Verteilung

#### 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 Prozessorkern, zwei Stapel (Stacks) von Elementen ohne Unterbrechung abgearbeitet. Der Algorithmus wird auf Seite 18 theoretisch beschrieben.

### Grundstruktur



### Stapelbildung

In diesem Abschnitt, wird die Implementierung der Stapel behandelt. Die Theorie und Eigenschaften zu den Stapeln, werden auf Seite 19 besprochen.

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 Prozessorkerne verteilt.

Beispiel für den ersten Schritt des RRTA mit drei Prozessorkernen:

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

Tabelle : Stapelverteilung mit der RRT-Matrix

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

### Implementierung

Die überschriebene „Calculate“-Funktion initialisiert die notwendigen Variablen, teilt die Elemente auf möglichst gleich große Stapel auf (Seite **Error! Bookmark not defined.**) und generiert die zugehörige RRTA-Matrix (Seite 14).  
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 Prozessorkerne und dem abschließenden Synchronisieren. Die ProzessorkernPool-Klasse wird auf Seite 35 beschrieben.

private void CalculateStep(int step)

{

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

{

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

}

CorePool.Synchronize();

}

Die Anweisungen (Seite 35) mit den Stapelpaaren für die Prozessorkerne 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



Tabelle : Messergebnisse - Synchronisierte RRT-Verteilung

#### 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 und der Stapelbildung 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 Prozessorkerne führt dies zu einer mindestens 3-fach so schnellen Ausführung, im Vergleich zu den anderen Algorithmen.

Abschließend lässt sich sagen, dass generell eine gleichmäßige Auslastung mit verhältnismäßig geringem Overhead erreicht wird. 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 Prozessorkerne, 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 Prozessorkerne linear an.

Ergebnis

Fazit

Nach Betrachtung der Messergebnisse, lässt sich feststellen, dass eine gleichmäßige Aufteilung der Berechnung, kein Garant für eine schnelle Ausführung ist.  
Es gilt vor allem, Zugriffskollisionen so gut wie möglich zu verhindern, oder, wie im Fall von DCA, komplett zu eliminieren.

Im allgemeinen Vergleich kann sich deshalb die Synchronisierte RRT-Verteilung, vor allem beim Praxisnahen Auslastungstest, als der beste Algorithmus zur Parallelisierung durchsetzen.  
Zusätzlich lässt sich lediglich bei dieser Verteilung ein Deadlock-Szenario ausschließen, was vor allem bei kritischen Anwendungen, von denen unter Umständen Menschenleben abhängen, alle anderen Verteilungen eliminiert.

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 Prozessorkerne lange blockiert werden, während bei einer sperrenden Lösung, die übrigen Prozessorkerne meistens nicht blockiert werden.  
Es sei angemerkt, dass sich, mit steigender Anzahl an Berechnungselementen, auch eine gleichmäßigere, statistische Verteilung der Berechnungslaufzeiten einstellt, was dieses Argument hauptsächlich für überschaubare Mengen an Elementen geltend macht.

Als Überblick lässt sich festhalten, dass sich die Synchronisierte RRT-Verteilung als zuverlässiger Standard einsetzen lässt, welcher selten, und nur unter sehr spezifischen Bedingungen, durch eine andere Verteilung ersetzt werden kann.

Ausblick und weiterführende Forschung

### Testumfang

Die Messungen wurden aus Hardwaremangel nur auf einem Computersystem durchgeführt. Dies lässt zum Beispiel die Frage offen, wie skalierbar die Verteilungen in der Realität sind. In der Theorie mag dies durchaus beantwortet sein, jedoch sollte eine solche Theorie immer von empirischen Tests untermauert werden, welche als Anstoß für weitere Arbeiten dienen könnten.

### Deadlock-Analyse

Die Möglichkeit des Verklemmens wurde in den Tests ignoriert, da auch bei mehreren 100.000 Durchläufen nie ein Deadlock aufgetreten ist.  
Nur bei der Synchronisierten RRT Verteilung kann ein Deadlock sicher ausgeschlossen werden.   
Bei sämtlichen sperrenden Algorithmen ist eine Deadlock-Situation nicht nur möglich, sondern sogar wahrscheinlich.

Ein mathematischer Beweis (oder Gegenbeweis) ist, aufgrund der unendlichen Anzahl an Kombinationen von Elementen und Prozessorkern-Zahlen, wahrscheinlich ebenso umfangreich oder sogar umfangreicher als die gesamte Arbeit und konnte deshalb nicht geliefert werden.

### Optimierung

Das Potential für zusätzliche Effizienzsteigerung dürfte vor allem im Bereich des Datencaching liegen. In der Optimierung könnten kleine Änderungen im Detail, hohe Auswirkungen auf die Gesamtlaufzeit haben, was eine Forschung in diese Richtung sehr attraktiv und lukrativ in Bezug auf Laufzeitenminimierung machen könnte.

### Auslagerung auf externe Prozessoren

Die Kompatibilität mit externen Prozessoren, zum Beispiel Grafikkarte oder vernetzte Rechner, wurde nicht untersucht. Hier würden andere Verteilungen, unter Umständen, besser abschneiden, beziehungsweise müssten die Verteilungen an solche Strukturen angepasst werden.

# Quellenverzeichnis

|  |  |
| --- | --- |
| [1] | J. Clark, 25 12 2010. [Online]. Available: http://www.zdnet.com/article/intel-why-a-1000-core-chip-is-feasible/. [Zugriff am 27 9 2017]. |
| [2] | A. Silberschatz, P. B. Galvin und G. Gagne, Operating System Concepts, Wiley, 2013. |
| [3] | D. Wellein und Gotz, Fundamentals of Parallel Processing, Roorkee, 2009. |
| [4] | J. H. Dinitz, „Designing Schedules for Leagues and,“ 11 2013. [Online]. Available: http://www.emba.uvm.edu/~jdinitz/preprints/design\_tourney\_talk.pdf. [Zugriff am 27 9 2017]. |
| [5] | C. E. L. a. R. L. R. Thomas H. Cormen, Introduction to Algorithms, Cambridge, Massachusetts: The MIT Press, 2004. |
| [6] | M. R. Radu Rugina, „Recursion Unrolling for Divide and Conquer Programs,“ in *Languages and Compilers for Parallel Computing*, Cambridge, Massachusetts, Springer, 2001, pp. 34-48. |
| [7] | W. D. Clinger, „Foundation of Actor Semantics,“ 5 1981. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/6935/AITR-633.pdf?sequence=2. [Zugriff am 27 9 2017]. |
| [8] | P. Rezaei, 5 8 2007. [Online]. Available: https://blogs.msdn.microsoft.com/pedram/2007/08/05/dedicated-thread-or-a-threadpool-thread/. [Zugriff am 27 9 2017]. |
| [9] | D. Carmona, 6 2002. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms973903.aspx?f=255&MSPPError=-2147217396. [Zugriff am 27 9 2017]. |

Anlagen

Validierungsergebnisse ……………………………………………………………… A-I

Anlagen, Validierungsergebnisse





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.

Innsbruck, den 10.10.2017

David Hofer