Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Formelverzeichnis

Abkürzungsverzeichnis

RRT - Round Robin Tournament

RRTA - Round Robin Tournament Algorithmus

DCA - Divide and Conquer Algorithmus

1 Einleitung

1.1 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 skalierbare, möglichst generische Lösungen für ganze Klassen von Problemstellungen. Eine dieser Klassen, welche im Umfang dieser Arbeit *paarweise, ungeordnete Berechnungen* genannt wird, wird deshalb das Thema sein.

Paarweise, ungeordnete Berechnungen ergeben sich aus der abzählenden Kombinatorik. „Paarweise“ sagt hierbei aus, dass der Ausgang eines Berechnungsschrittes nur von zwei Elementen aus einer Datenmenge abhängt. Die Berechnung ist dann abgeschlossen, wenn alle Berechnungsschritte ausgeführt wurden, also alle möglichen Paare, innerhalb der Datenmenge, durchlaufen wurden.
"Ungeordnet" ist die Berechnung, weil ihr Ergebnis nicht von der Reihenfolge der Berechnungsschritte beeinflusst wird.

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.

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.

1.2 Zielsetzung

Es sollen mögliche Parallelisierungsalgorithmen, auf ihre Kompatibilität mit dieser Problemstellung, untersucht werden. Die Berechnungen, zwischen den Elementen, sollen dabei auf mehrere Prozessorkerne verteilt werden. Die tatsächliche Art der Berechnung, welche zwischen den Elementen ausgeführt wird, ist dabei nicht relevant. Es soll eine Lösung gefunden werden, welche diese Berechnungsschritte variabel lässt und dadurch auf alle Probleme, mit dieser Struktur, anwendbar ist. Daraus folgt auch, dass ein Berechnungselement nicht zeitgleich, von mehreren Prozessorkernen, bearbeitet werden darf, da nicht ausschließlich von Readonly-Operationen, zwischen den Elementen, ausgegangen werden kann.

Es wird von einem lokalen Prozessorsystem ausgegangen, welches auf einen gemeinsamen Speicher Zugriff hat.
Die Lösung soll unabhängig von einem Betriebssystem anwendbar sein. Dies bedeutet, dass die Implementierung zwar mit Betriebssystem erfolgen kann, jedoch nicht auf ein spezifisches angewiesen ist.

Die Parallelisierungsmethoden sollen, durch empirische Tests, in möglichst repräsentativen Szenarien, welche alle Randbereiche abdecken, geprüft und validiert werden. Außerdem sollen die Tests vergleichbare Ergebnisse liefern, anhand derer, die verschiedenen Parallelisierungsmethoden, in Bezug auf ihre Beschleunigungsqualitäten, bewertet werden können.

Eine Implementierung, der erarbeiteten Algorithmen, ist für das Testen notwendig, soll jedoch auch dabei helfen die Lösungen zu veranschaulichen. Das Ziel ist es jedoch nicht, dem Leser dieser Arbeit, eine vollständige Implementierung vorzugeben, sondern stattdessen Optionen aufzuzeigen, welche sich unter realen Bedingungen bewährt haben.

1.3 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 Parallelisierungs-Algorithmus, welcher die Berechnungen auf die Prozessorkerne aufteilt.

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.

1.4 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.

Synchronisierung
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. Dies inkludiert sämtliche Vorgänge, welche für die Parallelisierung und nicht zur eigentlichen Berechnung beisteuern.
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

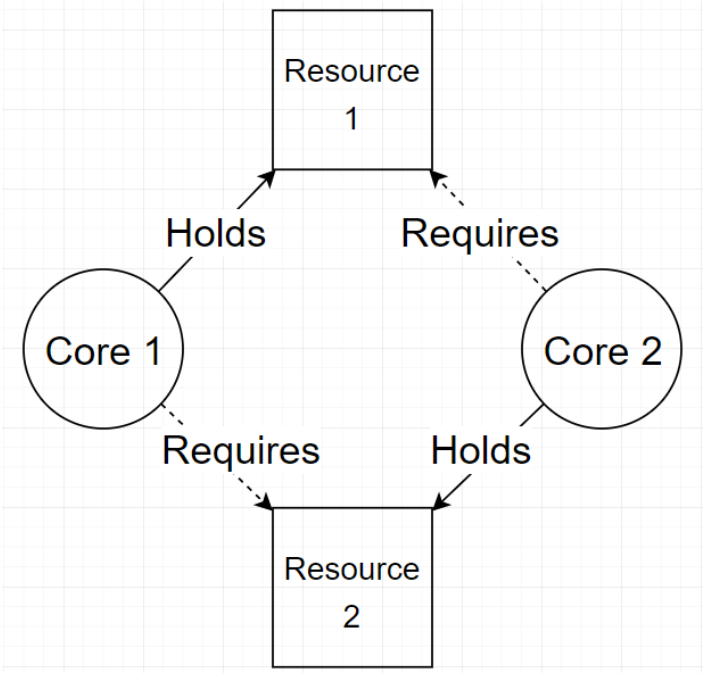


Abbildung 1: System im Deadlock

Skalierung

Mit steigender Anzahl von Prozessorkernen oder Berechnungselementen, kann es, je nach Art der Parallelisierung, negative Effekte geben, welche den Rechenvorgang verlangsamen. Diese Effekte würden beispielsweise dafür sorgen, dass, bei einer Verdopplung der Rechenkerne, keine Halbierung der Berechnungszeit stattfindet.

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

$$\binom{n}{k} = \frac{n!}{k! * (n-k)!}$$

Formel 1: Mögliche Kombinationen mit unbestimmten Teilmengen

Für die paarweise Variante ist k immer 2:

$$\binom{n}{2} = \frac{n!}{2 * (n-2)!}$$

Formel 2: Mögliche Kombinationen mit Paaren

Durch Umformung der Formel wird ersichtlich, dass die Anzahl der Kombination, quadratisch zu den Berechnungselementen, steigt:

$$\binom{n}{2} = \frac{n!}{2 * (n-2)!} = \frac{n * (n-1)}{2} = \frac{n^2 - n}{2}$$

Formel 3: Kombinationen in umgeformter Darstellung

Durch den quadratischen Anstieg, der Anzahl der Berechnungen, ist es besonders wichtig, dass Parallelisierungs-Algorithmen eine gute Skalierung, mit der Anzahl der Berechnungen, vorweisen. In der Praxis bedeutet das, dass eine Verdoppelung der Anzahl der Berechnungen bestenfalls nur eine Verdoppelung der Rechenlaufzeit nach sich tragen soll.

2 Hilfsstrukturen

In diesem Abschnitt werden Strukturen beschrieben, welche nicht unmittelbar zur Parallelisierung anwendbar sind. Die vorgestellten Konzepte dienen jedoch als Bausteine oder Startpunkte, um Parallelisierungsalgorithmen zu realisieren.

2.1 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++)
	{
		Calculation(elements[i], elements[j]);
	}
}
```

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.
Der Algorithmus wird außerdem als Grundstein, für andere Parallelisierungsmethoden, verwendet.

2.2 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].

2.2.1 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.

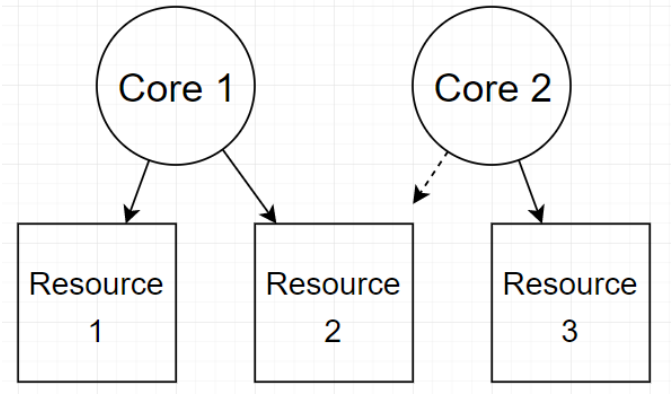


Abbildung 2: Ressourcensperrung

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.

2.2.2 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.
Für die Problemstellung gilt, dass, mit steigender Anzahl von Berechnungselementen, auch die theoretische Anzahl der möglichen Kollisionen, exponentiell zunimmt.

2.3 Round Robin Tournament Algorithmus

Der Round Robin Tournament Algorithmus (**RRTA**) wurde ursprünglich für das Zuweisen von Spielpartnern, bei zum Beispiel Schachturnieren, entwickelt [3]. 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, was eine kollisionsfreie Aufteilung der Berechnungen ermöglicht.

2.3.1 Vorgehensweise

Die Spieler werden durchnummeriert und die Indizes „kreisförmig“ in einer Tabelle mit zwei Reihen angeordnet (siehe Abbildung).
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.

0	1	2	3
7	6	5	4

Abbildung 3: RRTA - erste Runde

In der zweiten Runde werden alle Spieler, gegen den Uhrzeigersinn, um eins weitergerückt, mit der Ausnahme von einem Spieler (im Beispiel Spieler 0).
Nun spielt wieder die obere Reihe gegen die untere
(Spieler 0 gegen 1, Spieler 2 gegen 7, …).

0	2	3	4
1	7	6	5

Abbildung 4: RRTA - zweite Runde

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

0	7	1	2
6	5	4	3

Abbildung 5: 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 6: 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 4: RRTA Rundenanzahl - gerade Anzahl an Teilnehmer

N = ungerade Teilnehmeranzahl:
Benötigte Runden = N

Formel 5: RRTA Rundenanzahl - ungerade Anzahl an Teilnehmer

2.3.2 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.

2.3.2.1 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		
0	1	2
5	4	3

Speicheräquivalent als Integer-Array					
0	1	2	3	4	5

Abbildung 7: RRTA – Basisarray und Speicherabbild

2.3.2.2 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.

RRTA - 1. Schritt		
0	1	2
5	4	3

Speicheräquivalent als Integer-Array					
0	1	2	3	4	5

→Verschiebung →

RRTA - 2. Schritt		
0	2	3
1	5	4

Speicheräquivalent als Integer-Array					
0	2	3	4	5	1

Abbildung 8: RRTA - Schrittweise Verschiebung

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

2.3.2.3 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:

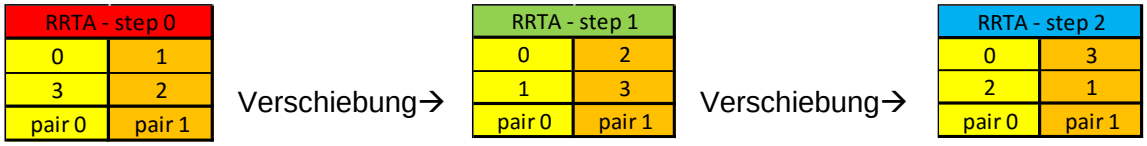


Abbildung 9: RRTA - Alle Schritte bei vier Spielern

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 }

Die vollständige Matrix für vier Spieler:

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

Abbildung 10: RRTA-Matrix für vier Spieler

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

3 Parallelisierungs-Algorithmen

3.1 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 [4].

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.

3.2 Auflösen der Schleifen

Bei dieser Herangehensweise, werden die Schleifen, in ihre Einzelberechnungen aufgeteilt, um eine gleichmäßige Verteilung auf die Prozessorkerne zu ermöglichen ([5] Kapitel 12.12 „Loop unrolling“).

Prozessorkern 1:

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

Prozessorkern 2:

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

Usw.

Die feinere Unterteilung bedeutet, im Vergleich zu der Aufteilung in innere und äußere Schleife, 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.

3.2.1 Parallelisierung durch Laufvariablenmodulo

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

$$Prozessorkernindex = Laufvariable \,\%\, Prozessorkernanzahl$$

Formel 6: Schleifenauflösung – Aufteilung mittels Laufvariablenmodulo

Für die doppelte Schleife der Problemstellung, muss für jeden Schleifendurchgang die Laufvariable einzigartig sein. Dafür kann ein Index nach jedem Berechnungsaufruf inkrementiert werden. In Kombination, kann jeder Schleifendurchlauf, einzeln, auf die Prozessorkerne verteilt werden.

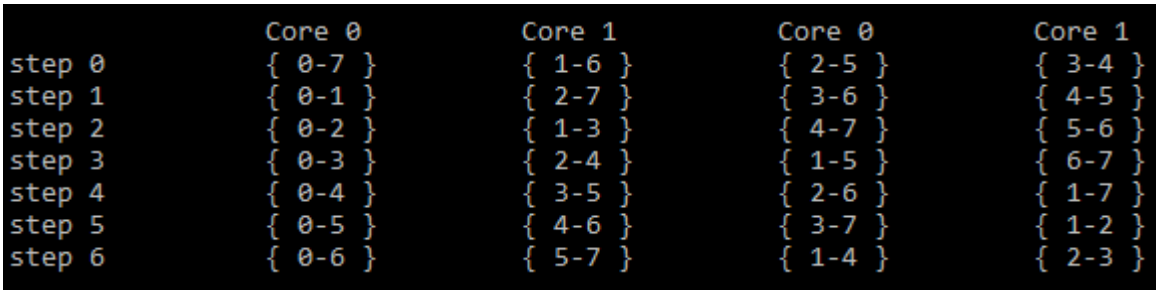
n… Anzahl der Elemente
coreCount... Anzahl der Prozessorkerne

```
int uniqueIndex = 0;
for (int i = 0; i < n - 1; i++)
{
	for (int j = i + 1; j < n; j++)
	{
		int coreIndex = uniqueIndex % coreCount;

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

		uniqueIndex++;
	}
}
```

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.

3.2.2 Parallelisierung durch Round Robin Tournament Algorithmus

Die „Spieler“ des RRTA werden durch die Berechnungselemente ersetzt. Anhand dieser wird die RRTA-Matrix gebildet. Die Berechnungspaare werden dann auf die Prozessorkerne aufgeteilt.
Die RRTA-Schritte geben dabei die zeitliche Abfolge vor.



	Core 0	Core 1	Core 0	Core 1
step 0	{ 0-7 }	{ 1-6 }	{ 2-5 }	{ 3-4 }
step 1	{ 0-1 }	{ 2-7 }	{ 3-6 }	{ 4-5 }
step 2	{ 0-2 }	{ 1-3 }	{ 4-7 }	{ 5-6 }
step 3	{ 0-3 }	{ 2-4 }	{ 1-5 }	{ 6-7 }
step 4	{ 0-4 }	{ 3-5 }	{ 2-6 }	{ 1-7 }
step 5	{ 0-5 }	{ 4-6 }	{ 3-7 }	{ 1-2 }
step 6	{ 0-6 }	{ 5-7 }	{ 1-4 }	{ 2-3 }

Abbildung 11: 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.

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.
Da mit Ressourcensperrung gearbeitet werden muss, 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.

3.3 Parallelisierung durch Divide and Conquer

Bei „Divide and Conquer“-Algorithmen (**DCA**) handelt es sich um spezialisierte Vorgehensweisen, große Aufgaben oder Berechnungen, in kleinere aufzuteilen [6], 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 [7].
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.

3.3.1 Realisierung durch RRTA

Durch den RRTA, lässt sich die Problemstellung sinnvoll als DCA darstellen.
Dafür werden die Berechnungselemente in Stapel aufgeteilt (nicht zu verwechseln mit Stapel/Stack im Speicher). 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.

Stack1	Stack2
A	C
B	D

Tabelle 1: 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 2: Beispiel - Elementkombinationen zwischen den Stapeln

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

Beispiel für die Stapelverteilung im ersten Schritt des RRTA mit drei Prozessorkernen:

RRTA - erster Schritt		
0	1	2
5	4	3

Abbildung 12: Beispiel - Stapelverteilung mit 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.

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 Prozessorkernlaufzeit, innerhalb eines RRTA-Schrittes, für die Gesamtlaufzeit des Schrittes ausschlaggebend ist. Es muss daher besonders darauf geachtet werden, die Last möglichst gleich auf die Prozessorkerne zu verteilen.

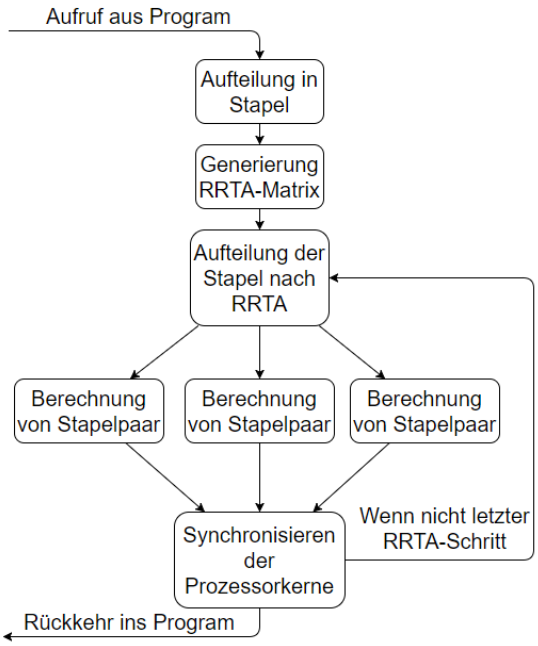


Abbildung 13: Divide and Conquer mittels RTTA

3.3.2 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:

Stack1	Stack2	Stack3	Stack4
elem1	elem2	elem3	elem4
elem5	elem6	elem7	elem8

Abbildung 14: Valide Stapel für zwei Prozessorkerne mit insgesamt 8 Elementen

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

Stack1	Stack2	Stack3	Stack4
elem1	elem2	elem2	elem4
elem5	elem6	elem7	elem8

Abbildung 15: 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 16: Valide Stapel für zwei Prozessorkerne mit insgesamt 7 Elementen

3.3.3 Merkmale

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

Die RRTA-Matrix wird nur noch 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.

4 Validierung der Parallelisierungs-Algorithmen

In diesem Kapitel werden Vorgehensweisen erläutert, um die erarbeiteten Algorithmen zu validieren. Dabei geht es nicht um eine Effizienz-Analyse, sondern um eine korrekte und komplette Ausführung aller Berechnungen.

4.1 Vergleich mit Ausgabe des sequentiellen Algorithmus

Der auf Seite 6 beschriebene sequentielle Algorithmus kann als Quelle, für sämtliche Berechnungspaare, verwendet werden. Dazu kann der Berechnungsaufruf umfunktioniert werden, um stattdessen eine Validierungsliste zu erstellen, welche alle Berechnungspaar-Indizes beinhaltet.

n… Anzahl der Elemente

```
for (int i = 0; i < n - 1; i++)
{
	for (int j = i + 1; j < n; j++)
	{
		WriteToValidationList(i, j);
	}
}
```

Der zu testende Parallelisierung-Algorithmus muss dann ebenfalls eine Vergleichsliste, mit allen Berechnungspaar-Indizes, bilden, welche im Laufe der Ausführung berechnet werden. Die Liste muss dabei vor zeitgleichem Zugriff, von mehreren Prozessorkernen, geschützt werden, da sonst das Ergebnis verfälscht werden könnte. Wie dies realisiert wird, hängt vom Algorithmus ab und ist Implementierungsdetail.

Abschließend wird die Vergleichsliste des Parallelisierungs-Algorithmus mit der Validierungsliste des sequentiellen Algorithmus verglichen. Dabei muss überprüft werden, ob jedes Berechnungspaar in der Validierungsliste, genau einmal, in der Vergleichsliste enthalten ist.
Sobald ein Berechnungspaar öfter als einmal oder nicht auf der Vergleichsliste zu finden ist, muss der Parallelisierungs-Algorithmus als fehlerhaft betrachtet werden.

4.2 Kombinationsliste innerhalb der Berechnungselemente

Bei dieser Art der Ergebnisvalidierung, wird die Prüfung innerhalb der Berechnungselemente vollzogen.
Die Problemstellung sieht vor, dass jedes Berechnungselement, mit jedem anderen Berechnungselement, kombiniert werden muss. Dies bedeutet im weiteren Sinn, dass man, bei der Betrachtung eines einzelnen Berechnungselementes, feststellen können muss, dass jedes andere Berechnungselement, für eine Berechnung, genau einmal herangezogen wurde.

Um dies prüfen zu können, kann, innerhalb der Berechnungselemente, eine Liste geführt werden. In diese Liste tragen sich alle Berechnungselemente ein, welche eine Berechnung durchführen.
Nachdem der Parallelisierungs-Algorithmus durchlaufen wurde, kann bei jedem Berechnungselement geprüft werden, ob es sämtliche Kombinationen, mit den anderen Elementen, gegeben hat.
Nur, wenn die Überprüfung bei sämtlichen Berechnungselementen positiv abgeschlossen werden kann, hat der Algorithmus richtig gearbeitet und kann als valide angesehen werden.

Diese Art der Validierung bietet mehrere Vorteile gegenüber dem Vergleich mit dem sequentiellen Algorithmus:

1.) Die internen Listen der Berechnungselemente, müssen nicht vor zeitgleichem Zugriff, durch mehrere Prozessorkerne geschützt werden, da die Problemstellung vorsieht, dass dies Aufgabe des Parallelisierungs-Algorithmus ist.
2.) Die Überprüfung beruht auf einem einzelnen Mechanismus, innerhalb der Berechnungselemente, und muss deshalb nicht, pro Parallelisierungs-Algorithmus, individuell implementiert werden.
3.) Es ist kein Referenz-Algorithmus notwendig, um die Validierung durchzuführen. Diese Unabhängigkeit schränkt die möglichen Fehlerquellen, bei der Implementierung, ein.

5 Implementierungskomponenten des Testsystems

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

5.1 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.

5.2 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.

5.3 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.

5.3.1 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.

5.3.2 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 [8].
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.

5.3.3 .NET Threadpool

Die .NET Implementierung des Threadpools vereint die Eigenschaften von Threads und Actors [9].
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 [10].

5.4 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:

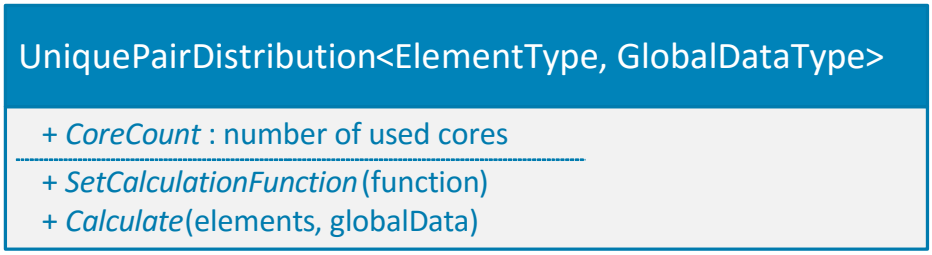
Hersteller	SK Hynix
Nummer	HMT351U6CFR8C-PB
Speichertyp	DDR3 1600
Speichergröße	8 GB (2 x 4 GB)
Modus	Dual-Channel Mode
Speichergeschwindigkeit	800 MHz
Timings	11-11-11-28

6 Implementierung der Verteilungsstrukturen

Dieser Abschnitt beschreibt die grundlegenden Verteilungsklassen, welche zur Implementierung für die Algorithmen verwendet werden.

6.1 Grundstruktur

Die abstrakte Basisklasse, von welcher sämtliche Verteilungsalgorithmen erben. Sie bildet die minimalen Anforderungen, an einen Parallelisierungsalgorithmus, ab.



“CoreCount” 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. Der Parameter wird mit dem Konstruktor übergeben und ist somit einstellbar.

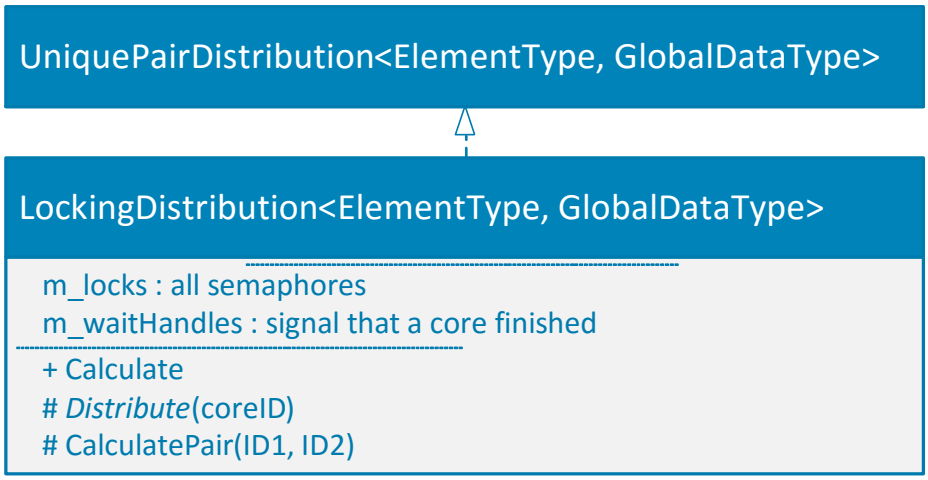
„SetCalculationFunction“ legt die Berechnungsfunktion fest, welche zwischen den Elementen angewandt wird.

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

```
public abstract class UniquePairDistribution<ElementType, GlobalDataType>
{
    public delegate void PairCalculationFunction(
        ElementType element1, ElementType element2, GlobalDataType globalData);

    int m_coreCount;
    public int CoreCount { get { return m_coreCount; } }

    PairCalculationFunction CalculationFunction { get; set; }

    public UniquePairDistribution(int coreCount)
    { m_coreCount = coreCount; }

    public void SetCalculationFunction(PairCalculationFunction function)
    { CalculationFunction = function; }

    public abstract void Calculate(
        ElementType[] elements, GlobalDataType globalData);
}
```

6.2 Verteilung mit Ressourcensperrung

Um Algorithmen mit gesperrten Ressourcen (Seite 7) 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.



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(ElementType[] 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.

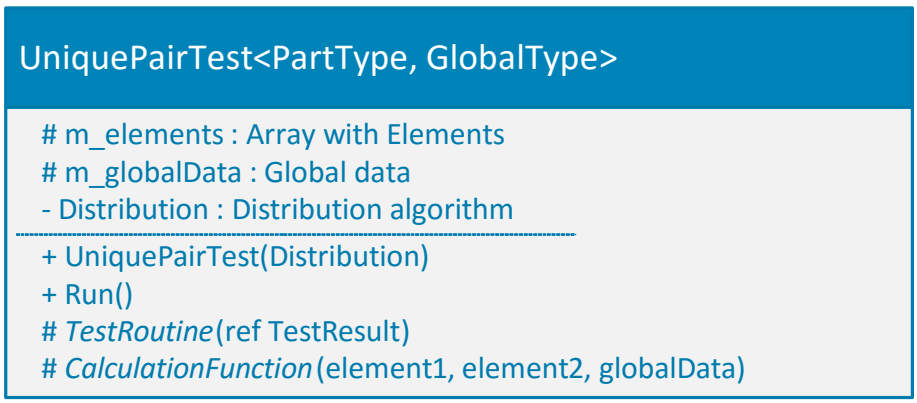
```
protected void CalculatePair(int id1, int id2)
{
    m_locks[id1].WaitOne();
    m_locks[id2].WaitOne();

    CalculationFunction(m_elements[id1], m_elements[id2], m_global);

    m_locks[id2].Release();
    m_locks[id1].Release();
}
```

7 Implementierung der Tests

Der Input ist abhängig von der Testroutine und kann entweder aus realen Daten bestehen, um Effizienz zu testen, oder aus Dummyobjekten bestehen, welche zusätzliche Auswertungen ermöglichen. Über den Output wird validiert, ob die Berechnungen korrekt durchgeführt wurde.

7.1 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 „Run“-Funktion initialisiert den Test und führt die Zeitmessungen durch.

```
public TestResult Run()
{
    TestResult result = new TestResult();
    Stopwatch stopWatch = new Stopwatch();

    stopWatch.Restart();
    TestRoutine(result);
    stopWatch.Stop();

    result.Duration = stopWatch.Elapsed;
    return result;
}
```

Die “TestResult”-Klasse stellt dabei das Ergebnis dar. In diesem Ergebnis wird eingetragen, wie lange die Durchführung des Tests gedauert hat und ob der Test fehlerfrei war.

```
public class TestResult
{
    public bool Successful { get; set; } = false;
    public TimeSpan Duration { get; set; } = new TimeSpan(0);
}
```

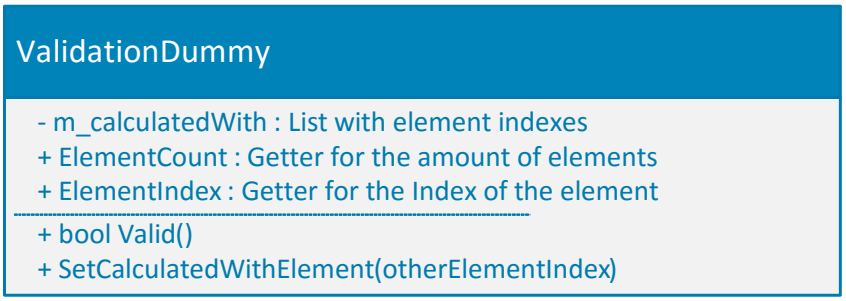
Die abstrakte Funktion „TestRoutine“ führt den eigentlichen Test durch und muss in den spezialisierten Testroutinen (folgende Kapitel) implementiert werden.

7.2 Output-Validierung

Der Output wird mithilfe von Kombinationslisten, innerhalb der Berechnungselemente, zu finden auf Seite 21, validiert. Dabei werden Dummyobjekte verwendet, welche ein Array mit allen Elementen, mit denen sie bereits berechnet wurden, mitführen.
Die Berechnung auf den Prozessorkerne, beinhaltet lediglich das Beschreiben dieser Arrays. Nach dem Abschluss der Berechnungen, werden sämtliche Arrays auf Fehler geprüft.

7.2.1 Validierungs-Input

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



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

Valid() liefert „false“ zurück

Tabelle 3: Validierungsarray - initialer Zustand

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

SetCalculatedWithElement(2); →

Index	Value
0	0
1	0
2	1
3	0

Tabelle 4: 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 5: 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

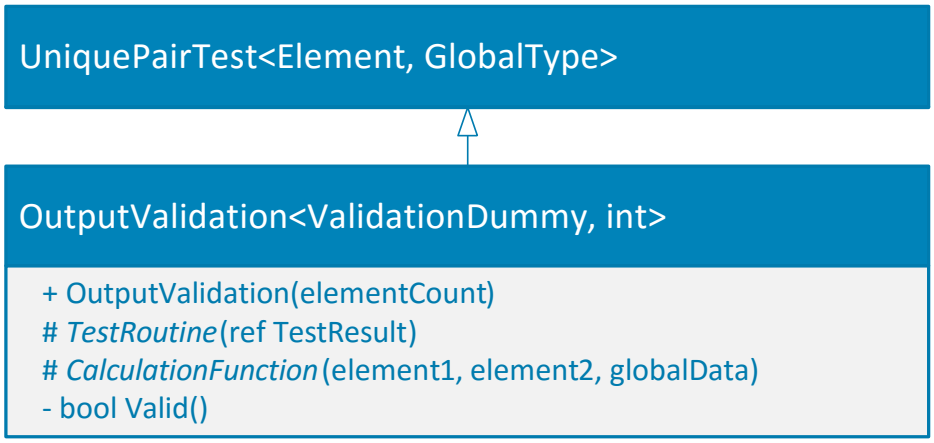
Valid() liefert „false“ zurück

Tabelle 6: Validierungsarray - Doppelberechnung

```
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;
    }
}
```

7.2.2 Test-Ablauf

Behandelt die Umsetzung des Validierungstests. Die Basisklasse wird auf Seite 28 beschrieben.



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.

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

Tabelle 7: Validierung - Getestete Zustände

Ausgabe der Validierungsergebnisse im Anhang 1 zu finden.

```
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;
    }
}
```

7.3 Effizienz-Messung

Die Effizienz-Messung wird ohne Output Validierung durchgeführt, da die Prüfung zusätzlichen Rechenaufwand verursacht, welcher die Zeitmessung verfälscht.

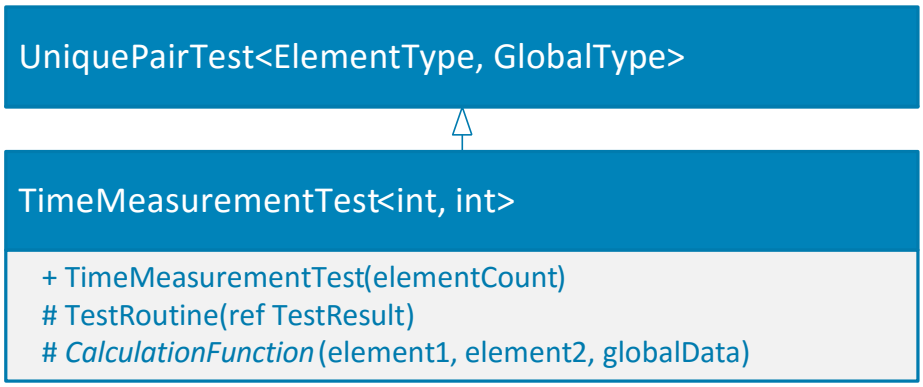
Für die Feststellung der Effizienz werden unterschiedliche Tests in Bezug auf Laufzeit oder Auslastung vorgenommen. Die Ergebnisse dürfen nur innerhalb des gleichen Tests verglichen werden.

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 10-mal durchgeführt, um stattdessen einen Mittelwert zu erhalten, welcher statistische Ausreißer leicht erkennbar macht und in diesen Messungen wesentlich aussagekräftiger ist.

Es werden verschiedene Testroutinen durchgeführt um möglichst viele Eigenschaften einer Verteilung messen zu können. Diese werden im späteren Verlauf des Kapitels beschrieben.

7.3.1 Struktur

Die Struktur ist bei allen Effizienz-Messungen gleich. Die übergeordnete Struktur ist auf Seite 28 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. Außerdem wird die Berechnungsfunktion zwischen den Elementen auf „CalculationFunction“ festgelegt. Diese wird in den erbenden Spezialisierungen überschrieben um verschiedene Messungen durchzuführen.

Die Funktion „TestRoutine“ startet die Berechnung mit der übergebenen Verteilung uns setzt das Ergebnis auf „true“, da es sich um eine reine Geschwindigkeitsmessung handelt.

```
abstract class TimeMeasurementTest : UniquePairTest<int, int>
{
    public TimeMeasurementTest(
        UniquePairDistribution<int, int> distribution,
        int elementCount) :
        base(distribution)
    {
        // initialize elements
        m_elements = new int[elementCount];
        for (int i = 0; i < elementCount; i++)
        {
            // set value to element index
            m_elements[i] = i;
        }

        // set calculation function from specialization
        Distribution.SetCalculationFunction(CalculationFunction);
    }

    protected override void TestRoutine(TestResult result)
    {
        // distribute calculations and execute them with the injected algorithm
        Distribution.Calculate(m_elements, m_globalData);

        result.Successful = true;
    }

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

7.3.2 Overhead-Messung

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

Der Berechnungsfunktion „CalculationFunction“, welche bei jedem Elementpaar 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
}
```

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

7.3.3 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(20);
}
```

Messdurchläufe:

Die Laufzeit einer Einzelberechnung ist auf 20ms fixiert.

Fixierte Rechenzeit 1	
Anzahl Elemente	16
Anzahl Berechnungen	120

Fixierte Rechenzeit 2	
Anzahl Elemente	128
Anzahl Berechnungen	8.128

7.3.4 Zeitmessung mit zufälliger Rechenzeit

Der Aufbau ist gleich wie bei der Zeitmessung mit fixierter Rechenzeit auf Seite 7.3.3. 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(10 + Random(20));
}
```

Messdurchläufe:

Die durchschnittliche Laufzeit einer Einzelberechnung ist 20ms.
Die mögliche Abweichung nach oben und unten beträgt 10ms.

Zufällige Rechenzeit 1	
Anzahl Elemente	16
Anzahl Berechnungen	120

Zufällige Rechenzeit 2	
Anzahl Elemente	128
Anzahl Berechnungen	8.128

7.3.5 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.

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

```
protected override void CalculationFunction(int element1, int element2, int global)
{
    int sum = 0;
    for (int i = 0; i < Difficulty; i++)
    {
         sum += element1 * element2;
    }
}
```

Messdurchläufe:

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.

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 3	
Anzahl Elemente	1024
Anzahl Berechnungen	523.776
Anzahl Summenberechnungen	536.346.624

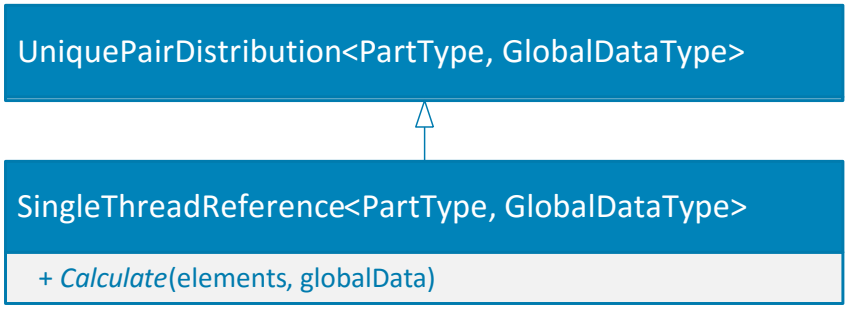
8 Algorithmus-Optimierung

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

Die Parallelisierung auf Betriebssystem-Ebene, wird einheitlich mittels Threadpool aus dem .NET-Framework (Seite 23) realisiert.

8.1 Single-Thread Referenz Algorithmus

Dieser Algorithmus wurde auf Seite 6 beschrieben.



In der auf Seite 25 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(ElementType[] 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);
        }
    }
}
```

8.1.1 Messergebnisse

Single Thread Reference										
	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.0002	0.0195	1.2089	2415.726	163603	2399.975	163335.1	0.0555	3.757	246.9038
Maximum[ms]	0.3235	0.0692	3.2902	2735.922	163962.1	2501.956	170648	0.2758	4.2625	275.5784
Durchschnitt[ms]	0.0326	0.025	1.5241	2451.09	163747.5	2440.344	165344.1	0.0795	3.9491	252.5167

Tabelle 8: Messergebnisse - Referenz-Verteilung

8.1.2 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.

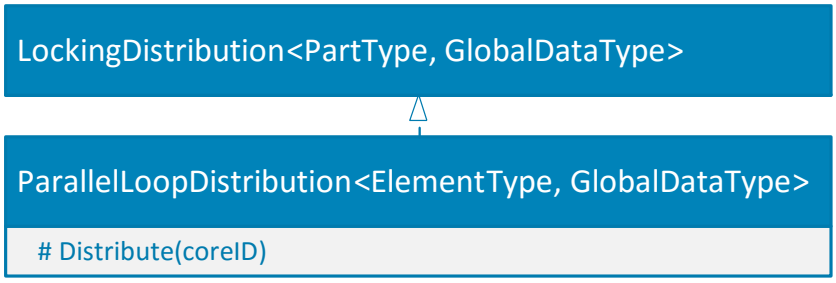
8.1.3 Optimierung

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.

8.2 Schleifen-Parallelisierung mit Ressourcensperrung

Der Algorithmus wurde auf Seite 12 theoretisch beschrieben.

Um mehrere Prozessorkerne verwenden zu können, werden Semaphore verwendet, um einzelne Elemente vor Mehrfachzugriff zu schützen. Dies wurde auf Seite 7 beschrieben.



Die innere Schleife wird auf die einzelnen Prozessorkerne aufgeteilt.

```
protected override void Distribute(int coreID)
{
    for (int i = coreID; i < m_elements.Length - 1; i += CoreCount)
    {
        for (int j = i + 1; j < m_elements.Length; j++)
        {
            CalculatePair(i, j);
        }
    }
}
```

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

8.2.1 Messergebnisse

Dual Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.1338	8.1653	512.7431	1317.138	83824.77	1291.681	82864.25	0.1744	10.1078	644.9474
Maximum[ms]	1.3317	8.964	555.0573	1418.048	83998.62	1447.414	84243.34	1.3174	11.4776	681.209
Durchschnitt[ms]	0.302	8.5377	523.7914	1377.769	83922.37	1382.013	83543.18	0.409	10.5714	656.6345
Vergleich[%]	10.79	0.29	0.29	177.90	195.12	176.58	197.91	19.44	37.36	38.46

Tri Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.1107	5.7857	348.7095	1241.687	70204.82	1223.122	69273.13	0.1566	7.0154	435.1918
Maximum[ms]	0.1797	6.7851	356.0263	1372.018	70766.42	1352.186	70434.93	1.1426	8.7229	443.2015
Durchschnitt[ms]	0.1343	6.0936	353.3013	1293.343	70507.29	1300.454	69936.47	0.2885	7.4246	438.9391
Vergleich[%]	24.27	0.41	0.43	189.52	232.24	187.65	236.42	27.56	53.19	57.53

Quad Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.1022	4.8675	312.3099	851.8468	42839.18	859.1767	42887.5	0.2197	5.6544	387.9573
Maximum[ms]	0.1882	8.0888	351.0655	952.1756	43392.33	995.4945	43550.63	0.4629	8.9944	424.3595
Durchschnitt[ms]	0.1335	5.5531	324.6621	915.7022	43183.29	907.9988	43230.86	0.2934	6.5319	399.6508
Vergleich[%]	24.42	0.45	0.47	267.67	379.19	268.76	382.47	27.10	60.46	63.18

Tabelle 9: Messergebnisse – Schleifen-Parallelisierung

8.2.2 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 meist um ein vielfaches schneller durchgeführt.

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

8.2.3 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:

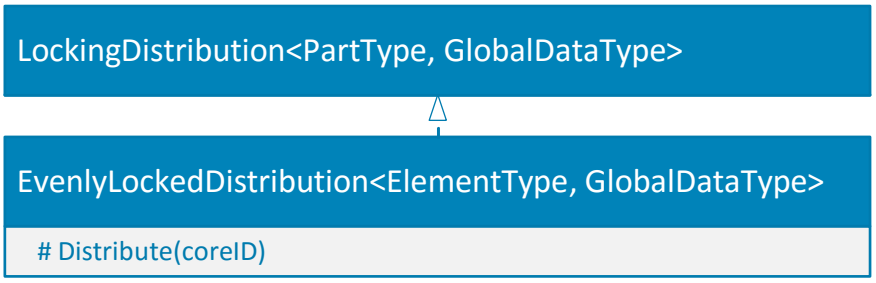
Prozessorkern 0: 36
Prozessorkern 1: 32
Prozessorkern 2: 28
Prozessorkern 3: 24

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

8.3 Parallelisierung durch Laufvariablenmodulo

Der Algorithmus wurde auf Seite 14 theoretisch beschrieben.

Durch die Implementierung soll eine simple aber gleichmäßige Aufteilung der Last, auf alle Prozessorkerne, erfolgen.



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.

```
protected override void Distribute(int coreID)
{
    int coreIndex = 0;

    for (int i = 0; i < m_elements.Length - 1; i++)
    {
        for (int j = i + 1; j < m_elements.Length; j++)
        {
            coreIndex++;   // increment counter

            // execute calulation on selected core
            if (coreIndex % CoreCount == coreID)
                CalculatePair(i, j);
        }
    }
}
```

8.3.1 Messergebnisse

Dual Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.1759	9.0666	546.6444	1724.006	115471.8	1738.023	118334.6	0.2717	12.7204	871.7266
Maximum[ms]	0.5418	18.3025	1312.986	1926.985	122034.7	1972.309	158260.5	1.5517	21.9429	1622.863
Durchschnitt[ms]	0.3087	14.2135	815.4361	1806.172	118414.6	1854.302	141014	0.5435	17.618	1272.614
Vergleich[%]	10.56	0.18	0.19	135.71	138.28	131.60	117.25	14.63	22.42	19.84

Tri Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.1724	8.3728	384.1544	1762.437	98606.58	1693.145	101138.9	0.1931	12.3177	498.4355
Maximum[ms]	0.4558	11.2005	761.6207	1994.034	107009.9	2023.866	108161.5	1.3501	14.5522	700.6197
Durchschnitt[ms]	0.2357	9.6125	526.1855	1861.503	103771.3	1819.25	105465.1	0.5177	13.2699	606.6349
Vergleich[%]	13.83	0.26	0.29	131.67	157.80	134.14	156.78	15.36	29.76	41.63

Quad Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.2042	6.1068	368.7909	1176.163	63586.12	1205.416	63384.68	0.1841	7.1487	376.183
Maximum[ms]	1.6581	9.3723	441.7456	1400.846	67221.87	1370.629	70984.4	0.5236	11.9723	534.4379
Durchschnitt[ms]	0.4553	7.4745	402.3684	1268.358	65270.41	1297.666	65797.16	0.2917	9.4686	443.3834
Vergleich[%]	7.16	0.33	0.38	193.25	250.88	188.06	251.29	27.25	41.71	56.95

Tabelle 10: Messergebnisse – Parallelisierung durch Laufvariablenmodulo

8.3.2 Diskussion

Die Verteilung schneidet, trotz gleichmäßiger Verteilung der Berechnungen, in sämtlichen Kategorien schlechter ab, als die Schleifen-Parallelisierung.

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

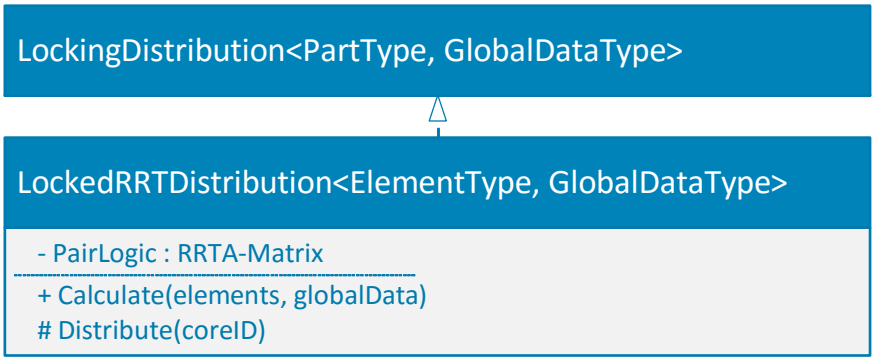
8.3.3 Optimierung

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

8.4 Round Robin Tournament Verteilung mit Ressourcen-sperrung

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

Wie bei der Laufvariablenmodulo-Parallelisierung, werden auch bei dieser Verteilung die Berechnungen einzeln verteilt. Allerdings wird die auf Seite 10 beschriebene RRT-Matrix verwendet, um die Berechnungspaare möglichst kollisionsfrei zu verteilen.



„PairLogic“ ist die Implementierung der RRTA-Matrix und beinhaltet, neben der Matrix selbst, die Parameter „StepCount“ und „PairCount“, welche die Höhe und die Breite der Matrix wiederspiegeln.
Die Matrix wird beim Hauptaufruf, mit der Anzahl der Berechnungselemente, initialisiert.

```
public override void Calculate(ElementType[] elements, GlobalDataType globalData)
{
    // generate matrix with the number of elements
    PairLogic.GenerateMatrix(elements.Length);

    // continue with base class call
    base.Calculate(elements, globalData);
}
```

Die tatsächliche Verteilung auf die Prozessorkerne wird dann, mittels der RRTA-Matrix, durchgeführt.

```
protected override void Distribute(int coreID)
{
    for (int step = 0; step < PairLogic.StepCount; step++)
    {
        for (int pair = coreID; pair < PairLogic.PairCount; pair += CoreCount)
        {
            // get element index from RRT matrix
            int id1 = PairLogic.PairMatrix[step][pair].ID1;
            int id2 = PairLogic.PairMatrix[step][pair].ID2;

            // execute calculation
            CalculatePair(id1, id2);
        }
    }
}
```

8.4.1 Messergebnisse

Dual Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.1344	7.9205	531.4916	1499.963	101905.5	1474.239	101428.2	0.1674	9.9436	630.209
Maximum[ms]	1.1578	8.2744	542.707	1580.502	102523.6	1543.341	102323.5	1.5453	10.8735	635.0563
Durchschnitt[ms]	0.2883	8.0915	535.4706	1522.588	102258.7	1511.787	101837	0.4109	10.2372	632.562
Vergleich[%]	11.31	0.31	0.28	160.98	160.13	161.42	162.36	19.35	38.58	39.92

Tri Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.1659	5.7193	348.8203	1183.252	70410.55	1162.985	56650.4	0.132	7.1104	434.9659
Maximum[ms]	1.1169	6.3488	360.0866	1244.798	70918.83	1251.767	70284.67	1.1128	8.303	444.4356
Durchschnitt[ms]	0.3914	5.8724	352.0078	1216.704	70673.73	1224.299	60550.96	0.2451	7.2927	438.9841
Vergleich[%]	8.33	0.43	0.43	201.45	231.70	199.33	273.07	32.44	54.15	57.52

Quad Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.1253	4.4209	281.4465	605.3337	41199	627.9038	41210.14	0.1665	5.3625	374.8016
Maximum[ms]	0.5555	6.2103	310.164	756.2625	48471.44	746.7116	48247.64	0.7013	10.0874	429.4079
Durchschnitt[ms]	0.2024	4.7931	293.27	663.2926	44157.03	690.36	44835.71	0.2952	6.3339	400.5516
Vergleich[%]	16.11	0.52	0.52	369.53	370.83	353.49	368.78	26.93	62.35	63.04

Tabelle 11: Messergebnisse - Sperrende RRT-Verteilung

8.4.2 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.

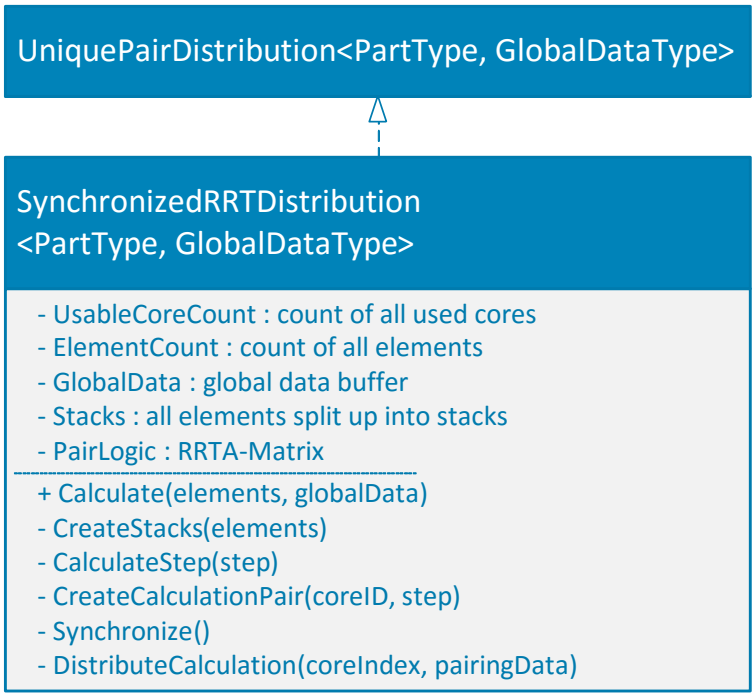
8.4.3 Optimierung

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.

8.5 Synchronisierte Round Robin Tournament Verteilung

Der Algorithmus wird auf Seite 16 theoretisch beschrieben.

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.



8.5.1 Synchronisierung

Der Aufruf von „Synchronize“ wartet, bis alle Prozessorkerne, ihre derzeitig zugewiesenen Berechnungen, abgeschlossen haben. Dies wird verwendet, um die Schritte des RRTA voneinander unabhängig zu machen, indem dafür gesorgt wird, dass kein Prozessorkern den nächsten Schritt anfangen kann, bevor nicht alle Prozessorkerne den derzeitigen RRTA-Schritt abgeschlossen haben.

8.5.2 Stapelbildung

In diesem Abschnitt, wird die Implementierung der Stapel behandelt. Die Theorie und Eigenschaften zu den Stapeln, werden auf Seite 18 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);
        }
    }
}
```

8.5.3 Berechnungsanweisung

Die Prozessor-Anweisungen, für das Berechnen von Stapelpaaren, werden mittels „PairingData“-Objekten an die Funktion „DistributeCalculation“ übergeben.

```
public class PairingData<ElementType, GlobalDataType>
{
    public ElementType[] Stack1 { get; set; }
    public ElementType[] 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

Tabelle 12: Berechnungsanweisung mit Stacks

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

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

8.5.4 Berechnungsaufruf

Die überschriebene „Calculate“-Funktion initialisiert die notwendigen Variablen, teilt die Elemente auf möglichst gleich große Stapel auf und generiert die zugehörige RRTA-Matrix (Seite 10).
Danach wird durch alle Schritte in der Matrix iteriert.
„PairLogic“ ist die Implementierung der RRTA-Matrix und beinhaltet, neben der Matrix selbst, die Parameter „StepCount“ und „PairCount“, welche die Höhe und die Breite der Matrix widerspiegeln.

```
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.

```
private void CalculateStep(int step)
{
    for (int i = 0; i < UsableCoreCount; i++)
    {
        DistributeCalculation(i, CreateCalculationPair(i, step));
    }

    Synchronize();
}
```

Die Anweisungen (Seite 50) 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);
}
```

8.5.5 Messergebnisse

Dual Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.0128	0.0254	0.6271	1466.719	101961.3	1481.895	101467.7	0.0476	2.0716	126.3747
Maximum[ms]	2.0512	0.1832	0.7004	1523.139	102660.9	1563.196	102228.4	0.2492	2.2248	132.0219
Durchschnitt[ms]	0.219	0.0448	0.6483	1497.297	102210.6	1530.233	101840.2	0.1311	2.1582	127.8332
Vergleich[%]	14.89	55.80	235.09	163.70	160.21	159.48	162.36	60.64	182.98	197.54

Tri Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.0298	0.0374	0.4053	1034.812	56164.15	1011.351	56142.82	0.0639	1.4509	90.7741
Maximum[ms]	0.3875	0.1019	0.6125	1041.325	57740.89	1085.702	56563.79	0.3281	3.076	107.7355
Durchschnitt[ms]	0.0659	0.0539	0.4822	1037.757	56339.8	1050.127	56356.85	0.1179	1.9851	97.2953
Vergleich[%]	49.47	46.38	316.07	236.19	290.64	232.39	293.39	67.43	198.94	259.54

Quad Core	Overhead			Fixiert		Random		Auslastung		
Elementzahl	16	128	1024	16	128	16	128	16	128	1024
Minimum[ms]	0.0298	0.0488	0.592	602.5765	41047.5	645.8992	41423.68	0.0751	1.0482	68.251
Maximum[ms]	0.1075	0.1294	0.7811	607.6243	46075.02	664.2423	41747.23	0.2291	1.9223	86.1284
Durchschnitt[ms]	0.0422	0.0636	0.6203	605.6623	42288.29	654.7052	41566.54	0.103	1.4583	74.4058
Vergleich[%]	77.25	39.31	245.70	404.70	387.22	372.74	397.78	77.18	270.80	339.38

Tabelle 13: Messergebnisse - Synchronisierte RRT-Verteilung

8.5.6 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 extrem schnellen Ausführung, im Vergleich zu den anderen Algorithmen. Nur bei 16 Elementen wirkt sich der Overhead noch zu stark aus, um die Durchführung der Berechnungen zu beschleunigen.

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.

9 Ergebnis

9.1 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 des 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.

9.2 Ausblick und weiterführende Forschung

9.2.1 Testumfang

Die Messungen wurden aus Hardwaremangel nur auf einem Computersystem durchgeführt. Dies lässt zum Beispiel die Frage offen, wie skalierbar die Verteilungen auf unterschiedlichen Systemen 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.

9.2.2 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 konnte eine Deadlock-Situation nicht ausgeschlossen werden.

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.

9.2.3 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.

9.2.4 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] 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].

[4] D. Wellein und Gotz, Fundamentals of Parallel Processing, Roorkee, 2009.

[5] F. Agner, „Agner Homepage," [Online]. Available: http://www.agner.org/optimize/optimizing_assembly.pdf. [Zugriff am 9 11 2017].

[6] C. E. L. a. R. L. R. Thomas H. Cormen, Introduction to Algorithms, Cambridge, Massachusetts: The MIT Press, 2004.

[7] 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.

[8] 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].

[9] 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].

[10] 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

Anlagen, Validierungsergebnisse

```
Starting unit test "Single Thread Reference validation with 4 elements".
Test duration: 0.9131ms
Test was successful.

Starting unit test "Locked Resource validation with 4 elements".
Test duration: 95.8952ms
Test was successful.

Starting unit test "Evenly Locked validation with 4 elements".
Test duration: 0.5263ms
Test was successful.

Starting unit test "Locked Round Robin Tournament validation with 4 elements".
Test duration: 1.6209ms
Test was successful.

Starting unit test "Synchronized Round Robin Tournament validation with 4 elements".
Test duration: 4.4423ms
Test was successful.

Starting unit test "Single Thread Reference validation with 16 elements".
Test duration: 0.0112ms
Test was successful.

Starting unit test "Locked Resource validation with 16 elements".
Test duration: 0.3665ms
Test was successful.

Starting unit test "Evenly Locked validation with 16 elements".
Test duration: 0.5218ms
Test was successful.

Starting unit test "Locked Round Robin Tournament validation with 16 elements".
Test duration: 0.5458ms
Test was successful.

Starting unit test "Synchronized Round Robin Tournament validation with 16 elements".
Test duration: 0.2492ms
Test was successful.

Starting unit test "Single Thread Reference validation with 1024 elements".
Test duration: 43.4398ms
Test was successful.

Starting unit test "Locked Resource validation with 1024 elements".
Test duration: 471.5162ms
Test was successful.

Starting unit test "Evenly Locked validation with 1024 elements".
Test duration: 602.1295ms
Test was successful.

Starting unit test "Locked Round Robin Tournament validation with 1024 elements".
Test duration: 631.2832ms
Test was successful.

Starting unit test "Synchronized Round Robin Tournament validation with 1024 elements".
Test duration: 33.0819ms
Test was successful.
```

```
Starting unit test "Single Thread Reference validation with 59 elements".
Test duration: 0.1513ms
Test was successful.

Starting unit test "Locked Resource validation with 59 elements".
Test duration: 1.4961ms
Test was successful.

Starting unit test "Evenly Locked validation with 59 elements".
Test duration: 2.4221ms
Test was successful.

Starting unit test "Locked Round Robin Tournament validation with 59 elements".
Test duration: 1.7938ms
Test was successful.

Starting unit test "Synchronized Round Robin Tournament validation with 59 elements".
Test duration: 0.2055ms
Test was successful.
```

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 22.11.2017

David Hofer