Speedupanalyse eines parallelen Grace Hash Join Algorithmus in Go

Christian Dienbauer

Institut für Informatik - Währingerstraße 29, 1090 Wien

Zusammenfassung. Moderne Multicore-CPU's erlauben mehrere Operationen parallel auszuführen. Gerade Datenbankoperationen wie der Grace Hash Join Algorithmus können davon sehr profitieren. Wie sehr ein Algorithmus von der Nutzung mehrerer Kerne profitiert, zeigt die Speedupanalyse. In dieser Arbeit wird der Join Algorithmus in Go realisiert und gezeigt, wie wichtig die richtige Anzahl an Goroutines ist. Der Algorithmus erreicht dabei einen nahezu linearen Speedup und eine parallelen Effizienz $E(P, N) \geq 1$. Dies ist auf das parallele Design zurückzuführen, wodurch der Algorithmus, erst durch das Verarbeiten mehrerer Operationen gleichzeitig, effizient wird.

Schlüsselwörter: Parallelität, Speedup, parallele Effizienz, Grace Hash Join, Golang

1 Einleitung

Aufgrund immer größer werdender Datenmengen, werden Algorithmen benötigt, welche das parallele Abarbeiten von Aufgaben ermöglichen. Grundsätzlich gibt es zwei Herangehensweisen für paralleles Ausführen von Datenbankoperationen: Inter- und Intra-Operator Parallelität [1]. Während Inter-Operatoren die Aufgaben auf verteilte Systeme aufteilt, legen Intra-Operatoren den Fokus auf das parallele Ausführen auf einer Maschine. Bei der Entwicklung von Multicore-CPU's hat sich in den letzten Jahren einiges getan. Heutige CPU's haben vier oder sogar mehrere Kerne und können pro Kern mehrere Threads gleichzeitig ausführen. Ein weiterer wichtiger Aspekt bei dieser Entwicklung, sind die wachsenden Kapazitäten in der gesamten Hierarchie von Cache-Speicher. Auch steigen, bei fallenden Kosten, die Kapazitäten der Arbeitsspeicher. Dies ermöglicht DBMS, ihre gesamten Daten, welche verarbeitet werden sollen, im Hauptspeicher zu halten. Dieser Trend wird sich in Zukunft vermutlich weiterhin fortsetzen, wobei es notwendig wird, wesentliche Datenbankoperationen an diese Gegebenheiten anzupassen. Der Join Operator ist einer der wichtigsten Operationen in einem relationalen DBMS. Er erlaubt es, verschieden Informationen in Relationen, über ein gemeinsames Attribut miteinander zu verknüpfen. Es gibt verschiedene Arten von Join-Operationen: Nested-Loop Join, Sort-Merge Join und Hash-Join. Wobei die letzten beiden die am meist genutzten in den heutigen DBMS sind [2]. In Zukunft wird das wesentliche an solchen Operationen sein,

dass sie das parallele Arbeiten auf Thread und Daten-Ebene unterstützen[1]. Parallelität gehört jedoch auch koordiniert wobei ein sogenannter Overhead entsteht. Wie groß dieser Overhead ist, gibt unter anderem Auskunft wie gut der Speedup eines Algorithmus ist.
In [1] wird auf einem Cell-Prozessor eine Analyse einer parallelen Grace Hace Join Implementation durchgeführt. Der Cell-Processor ist ein kostengünstiger weit verbreiterter Prozessortyp. Es wird gezeigt, dass sogar auf einfachster Hardware, erstaunliche Ergebnisse bei der parallelen Ausführung von Algorithmen realisiert werden können. [3] beschäftigt sich mit den Unterschieden zwischen Algorithmen, welche die zugrundeliegende Hardware berücksichtigen und welche die das nicht tun. Résumé dieser Arbeit ist es, dass jene, die auf Hardware abgestimmt sind, bessere Leistung bringen.
Google mit ihren riesigen Datenmengen ist immer wieder auf der Suche nach neuen Softwarelösungen. Keine der Programmiersprachen waren bis dato dafür entworfen worden, in so einem Ausmaß zu skalieren wie es heute bei Google notwendig ist. Nicht nur die dementsprechende Skalierbarkeit von Benutzer und Server zu realisieren, sondern auch um Skalierbarkeit über tausende von ProgrammiererInnen zu erzielen, welche an einer gemeinsamen Software entwickeln. Bei Go ergab sich für Google die Möglichkeit, Probleme die bei der Skalierbarkeit entstehen, in das Design einer Programmiersprache zu berücksichtigen[4].
Motivation dieser Arbeit ist es, den Speedup eines parallelen Grace Hash Join Algorithmus, in der Programmiersprache Go, zu analysieren.
In Kaptiel 2 werden zunächst Begriffe erwähnt, die häufig im Zusammenhang mit parallelen Algorithmen auftreten. Danach wird der Algorithmus schematisch erklärt und unter 4 ein kurzer Einblick in die Programmiersprache von Google gegeben. Kapitel 5 zeigt dann die Umsetzung des Algorithmus in Go. Tests und deren Ergebnisse werden unter 6 beschrieben.

2 Definitionen

2.1 Speedup

Sei $T(1, N)$ die Zeit die vom Algorithmus benötigt wird um ein Problem mit der Größe N mit nur einem Prozessor zu lösen und $T(P, N)$ die Zeit die er für das selbe Problem mit P Prozessoren benötigt, dann ist der Speedup des Algorithmus wie folgt definiert[5]:

$$S(P, N) = \frac{T(1, N)}{T(P, N)}. \tag{1}$$

- Normal: $S(P, N) \leq P$
- Ideal: $S(P, N) = P$
- Selten: $S(P, N) \geq P$

2.2 Parallele Effizienz

Parallele Effizienz ist definiert[5]:

$$E(P,N) = \frac{S(P,N)}{P} \tag{2}$$

- Normal: $E(P,N) \leq 1$
- Ideal: $E(P,N) = 1$
- Selten: $E(P,N) \geq 1$

Ein linearer Speedup entsteht wenn $E(P,N) = c$, wobei c unabhängig von N und P ist. Algorithmen die eine solche parallele Effizienz aufweisen, nennt man skalierbar.

2.3 Lastungleichgewicht

Wenn ein Prozessor i eine Zeit T_i dafür verwendet produktiv Arbeit zu verrichten, dann ist Gesamtzeit die von allen Prozessoren aufgewendet wird $\sum_{i=1}^{P} T_i$ und die durchschnittliche Zeit pro Prozessor[5]:

$$T_avg = \frac{\sum_{i=1}^{P} T_i}{P} \tag{3}$$

Die maximale Zeit die ein Prozessor Arbeit verrichtet $Tmax = maxTi$ wobei dann das Lastungleichgewicht wie folgt definiert ist:

$$I(P,N) = \frac{T_{max}}{T_{avg}} - 1 \tag{4}$$

2.4 Overhead

Bei der parallelen Ausführung von Programmen, beeinflussen der Kommunikations- und Lastenungleichgewichtsoverhead die parallele Effizienz wobei die Kosten eines Overhead auf die produktive Zeit eines Prozessors fallen. Der Overhead ist wie folgt definiert[5]:

$$H(P,N) = \frac{P}{S(P,N)} - 1 \tag{5}$$

2.5 Skalierbarkeit

Algorithmen welche ihre parallelen Effizienz $E(P,N)$ bei gleicher Problemgröße und steigender Anzahl an Prozessoren konstant halten, werden als stark skalierbar bezeichnet. Andere bei denen die Effizienz nur konstant bleibt wenn die Problemgröße zusammen mit der Anzahl an Prozessoren steigt, werden als schwach skalierbar bezeichnet[5].

3 Der Algorithmus

Es gibt verschiedene Hash Join Varianten. Der Speedup wird in dieser Arbeit anhand eines Grace Hash Join verglichen. Der Grace Hash Join Algorithmus arbeitet in drei Phasen. In der ersten Phase wird jedes der Tupel der inneren Relation R über das Join Attribut in N Partitionen gehasht. In der zweiten Phase geschieht das gleiche mit der selben Hashfunktion mit der äußeren Relation S. In der finalen Phase Vergleicht der Algorithmus dann jeweils die i_{th} Partitionen von R mit der i_{th} Partition von S und gibt das Ergebnis in einer neuen Relation aus. Die Anzahl der Partitionen N sollte möglichst groß gewählt werden. Damit soll verhindert werden, dass die Größen der einzelnen Partitionen der Relation R, die Kapazität der Cache-Speicher des Prozessors übersteigen. Die Partitionen der äußeren Relation müssen nicht zwingend zur Gänze in den Cache passen, da jedes Tupel nur einmal gelesen und anschließend das Ergebnis in eine Relation geschrieben wird[1]. Die einzelnen Partitionen der Relationen R und S sind somit voneinander unabhängig, was die Grundlage für eine Parallelität auf Daten-Ebene ist. Beim parallelen Grace Hash Join muss der Algorithmus nur mehr dafür sorgen, dass die Join-Operationen der einzelnen Partitionen in eigenen Threads ausgeführt werden, um die Aufgabe auf die einzelnen Kerne der CPU's aufzuteilen zu können.

```
Data: relations R and S
Result: joined tuples of R and S
for each tuple r in R relation do
    apply hash function to the join attribute of r;
    put r into the appropriate bucket R_i;
end
for each tuple s in S relation do
    apply hash function to the join attribute of s;
    put s into the appropriate bucket S_i;
end
for each bucket i do in PARALLEL do
    build the hash table from R_i;
    for each tuple s in S_i do
        apply hash function to the join attributes of s;
        use s to probe the hash table;
        output any matches to the result relation;
    end
end
```

Abb. 1. Schema eines parallalen Grace Hash Join Algorithmus[1]

4 GoLang

Go wurde von Robert Griesemer, Rob Pike und Ken Thompson als Mittarbeiter von Google ins Leben gerufen. Die erste stabile Version dieser Sprache wurde als Open-Source Projekt 2009 herausgegeben. Google hat mittlerweile 400 Mitarbeiter für das Projekt und eine beachtliche non-Google Community.

4.1 Grundidee

Go hat ein interessantes Objekt-Model. Es ist zwar eine objektorientierte Sprache, im Vergleich zu Sprachen wie zum Beispiel Java und C++ gibt es allerdings keine Klassen und auch keine Vererbung von Eigenschaften im selben Sinne. Das Objekt-Modell von Go handhabt dies mit Interfaces. Hier wird ein Interface Typ definiert welcher Spezifikationen von Methoden enthält. Es reicht nun einfach diese Methoden zu verwenden wenn sie gebraucht werden, ohne diese jemals irgendwo deklarieren zu müssen. Alleine dadurch, dass sie verwendet werden, werden sie implementiert. Das Endergebnis ist, dass Objekte weniger Verantwortungen besitzen und dazu tendieren, kleiner als Objekte in anderen Sprachen zu sein[4]. Somit sollen Systeme nicht durch das bilden von Typ-Hierarchien entstehen, sonder durch das Entwerfen von Bausteinen welche je nach Bedarf zusammengesetzt werden. Go ist eine Allzwecksprache mit Systemprogrammierung im Kopf. Es ist stark typisiert, besitzt eine automatische Speicherbereinigung und hat explizite Unterstützung für Parallelität. Seine Parallelitätsmechanismen machen es einfach, Programme zu schreiben, die das Meiste aus Multicore- und vernetzten Maschinen herausholen, während sein neuartiges System eine flexible und modulare Programmkonstruktion ermöglicht[6]. Programme werden aus Paketen aufgebaut, deren Eigenschaften eine effiziente Verwaltung von Abhängigkeiten ermöglichen. Die vorhandenen Implementierungen verwenden ein herkömmliches Compile / Link-Modell, um ausführbare Binärdateien zu erzeugen[7].

4.2 Nebenläufigkeit

Go realisiert diese Nebenläufigkeit mit zwei wesentlichen Elementen: Goroutines und Channels. Goroutines sind leichtgewichtige Threads, koordiniert von der Go runtime. Typischerweise laufen viele Goroutines in einigen wenigen Threads. Jeder Go Code läuft in Goroutines. Threads sind sehr teuer und Programme sollten nur soviele erzeugen, wie sie für das parallele Arbeiten benötigen. Im Vergleich dazu, können Goroutines beliebig oft erzeugt und vernichtet werden[4]. Um zwischen Goroutines kommunizieren zu können, werden Channels benutzt. Channels sind typisierte Leitungen, worüber Werte gesendet und empfangen werden. Diese erlauben es Goroutines, ohne explizitem Sperren von Variablen, sich zu synchronisieren[6].

5 Das Programm

Die verschiedenen Komponenten dieses Programms sind:

- Grace Hash Join
- Zwei Relationen R und S
- Eine Output-Relation
- Webserver
- Homepage

Webserver und Homepage dienen Schlussendlich nur als graphische Veranschaulichung der Werte von Tabellen und dem Endergebnis. Diese haben mit der Analyse nicht unmittelbar etwas zu tun und werden somit nicht weiter erwähnt.

5.1 Die Daten

Es gibt zwei Relationen. Die Anzahl der Tupel sowie der Speicherbedarf der Attribute können je nach Bedarf generiert werden. Die innere Relation R repräsentiert Lehrveranstaltungen und die äußere Relation S Studenten, welche eine oder mehrere Lehrveranstaltungen besuchen. Relation S wird zufällig aus den Einträgen der Relation R erzeugt. Beide Relationen bestehen jeweils aus zwei Attributen. Relation R besteht aus einer Kursnummer und einem Kursnamen. Relation S hingegen aus einer Matrikelnummer und der Kursnummer von ein Lehrveranstaltung, welche besucht wird. Dadurch, dass das Erzeugen von einigen Millionen Einträge in Relation S viel Zeit in Anspruch nimmt, werden beide Relationen als JSON Datei gespeichert. Dies hat außerdem den Vorteil, dass immer wieder mit den selben Daten getestet wird und somit Unregelmäßigkeiten, welche sich aus der zufälligen Erzeugung von Einträgen in Relation S ergeben, beseitigt werden.

5.2 Join

Wie bereits zuvor beschrieben, besteht der Grace Hash Join aus drei Phasen. In Abbildung 2 sieht man die ersten beiden Phasen des Algorithmus. Die Partitionierung der beiden Relationen R und S:

In Zeile 61 wird zunächst die Anzahl der Partitionen bestimmt. Diese wird mit der zweifachen Größe der Anzahl an Tupeln der inneren Relation R festgelegt. Hier sollte darauf geachtet werden, dass jede Partition der Relation R vollständig in die Cache Speicher der CPU passt. In den nächsten beiden Zeilen werden die Hashtabellen für die Tupel der beiden Relationen erzeugt. Der erste Index der Hashtabelle repräsentiert die einzelnen Partitionen. Der zweite Index zeigt dann auf das jeweilige Tupel. In den nächsten beiden for-Schleifen wird für jedes einzelne Tupel der beiden Relationen, ein Hashvalue über das Joinattribut erzeugt. Dies geschieht mit der Funktion “GenerateHashvalue()” welches auf das Paket “hash/fnv” zurückgreift. Steht die Partition fest, wird das Tupel an das Slice dieser Partition angehängt. Die Rückgabewerte dieser Funktion sind die

```
59 //Partitions courses and students table.
60 func partitioning(courses []mylib.Course, students []mylib.Student) ([][]mylib.Course, [][]mylib.Student) {
61     hashTableSize := len(courses) * 2 //Defines the quantity of partitions
62     courseHashTable := make([][]mylib.Course, hashTableSize)
63     studentHashTable := make([][]mylib.Student, hashTableSize)
64
65     //partitioning courses
66     for _, c := range courses {
67         hv := mylib.GenerateHashvalue(c.Id) % len(courseHashTable)
68         courseHashTable[hv] = append(courseHashTable[hv], c)
69     }
70
71     //partitioning students
72     for _, s := range students {
73         hv := mylib.GenerateHashvalue(s.CourseId) % len(studentHashTable)
74         studentHashTable[hv] = append(studentHashTable[hv], s)
75     }
76
77     return courseHashTable, studentHashTable
78 }
```

Abb. 2. Relation R und S werden partitioniert

```
79 //Splits the partitions to as many pieces as their are given goroutines.
80 func Join(courseHashTable [][]mylib.Course, studentHashTable [][]mylib.Student, cores int) {
81     hashTableSize := len(courseHashTable)
82     dataTables.JoinedT = make([]mylib.JoinElement, 0) //Output Table
83
84     runtime.GOMAXPROCS(cores) //Number of CPU's which the programm can use in parallal
85
86     //start a given number of goroutines
87     joinChannels := make([]<-chan mylib.JoinElement, maxGoroutines)
88     for i := 0; i < maxGoroutines; i++ {
89         joinChannels[i] = joinRangeofPartitions(courseHashTable[i*hashTableSize/maxGoroutines:(i+1)*hashTableSize/maxGorout
90     }
91
92     for n := range mylib.Merge(joinChannels) { //Merge alle channels
93         dataTables.JoinedT = append(dataTables.JoinedT, n) //Insert each output-tupel in output-table
94     }
95 }
```

Abb. 3. Relation R und S werden zusammengefügt

Partitionen der beiden Relationen R und S.
Nach der Partitionierung der beiden Relationen werden die einzelnen Partitionen miteinander verknüpft:

In der Funktion “Join()” (Abb. 3) wird zunächst eine leere Output-Relation erzeugt, in der die verknüpften Werte eingetragen werden. Danach wird die Anzahl an Goroutines festgelegt, welche die einzelnen Partitionen miteinander verbinden. Mit der Funktion “GOMAXPROCS()” in Zeile 84 wird die maximale Anzahl an Kernen, welche das Programm gleichzeitig ausführen, festgelegt. Dieser Funktion wird ein Integerwert, zwischen eins und der Anzahl von Kernen der CPU, übergeben[8]. Es können trotzdem mehr Threads als Kerne erzeugt werden, allerdings werden maximal nur so viele gleichzeitig ausgeführt, wie der Funktion übergeben wird. In der nächsten Zeile wird ein Slice erzeugt, wo die Länge gleich der Anzahl der Goroutines ist. Dieses Slice beinhaltet Channels vom Typ der Output-Relation, durch welche die einzelnen Goroutines ihre Output-Tupel schickten. In der folgenden for-Schleife werden dann die einzelnen Partitionen auf die Anzahl der zu verwendeten Goroutines aufgeteilt. Es wird dann für jedes Kollektiv der Partitionen die Funktion “joinRangeofPartitions()” aufgerufen, in welcher eine neue Goroutine gestartet wird. Diese Funktion liefert als Rückgabewert einen Channel welcher einem Index des Slices zugewiesen wird.

Abb. 4. Partitionen werden in einer neuen goroutine zusammengefügt

```
37 func joinRangeofPartitions(courseHashTable [][]mylib.Course, studentHashTable [][]mylib.Student) <-chan mylib.JoinElement {
38     out := make(chan mylib.JoinElement)
39
40     go func() {
41         for hi, _ := range studentHashTable {
42             for _, s := range studentHashTable[hi] { //For each student in partition
43                 for _, c := range courseHashTable[hi] { //compare with each course in same partition
44                     if s.CourseId == c.Id { //if the courseid is the same -> join them
45                         out <- mylib.JoinElement{StudentId: s.Id, CourseId: s.CourseId, CourseName: c.Name} //Create output-tupel
46                         break
47                     }
48                 }
49             }
50         }
51         close(out)
52     }()
53     return out
54 }
```

In Abbildung 4 ist zu sehen, wie zuerst ein Output-Channel erzeugt wird, durch den die Output-Tupel geschickt werden. Nun startet das Programm eine neue Goroutine. Hier wird nach dem selben Schema verfahren, wie in Abbildung 1 dargestellt. Nachdem die passenden Tupel gefunden wurden, wird ein Output-Tupel erstellt und durch den Channel gesendet. Nachdem alle Tupel abgearbeitet wurden, wird der Channel geschlossen. Dies symbolisiert der Funktion "Merge()" (Abb.5), wann eine Goroutine mit ihrer Arbeit fertig ist. Erst wenn alle Goroutines, welche die Partitionen zusammenfügen, ihre Channels geschlossen haben, schließt die Funktion "Merge()" ihren Channel und das bilden der Output-Relation kann abgeschlossen werden (Siehe Abb. 3 Zeile 92).

```
83 func Merge(cs []<-chan JoinElement) <-chan JoinElement {
84     var wg sync.WaitGroup //Waits for a collection of goroutines to finish
85     out := make(chan JoinElement)
86
87     // Start an output goroutine for each input channel in cs.  output
88     // copies values from c to out until c is closed, then calls wg.Done.
89     output := func(c <-chan JoinElement) {
90         for n := range c { //Waits till the goroutine closes the channel
91             out <- n
92         }
93         wg.Done()
94     }
95     wg.Add(len(cs)) //Add quantity of goroutines to wait for
96     for _, c := range cs {
97         go output(c)
98     }
99
100    // Start a goroutine to close out once all the output goroutines are
101    // done.  This must start after the wg.Add call.
102    go func() {
103        wg.Wait() //Waits till all goroutines are finished
104        close(out)
105    }()
106    return out
107 }
```

Abb. 5. Werte mehrerer Channels werden zusammengefügt

Zu Beginn der Funktion "Merge()" (Abb. 5) wird eine Variable vom Typ "WaitGroup" definiert. Dieser Typ stellt einen einfachen Mechanismus zur Synchronisation von Goroutines zur Verfügung. WaitGroup wartet auf eine Sammlung von Goroutines bis diese ihre Channes geschlossen haben. In Zeile 89 wird

eine weitere Funktion definiert, welche die Werte aus den Input-Channels in den Output-Channel übergibt und anschließend der WaitGroup mitteilt, dass alle Werte aus einem Input-Channel übertragen wurden. Nun wird der WaitGroup mitgeteilt auf wie viele Goroutines gewartet wird und für jede Goroutine die Funktion aufgerufen, um ihre Werte in den Output-Channel zu übertragen. Zum Schluss wird in einer weiteren Goroutine überprüft, ob alle Werte übertragen wurden. Das ist dafür notwendig, damit das Programm an dieser Stelle nicht hängen bleibt und auf das Fertigwerden der Goroutines zu wartet.

Zu guter Letzt, werden alle verbundenen Tupel aus dem Channel der Funktion “Merge()” in die Output-Relation geschrieben.

5.3 Webserver

Zum Erkennen von etwaigen Fehlern, wurde ein GUI mittels eines Webserver realisiert. Hier wurde eine Homepage eingebunden, welche unter localhost:8080 zu erreichen ist. Siehe Abbildung 6. Unter dem Menüpunkt “Tables” werden die Tupel beider Relationen R und S sowie die Output-Relation angezeigt.

```
162 func main() {
163     http.HandleFunc("/", index)             //welcome page
164     http.HandleFunc("/tables.html", tables) //generate and show tables and if hash join was executet also the joined
165
166     //Create a Filesystem so that the paths in the html files work
167     http.Handle("/assets/", http.StripPrefix("/assets/", http.FileServer(http.Dir(path.Join(tplPath, "/assets/")))))
168
169     //specifying that it should listen on port 8080. Blocks until the program terminates.
170     http.ListenAndServe(":8080", nil)
171 }
```

Abb. 6. Starten eines Webservers

6 Test auf Speedup und parallele Effizienz

Verwendet wurde die aktuelle Version von Go. Version 1.8.1.

6.1 Hardware

Die Tests wurden durchgeführt auf einem Rechner mit:

- Windows 10,
- Intel i5-2500K und
- 12GB Hauptspeicher.

Der Intel i5-2500K besteht aus vier Kernen mit einer Taktgeschwindigkeit von maximal 3,66GHz. Diese CPU besitzt keine Technik um pro Kern mehr als einen Thread verarbeiten zu können. Die Größen der Cache-Speicher setzen sich wie folgt zusammen: L1 = 256KB, L2 = 1.0MB und L3 = 6MB. Jeder der vier Kerne hat gemeinsamen Zugriff auf den L3 Cache. Zum Einsatz kommt eine 64Bit Architektur.

6.2 Testablauf

Die Größe der beiden Relationen R und S betragen jeweils 10Million Tupel. Wobei jedes der Attribute einen Speicherbedarf von 4Byte hat. Nachdem die Relationen erzeugt wurden, werden sie als JSON-Datei gespeichert. Jeder Test arbeitet nun mit diesen beiden Relationen. Beide Relationen werden auf Partitionen aufgeteilt, wobei die Anzahl der Partitionen, die doppelte Anzahl der Tupel aus R ist. Es werden nun in mehreren Durchgängen die beiden Relationen miteinander verknüpft. Zuerst nur mit einem Kern. Danach mit zwei, drei und vier Kernen. Jeder der vier möglichen Durchgänge wird mehrmals ausgeführt, wobei zum Schluss das Mittel der jeweiligen Durchgänge gebildet wird. Um nicht unnötig viel Speicher zu verbrauchen, wird die Output-Relation nach jedem Durchgang gelöscht. Die Speedupanalyse bezieht sich bei den Tests nur auf das Verbinden der einzelnen Partitionen und das erstellen der Output-Relation (Abb. 7).

```
106 //Tests the speedup of parallel computing of the algorithm
107 func testHashjoin(courseHashTable [][]mylib.Course, studentHashTable [][]mylib.Student) {
108     var i time.Duration = 1
109     var mean time.Duration = 0
110     var elapsed time.Duration
111 
112     fmt.Println("Start testing hashjoin")
113     mean = 0
114     for k := 1; k <= 4; k++ { //for one till four cores
115         for i = 1; i <= 10; i++ { //test each 10 times
116             start := time.Now()
117             Join(courseHashTable, studentHashTable, k) //join tables
118             elapsed = time.Since(start)
119             mean += elapsed
120             dataTables.JoinedT = dataTables.JoinedT[:0] //clear memory
121         }
122         fmt.Println("Time for joining tables", k, " cores: ", elapsed)
123     }
124 }
```

Abb. 7. Testroutine für Join-Operation

6.3 Erstellen der Daten

Die Daten werden in einer for-Schleife generiert und in ein Slice gespeichert (siehe Abb. 8). Jedes Element besitzt zwei Variablen mit jeweils 4B an Speicherbedarf. Beim Erstellen der Daten wird dem Programm keine Restriktion gegeben, wie viele Kerne es dafür verwenden darf. Hier versucht es alle zur Verfügung stehenden Kerne gleichzeitig zu nutzen. Jedoch werden hier keine weiteren Goroutines erstellt womit die Auslastung der CPU nur bei etwa 25 Prozent liegt.

6.4 Partitionieren der Daten

Beide Relationen werden nacheinander partitioniert. Dies dauert 5.9 Sekunden für beide Relationen bei einer Auslastung der CPU an die 25 Prozent.

```go
func CreateCourseTable(size int) []Course {
    courses = make([]Course, size) //size of the table
    idSize := 4 //byte
    nameSize := 4 //byte

    for i := 0; i < size; i++ {
        id := make([]byte, idSize)     //allocate memory
        name := make([]byte, nameSize) //allocate memory
        binary.LittleEndian.PutUint32(id, uint32(i))
        courses[i] = Course{Id: id, Name: name}
        fmt.Println("Building course table:", i*100/size, " %") //Progress
    }

    return courses
}
```

Abb. 8. Es wird die innere Relation R generiert

Es wurden insgesamt vier Tests durchgeführt. Zuerst wurde für jeden Kern eine eigene Goroutine gestartet. Beim zweiten Test für jede Partition die miteinander verbunden wurde. Test vier ermittelt die optimale Anzahl an Groutinen für die zugrundeliegende Hardware welche beim vierten Test zum Verbinden der Relationen herangezogen wurde.

6.5 Erster Test

Beim ersten Test, wurden die Partitionen auf die Anzahl der Kerne aufgeteilt, welche für das parallele abarbeiten vorgesehen waren und für jeden der Kerne nur eine Goroutine gestartet. Bei vier Kernen arbeitetet ein Kern mit dem ersten Viertel der Partitionen P_0 - $P_{n/4}$, ein zweiter Kern mit dem zweiten Viertel $P_{n/4}$ - $P_{n/2}$, usw. (Abb. 9).

Anzahl Kerne	Zeit [s]	Speedup	paralle Effizienz
1	53.38	1	1
2	23.27	2.29	1.15
3	15.30	3.49	1.16
4	10.52	10.52	1.27

Tabelle 1. Ergenbnis aus dem ersten Test

Hier kam es zu einem beinahe linearen Speedup bei der Verwendung von mehreren Kernen (Abb. 10). Allerdings kann es passieren, dass bei einer ungleichmäßigen Verteilung der Tupel auf die Partitionen, jeder Kern eine unterschiedliche Anzahl an Tupel miteinander verbindet. Dadurch, dass für jeden Kern nur eine Goroutine gestartet wurde, kann nachdem ein Kern mit seinen Partitionen fertig ist, dieser nicht mehr bei dem Verbinden der restlichen Partitionen mitwirken und es entsteht ein Lastenungleichgewicht.
Der Speedup von 2.3 von einem auf zwei Kerne wird dadurch erklärt, da der Algorithmus für die Ausführung auf mehreren Kernen konzipiert wurde. Bei der

```
79 //Splits the partitions to as many pieces as their are cores. For each piece a new goroutine starts.
80 func Join(courseHashTable [][]mylib.Course, studentHashTable [][]mylib.Student, cores int) {
81     hashTableSize := len(courseHashTable)
82     dataTables.JoinedT = make([]mylib.JoinElement, 0) //Output Table
83
84     runtime.GOMAXPROCS(cores) //Number of CPU's which the programm can use in parallal
85
86     //start for each core a goroutine
87     joinChannels := make([]<-chan mylib.JoinElement, cores)
88     for i := 0; i < cores; i++ {
89         joinChannels[i] = joinRangeofPartitions(courseHashTable[i*hashTableSize/cores:(i+1)*hashTableSize/cores], st
90     }
91
92     for n := range mylib.Merge(joinChannels) { //Merge alle channels
93         dataTables.JoinedT = append(dataTables.JoinedT, n) //Insert each output-tupel in output-table
94     }
95 }
```

Abb. 9. Partitionen werden auf die Anzahl der Kerne aufgeteilt

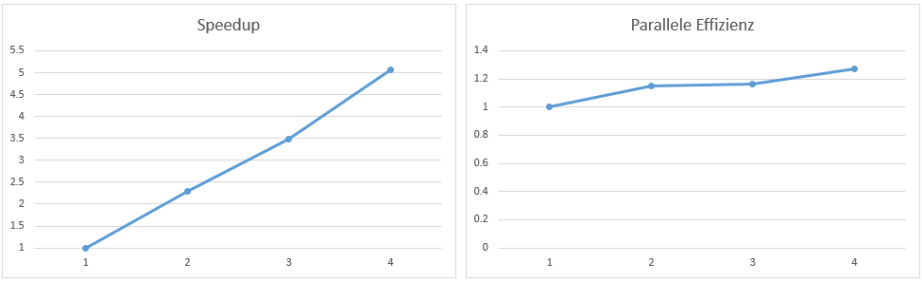

Abb. 10. Speedup und parallele Effizienz des ersten Tests

Ausführung auf nur einem Kern müsste keine Kommunikation zwischen den Goroutines stattfinden. In Funktion “joinRangeofPartitions()” (Abb. 4) wird eine neue Goroutine gestartet in welcher die Partitionen miteinander verbunden werden und in einen Channel gegeben werden. Auf der anderen Seite des Channels, in der Funktion “Join()” (Abb. 3), müssen die Werte aus dem Channel genommen werden. Dadurch ist die CPU ständig damit beschäftigt, zwischen den zwei Goroutines zu wechseln.
Laut dieses Tests ist die parallele Effizienz über alle Kerne immer steigend. Was darauf schließen würde, dass der Algorithmus bei zusätzlicher Parallelität immer effizienter wird.

6.6 Zweiter Test

Beim zweiten Test wurde die Aufteilung der Partitionen auf die einzelnen Kerne verändert. Nun wird für jede der Partitionen, welche miteinander verbunden werden, eine eigene Goroutine gestartet (Abb. 11). Dies soll den Vorteil haben, dass bis zum Schluss eine Thread-Parallelität unter den einzelnen Kernen vorliegt.

Zwar kommt es zu dem erhofften Ergebnis, dass mit jedem Kern der mehr zur Verfügung steht, es zu einem Speedup kommt, jedoch auf Kosten vom Hauptspeicher. Eine Goroutine benötigt zirka 2KB an zusätzlichen Speicher[9]. Bei einer Anzahl von 10Millionen Tupel und der doppelten Anzahl an Partitionen,

```
79 //Splits the partitions to as many pieces as their are partitions. For each piece a new goroutine starts.
80 func Join(courseHashTable [][]mylib.Course, studentHashTable [][]mylib.Student, cores int) {
81     hashTableSize := len(courseHashTable)
82     dataTables.JoinedT = make([]mylib.JoinElement, 0) //Output Table
83
84     runtime.GOMAXPROCS(cores) //Number of CPU's which the programm can use in parallal
85
86     //start for each partition a goroutine
87     joinChannels := make([]<-chan mylib.JoinElement, hashTableSize)
88     for i := 0; i < hashTableSize; i++ {
89         joinChannels[i] = joinRangeofPartitions(courseHashTable[i:(i+1)], studentHashTable[i:(i+1)])
90     }
91
92     for n := range mylib.Merge(joinChannels) { //Merge alle channels
93         dataTables.JoinedT = append(dataTables.JoinedT, n) //Insert each output-tupel in output-table
94     }
95 }
```

Abb. 11. Für jede Partition wird eine Goroutine gestartet

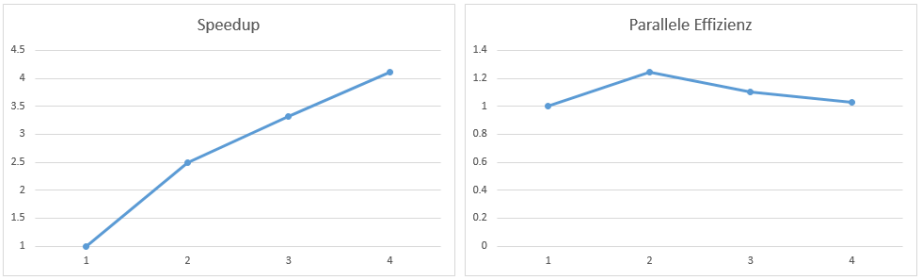

Abb. 12. Speedup und parallele Effizienz des zweiten Tests

wären das 40GB an Hauptspeicherkapazität, welche nur für die Erzeugung von Goroutines benötigt wird. Go hat mit dieser Hardware auch Probleme dabei ein Programm mit 20 Millionen Goroutines zu bewältigen. Es kommt bei 10 Millionen Tupel somit zu einem Absturz des Programms. Die Werte aus dem zweiten Test wurde mit nur 100000 Tupel realisiert, da ansonsten der Overhead dieses Algorithmus zu groß ist um in den Hauptspeicher zu passen.

6.7 Dritter Test

Aus den Erkenntnissen der zwei vorherigen Tests, wird nun nach der optimalen Anzahl an Goroutines für vier parallel arbeitende Threads für die Testdaten gesucht. Wie bereits zuvor erwähnt, liegt bei zu wenigen Goroutines eine Lastungleichgewicht zwischen den einzelnen Kernen vor. Bei zu vielen Goroutines gibt es einen großen Overhead und der Algorithmus ist zu sehr damit beschäftigt, zwischen den Goroutines zu kommunizieren. Abbildung 13 zeigt die Ausführungszeit in Sekunden bei der gleichzeitiger Verwendung von vier Kernen und einer Variablen Anzahl an Goroutines.

Es ist gut zu erkennen, dass bei 4 Goroutines es zu einem massiven Lastungleichgewicht kommt, welches bei einer Verdoppelung der Goroutines bereits erheblich abnimmt. Dies kann damit erklärt werden, dass bei der doppelten Anzahl an Goroutines zu den Kernen, die einzelnen Partitionen besser auf die

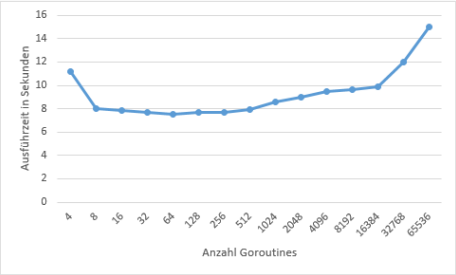

Abb. 13. Ausführzeit in Sekunden bei unterschiedlicher Anzahl an Goroutines

einzelnen Kerne aufgeteilt werden können. Ab einer Anzahl von 512 Goroutines wird der Overhead des Algorithmus so groß, dass er immer mehr mit der Kommunikation zwischen den Goroutines beschäftigt ist. Der Testablauf sah wie in Abbildung 14 abgebildet aus.

```
126 //Tests the optimal amount of goroutines
127 func testGoroutines(courseHashTable [][]mylib.Course, studentHashTable [][]mylib.Student) {
128     maxGoroutines = 2
129     for i := 0; i < 15; i++ { //Test up to 2048 goroutines
130         maxGoroutines *= 2 //double goroutines each round
131         var mean time.Duration = 0
132         for k := 0; k < 10; k++ { //test each 10 times
133             start := time.Now()
134             Join(courseHashTable, studentHashTable, 4)
135             elapsed := time.Since(start)
136             mean += elapsed
137         }
138         fmt.Println("Goroutines: ", maxGoroutines)
139         fmt.Println("Time for joining tables 4 cores: %s", mean/10)
140     }
141 }
```

Abb. 14. Testroutine für die optimale Anzahl an Goroutines

6.8 Vierter Test

Beim vierten Test wird die Anzahl der Goroutines, welche für das Verbinden der beiden Relationen erzeugt werden, vorgegeben. Dies entspricht Abbildung 3. Damit soll verhindern werden, dass der Algorithmus mit unnötig vielen Goroutines arbeitet. Es müssen ja nur genau so viele Goroutines erzeugt werden, wie dafür benötigt werden, um eine Thread-Parallälität über die gesamte Zeit der Joinoperation zu erhalten. In diesem Test wurde das Ergebnis aus dem Kapitel 6.7 herangezogen und mit 64 Goroutines gearbeitet.

Bei der Verwendung von 64 Goroutines und unter der Verwendung mehrerer Kerne, wird wieder eine nahezu lineare Steigerung des Speedups erzielt. Die parallele Effizienz von $E(P, N) \geq 1$, ist wieder zurückzuführen auf das parallele Design des Algorithmus. In Abbildung 16 sind die Werte aus den Tabellen 1 und 2 graphisch abgebildet. Im Vergleich zum ersten Test aus Kapitel 6.5,

Anzahl Kerne	Zeit [s]	Speedup	paralle Effizienz
1	45.36	1	1
2	15.96	2.84	1.42
3	10.23	4.43	1.47
4	7.43	6.11	1.53

Tabelle 2. Ergenbnis aus dem vierten Test

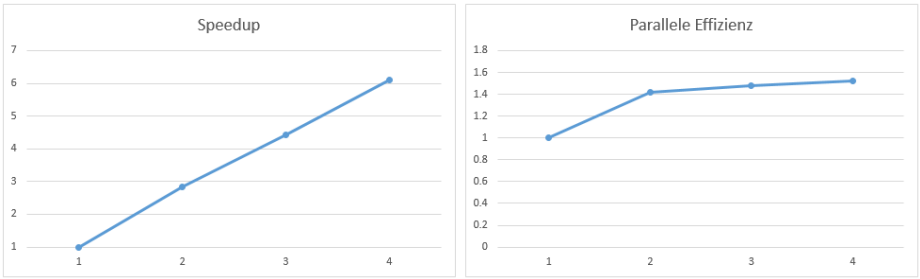

Abb. 15. Speedup und parallele Effizienz des vierten Tests

kommt es hier zu einer generellen Leistungssteigerung durch das reduzieren des Lastungleichgewichts.

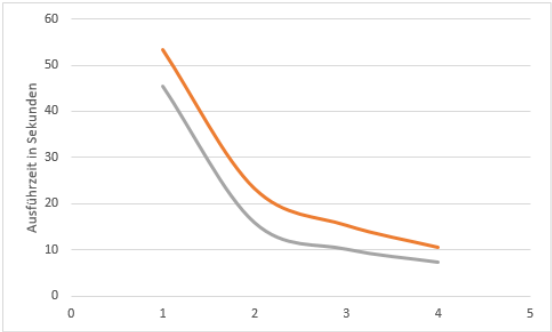

Abb. 16. Ausführzeit von Test1 und Test4

7 Conclusio

Wie aus den Tests aus Kapitel 6 hervor geht, ist die Anzahl der Goroutines ein entscheidender Faktor für die Ausführungszeit des Algorithmus. Bei je einer Goroutine pro Kern kommt es zu einem erheblichen Lastenungleichgewicht. Bei zu vielen, ist der Algorithmus nur mehr mit der Kommunikation zwischen

den Goroutines beschäftigt, was sich wiederum negativ auf die Ausführungszeit auswirkt. 40 Millionen Goroutines waren für die Hardware auf denen getestet wurde zu viel. Es kam hierbei zu einem Programmabbruch. Bei dem Test aus Kapitel 6.7 geht hervor, dass die optimale Anzahl an Goroutines bei vier parallel arbeitenden Kernen bei 64 liegt. Dies kommt natürlich auf die zugrunde liegende Hashfunktion an, die für die Partitionierung der Relationen verwendet wird. Hier konnte beim vierten Test (Abb. 6.8), durch die Optimierung der Goroutines und unter der Verwendung mehrerer Kerne, eine generelle Steigerung der Performance von 1.46 erzielt werden (Abb. 16). Dies Erkenntnis wird mit dem Ergebnis aus dem dritten Test (Kapitel 6.7) unterstützt.
Der Speedup $S(P, N) \geq P$ wird begründet durch das Design des Algorithmus. Dieser ist für das parallele Ausführen konzipiert. Bei der Verwendung eines einzelnen Kerns, ist der Algorithmus unnötigerweise damit beschäftigt, zwischen den Goroutines zu wechseln. Dies erzeugt Overhead, welche auf Kosten der produktiven Arbeitszeit der CPU fällt. In Abbildung 15 ist zu sehen, dass der entworfene Algorithmus in dieser Arbeit einen nahezu linearen Speedup bei der Verwendung mehrerer Kerne aufweist. Die parallele Effizienz steigt sogar mit jedem weiteren Kern. Hier wird angenommen, dass sich der Algorithmus, ab einer gewissen Anzahl an parallelen Operationen, sich einer konstanten parallelen Effizienz annähern wird. Das würde heißen, dass dieser Grace Hash Join Algorithmus in Go eine starke Skalierbarkeit aufweist. Dies muss allerdings auf einer dementsprechenden Hardware nachgewiesen werden.

Literatur

1. Mach, W., Schikuta, E., Pittl, B.: Analysis of a parallel grace hash join implementation on the cell processor. In: 2014 International Conference on High Performance Computing Simulation (HPCS). (July 2014) 1018–1022
2. YUAN, T., LIU, Z., LIU, H.: Optimizing hash join with mapreduce on multi-core cpus. IEICE Transactions on Information and Systems **E99.D**(5) (2016) 1316–1325
3. Balkesen, ., Teubner, J., Alonso, G., Özsu, M.T.: Main-memory hash joins on modern processor architectures. IEEE Transactions on Knowledge and Data Engineering **27**(7) (July 2015) 1754–1766
4. Meyerson, J.: The go programming language. IEEE Software **31**(5) (Sept 2014) 104–104
5. Deng, Y.: Applied Parallel Computing. World Scientific (2012)
6. : The go programming language. https://golang.org/ (Accessed on 04/17/2017).
7. : The go programming language specification - the go programming language. https://golang.org/ref/spec (Accessed on 04/17/2017).
8. : runtime - the go programming language. https://golang.org/pkg/runtime/ (Accessed on 04/18/2017).
9. : Go 1.4 release notes - the go programming language. https://golang.org/doc/go1.4 (Accessed on 04/22/2017).