Parallele Algorithmen

Richard Riedel

Professor: Prof. Dr. Christel Baier

Betreuer: Max Korn

Technische Universität Dresden
Fakultät Informatik

30. September 2020

1 Abstract

Seit etwa 2005 ist es aufgrund von ILP- und Powerwalls nicht mehr möglich die Single-thread-Geschwindigkeit von Prozessoren in dem Maße zu erhöhen, wie man es aus den vergangenen Jahren gewohnt war. Dadurch, dass man die Taktgeschwindigkeit nicht mehr mit jeder Generation erhöhen kann und auch die Parallelisierung von Assemblerbefehlen an ihre Grenzen gestoßen ist, musste man ab diesem Zeitpunkt andere Wege gehen, um die Rechengeschwindigkeit von Prozessoren weiter zu erhöhen. Daher begannen Firmen Prozessoren mit mehreren Kernen, welche parallel arbeiten, zu bauen. Aufgrund dieser neuen Art von Prozessoren wurde es nötig, Algorithmen so anzupassen, bzw. neu zu entwickeln, dass sie die neu gewonnenen Ressourcen nutzen können, um in ihrer Ausführung schneller zu werden.

In dieser Ausarbeitung, welche hauptsächlich auf Kapitel 27 des Buches "Introduction to Algorithms, Third Edition" basiert, soll es darum gehen, wie man solche Algorithmen analysiert und wie man sie entwickelt, bzw. wie man bereits bestehende Algorithmen parallelisiert. Dabei wird zunächst auf Probleme eingegangen, welche entstehen, wenn man von der seriellen Ausführung eines Algorithmus abweicht. Anschließend wird gezeigt, wie Concurrency Plattformen uns dabei helfen mit diesen Problemen umzugehen. Genauer wird dabei auf die Klasse des Dynamic Multithreading eingegangen, welche sich dadurch auszeichnet, dass man Algorithmen, durch einfügen einfacher Schlüsselwörter in den seriellen Code, parallelisieren kann ohne Änderungen am Algorithmus an sich vornehmen zu müssen. Was bei diesem Vorgehen schief gehen kann wird im Abschnitt Race Conditions behandelt, welcher zeigt wie durch Parallelisierung unerwartete Ergebnisse entstehen können und wie dies zu einem großen Problem werden kann. Daraufhin wird besprochen wie man die Performance eines parallelen Algorithmus bestimmen kann und von welchen Faktoren diese abhängt, um anschließend Aussagen über die Parallelisierbarkeit und mögliche Bottlenecks des Algorithmus treffen zu können. Zusätzlich wird noch kurz auf den Scheduler der Concurrency Plattform eingegangen und dessen Funktion genauer erläutert und ein Modell zur Darstellung paralleler Algorithmen eingeführt.

Im Folgenden werden dann alle Erkenntnisse zusammengeführt, um das Problem der Matrix-Multiplikation zu analysieren und auf verschiedenen Wegen zu parallelisieren. Um die Parallelisierung zu testen und zu überprüfen ob unsere, auf den eingeführten Metriken und Gesetzen beruhenden, Vermutungen in der Praxis tatsächlich auftreten ist im letzten Abschnitt der Ausarbeitung ein Python3 Programm geben. In diesem Programm ist es möglich einen Matrixmultiplikations-Algorithmus auf einem beliebigen Teil der zur Verfügung stehenden Kerne auszuführen um unsere Vermutungen bezüglich Auslastung und Laufzeit zu überprüfen.

Inhaltsverzeichnis

2 Einführung

Bevor wir über Parallelisierung sprechen kommen wir zunächst zur Ursache unseres Anliegens. Bis zum Jahr 2005 war es für Entwickler normal ein paralleles Programm zu schreiben und falls es zu langsam war auf die nächste Generation von Prozessoren zu warten, um die Ausführungsgeschwindigkeit zu erhöhen. Getreu Moores Law verdoppelte sich nicht nur die Transistorzahl pro Fläche etwa jährlich, sondern auch die Singlethread-Performance stieg exponentiell an. Um das Jahr 2005 entstanden dabei jedoch Probleme, durch ILP und Powerwalls wurde die Singlethread-Performance plötzlich limitiert und konnte ihr gewohntes Wachstum nicht mehr weiter fortsetzen.

ILP (Instruction Level Parallelism) -Walls beschreibt hierbei, dass es nicht mehr möglich ist mehr Befehle aus demselben Befehlsstrom gleichzeitig auszuführen als bisher, diese "Wall" kommt dadurch zustande, dass Befehle in einem Befehlsstrom Abhängigkeiten untereinander haben. Solche Abhängigkeiten müssen nun in der Ausführung des Programms beachtet werden, daher lohnt es sich nicht immer mehr Parallelismus einzubauen, da dann ggf. sehr viel Zeit mit Warten verbracht wird. Power-Walls auf der anderen Seite entstehen dadurch, dass ein höherer Prozessortakt mehr Leistung braucht und somit auch mehr Wärme abgibt. Wurde bislang also die Prozessortaktgeschwindkeit hauptsächlich dadurch limitiert, dass die Transistoren nicht fähig waren höher zu Takten, ist nun ein neues nicht so leicht zu lösendes Problem aufgetaucht. Denn ab einer bestimmten Menge an Energie pro Fläche kann diese nichtmehr effizient abgeführt werden. Das Bedeutet in der Praxis entweder man muss für jeden seiner Prozessoren eine enorm umfangreiche Kühlung zur Verfügung stellen, oder sich eine Möglichkeit überlegen auch mit einem geringeren Takt seine Rechenleistung zu erhöhen.

Hier kommen wir zum Thema des parallelen Programmierens, wobei die meisten der folgenden Aussagen und Erkenntnisse zu diesem Thema auf Kapitel 27 des Buches "Introduction to Algorithms, Third Edition" basieren. Wie bereits angesprochen ist zwar die Singlethread Performance, also die Ausführungsgeschwindigkeit, die ein einzelner Prozessor erreichen kann, limitiert, jedoch wächst die Transistorzahl, die uns Pro Fläche zur Verfügung steht weiterhin exponentiell an. So ergibt sich für uns die Möglichkeit mehrere Recheneinheiten, also Kerne, auf einen Prozessor zu implementieren. Womit wir theoretisch unsere Rechenleistung verdoppeln könnten, indem wir unsere Kernanzahl Verdoppeln. Genau dieses Prinzip wurde ab dem Jahr 2005 etabliert, statt einen Kern immer schneller zu machen wurde begonnen immer mehr Recheneinheiten auf einem Prozessor zu implementieren. Daher liegt es nun an uns diese Mehrzahl an Prozessoren und die damit verbundene Mehrleistung für unsere Applikationen zu nutzen. Wie wir dabei vorgehen, was es für Probleme geben kann und wo unsere Grenzen liegen wird im Folgenden erläutert.

3 Grundlagen

Im Grunde genommen geht es bei der Parallelisierung von Algorithmen darum, den linearen Berechnungsstrang durch einbauen von Verzweigungen und das Anschließende wieder Zusammenführen dieser Verzweigten Berechnungsfäden, im folgenden oft Threads genannt, so umzubauen, dass man am Ende eine schnellere Ausführungszeit erreicht und dennoch das richtige Ergebnis erhält. Vorausgesetzt ist dafür natürlich, dass man mehrere Ausführungseinheiten zur Verfügung hat. Diese Ausführungseinheiten werden im Folgenden auch oft als Kerne oder Prozessoren bezeichnet.

3.1 Probleme der parallelen Programmierung

Das Ziel einer schnelleren Ausführung unseres Algorithmus erreichen wir jedoch nicht ohne Aufwand. Um die neu gewonnen Ressourcen sinnvoll zu nutzen muss man sich mit einigen Problemen auseinandersetzen. Welche das sind und wie sie entstehen wird hier kurz erläutert.

Da es passieren kann, dass wir mehr Berechnungsfäden zur Verfügung haben als Kerne, unsere Threads verschieden komplex sind, oder falls unser parallelisiertes Programm einfach auf einem älteren Prozessor ausgeführt werden soll, welcher nur über einen Kern verfügt, ergibt sich für uns die Frage, welche Reihenfolge zur Ausführung der Berechnungsfäden am sinnvollsten ist bzw. welche Reihenfolge überhaupt unser Programm korrekt auf die Recheneinheiten abbilden kann. Diese Aufgabe wird als Scheduling bezeichnet. Im Allgemeinen kümmert sich das Betriebssystem darum unsere Threads auf die Kerne zu verteilen, wir müssen dabei nur Threads erstellen und sie an das Betriebssystem weiter geben. Der Einfachheit halber nehmen wir jedoch an wir könnten die Threads direkt auf die Recheneinheiten verteilen. Im Folgenden werden wir mit statischen Threads arbeiten. Statische Threads sind Threads, welche über die gesamte Laufzeit der Berechnung existieren. So besteht unser Problem also darin unsere Berechnungsarbeit möglichst gleich auf die Threads zu verteilen und sie in der richtigen Reihenfolge auszuführen.

Ein weiterer wichtiger Punkt ist die Frage nach der Kommunikation zwischen unseren einzelnen Threads. Sind quasi zwei Berechnungsfäden voneinander abhängig und laufen sie auf verschiedenen Kernen mit eigenen Caches stellt sich die Frage wie der Thread, welcher von dem Ergebnis des anderen Threads abhängt an dieses Ergebnis heran kommt. Hierbei müssen aufwändige Kommunikationsprotokolle implementiert werden und es gibt wieder zahlreiche Problemfälle die beachtet werden müssen um einen reibungslosen Ablauf zu garantieren.

Das dritte große Problem, welches uns begegnet ist die Frage wie der Compiler den von uns generierten Parallelismus in Maschinensprache übersetzt, sodass die Prozessoren beispielsweise die eben angesprochene Kommunikation untereinander durchführen können.

Ein weiters Problem stellen nun Cache Kohärenzen dar. Wie bereits besprochen kommt es vor, dass ein Kern Werte aus dem Cache eines anderen Kerns benötigt. Zusätzlich gibt es aber auch noch den Fall, dass zwei Kerne eine Instanz derselben Variable in ihrem Cache haben. Hierbei muss garantiert sein, dass für den Fall, dass ein Prozessor diese Variable verändert der andere davon mitbekommt und es somit zu keinem inkonsistenten Zustand, welcher dann zu einer Race Condition führen würde, kommt. Das Problem der Race Conditions wird zu einem späteren Punkt in dieser Ausarbeitung noch einmal etwas genauer erläutert, am aktuellen Punkt reicht es zu wissen, dass eine Race Condition dann auftritt, wenn mehrere Prozessoren auf derselben Variable arbeiten und durch ungünstige Abfolge des Schreibens und Rückschreibens Berechnungen verloren gehen.

Unsere letzte Frage für diese Aufzählung ist nun, was die untätigen Kerne machen, wenn beispielsweise ein sequenzieller Algorithmus ausgeführt wird. Da es äußerst ineffizient wäre, wenn alle Prozessoren gleich hoch Takten würden, obwohl nur einer Berechnungen durchführt, ist es bei den meisten modernen Prozessoren möglich, dass die einzelnen Kerne unabhängig voneinander z.B. in einen Energiesparmodus gebracht werden können.

3.2 Concurrency Plattforms

Da die grade angesprochenen Probleme unser Ziel der Parallelisierung unserer Algorithmen sehr aufwändig erscheinen lassen und es ineffizient wäre, wenn sich jeder Entwickler mit den gleichen Problemen befassen müsste, wurden in den vergangenen Jahren zahlreiche Concurrency Plattformen entwickelt. Concurrency Plattformen kann man sich als Zwischenebene zwischen Entwickler und Betriebssystem bzw. Hardware vorstellen. Je nach Plattform kümmern sie sich um beinahe alle in 3.1 besprochenen Probleme. So muss der Programmierer nur noch bestimmen, welche Teile seines Codes nebeneinander ausgeführt werden können ohne, dass es Abhängigkeiten zwischen ihnen gibt, die beispielsweise zu Race Conditions führen könnten.

Hierbei gibt es zahlreiche verschiedene Arten von Concurrency Plattformen. Unser Fokus soll im Folgenden auf der Klasse des dynamischen Multithreading liegen. Wobei der Entwickler, mithilfe von einfachen Schlüsselwörtern, kennzeichnen kann, welche Teile, seines zu Beginn seriellen Programms, parallelisierbar sind. Also gleichzeitig ausgeführt werden können ohne, dass dies zu Fehlern führt. Der Vorteil bei dieser Art von Concurrency Plattform liegt ganz klar in der Einfachheit für den Entwickler, er kann hier wie gewohnt ein serielles Programm schreiben und erst anschließend durch Einbauen der Schlüsselwörter dieses Programm optimieren. Natürlich kann es von Vorteil sein schon bei den Überlegungen zur Programmlogik die spätere Parallelisierbarkeit des Codes in Betracht zu ziehen. Auf diese Weise können auch bereits bestehende serielle Programme im Nachhinein parallelisiert werden.

Weitere Möglichkeiten für Concurrency Plattformen sind unter anderem das Einbinden von parallelen Bibliotheken, wobei man bereits bestehenden Code, welcher schon

parallelisiert wurde, verwendet, um sein Programm zu beschleunigen, oder Laufzeitbibliotheken auf welche vom Compiler zugegriffen werden kann. Ein gutes Beispiel für parallele Bibliotheken ist die Matrixmultiplikation, da diese Aufgabe sich enorm gut parallelisieren lässt und in der Praxis sehr oft gebraucht wird, gibt es für die meisten Programmiersprachen Bibliotheken, welche parallelisierte Funktionen für sie beinhalten.

3.3 Dynamic Multithreading

Die Klasse des Dynamic Multithreading zeichnet sich wie bereits erwähnt dadurch aus, dass der Programmierer, mithilfe einfacher Schlüsselwörter, in seinem Code kennzeichnen muss, welche Berechnungen voneinander unabhängig sind und somit nebeneinander ausgeführt werden können.

Dabei gibt es zwei Typen von Parallelität. Zunächst betrachten wir die "Schleifenparallelität". Wobei man in unserem Fall durch das Schlüsselwort **parallel** kennzeichnet, dass die einzelnen Iterationen einer Schleife voneinander unabhängig sind und somit parallel ausgeführt werden können. Ein Beispiel für eine solche Schleife wäre eine Iteration über einen Array, wobei wir jedes Element inkrementieren wollen, ein Gegenbeispiel, wo diese Art der Parallelisierung nicht funktionieren würde, wird im folgenden Abschnitt "Race Conditions" gegeben. Die zweite Art von Parallelität ist die "Geschachtelte Parallelität". Hierbei kennzeichnet man in unserem Fall durch das Schlüsselwort **spawn**, dass eine aufgerufene Funktion parallel zu den folgenden Funktionsaufrufen ausgeführt werden kann. Also, dass diese nicht vom Rückgabewert der "gespawnten" Funktion abhängen. Hierbei kann es sein, dass nicht alles was Folgt unabhängig vom Rückgabewert der Funktion ist, sondern beispielsweise der übernächste Funktionsaufruf den Rückgabewert unserer Funktion als Parameter benötigt. In diesem Fall gibt es das Schlüsselwort **sync** dieses sagt unserer Concurrency Plattform, dass hier gewartet werden muss bis alle "gespawnten" Funktionen abgeschlossen sind. Ein Return impliziert immer auch einen sync, dass bedeutet, dass immer bevor eine Funktion etwas zurückgegeben kann zunächst alle "gespawnten" Unterfunktionen abgeschlossen sein müssen.

Ein Beispiel dafür ist Folgende Funktion.

```
function HARDWORK(x)
    sleep(5000);
    return x += 1;
end function
function MAIN
    var x = Hardwork(5);
    var y = Hardwork(10);
    printf(x+y);
end function
```

$\Rightarrow$

```
function HARDWORK(x)
    sleep(5000);
    return x += 1;
end function
function MAIN
    var x = spawn Hardwork(5);
    var y = Hardwork(10);
    sync
    printf(x+y);
end function
```

Hier wird unsere Main Funktion parallelisiert indem wir die beiden Funktionsaufrufe von Hardwork gleichzeitig statt nacheinander ausführen dies spart uns im Idealfall 50% der Zeit. Zusätzlich verwenden wir den **sync** Befehl, um sicherzugehen, dass die Ergebnisse unserer Funktionen zur Verfügung stehen, wenn wir sie in printf(x+y) ausgeben wollen.

3.4 Race Conditions

Race Conditions sind eine der größten Gefahren der parallelen Programmierung da sie relativ zufällig auftreten können. Dieser Zufall entsteht, da es sich nur extrem schwer sagen lässt wann genau welche Operation eines Threads auf welchem Prozessor ausgeführt wird. Grund dafür ist der Nondeterminismus unseres Schedulings. Um erst einmal zu verstehen, was eine Race Condition überhaupt bedeutet hier ein kleines Beispiel in Pseudocode

```
var x = 0
for i = 1 to 2 do
    x = x + 1;
end for
```

Bei diesem Codeabschnitt wird zunächst eine Variable mit 0 initialisiert anschließend wird sie bei serieller Ausführung um 2 hochgezählt und hat am Ende des Programms den Wert 2. Parallelisiert man dieses Programm nun folgendermaßen und achtet dabei nicht darauf, dass die Schleifeniterationen nicht logisch unabhängig voneinander sind erhält man folgenden Code.

```
var x = 0
parallel for i = 1 to 2 do
    x = x + 1;
end for
```

Hier haben wir gekennzeichnet, dass die Iterationen der Schleife parallel ausgeführt werden können. Dies wird dann von unserer Concurrency Plattform auch so gehandhabt. Nun kann es zwar passieren, dass unsere Schleifendurchläufe nacheinander ausgeführt werden und wir das korrekte Ergebnis 2 am Ende in unserer Variable x stehen haben, jedoch kann es auch passieren, dass die beiden Schleifendurchläufe "gleichzeitig" (insofern gleichzeitig, dass im Folgenden beschriebener Fall auftritt) ausgeführt werden. In diesem Fall wiederum ist es möglich, dass beide Ausführungseinheiten den gleichen Wert $x = 0$ lesen und dann die Funktion in der Schleife auf diesen Wert anwenden. So schreibt dann eine Ausführungseinheit seinen berechneten Wert 1 zurück in den Hauptspeicher, und anschließen schreibt die andere Ausführungseinheit ebenfalls noch einmal 1 auf denselben Speicherplatz und wir haben, nach Ausführung unserer Funktion, fälschlicherweise den Wert 1 in unserer Variable x stehen. Dies wäre nicht so fatal, wenn dieser Fall konsistent auftreten würden, denn dann würde man relativ schnell mer-

ken, dass das Programm nicht das Ergebnis liefert, welches man erwartet. Die große Gefahr besteht hier, wie schon angedeutet, darin, dass es sein kann, dass in einem sehr großen Teil der Programmausführungen unsere beiden Schleifendurchläufe eben nicht gleichzeitig ausgeführt werden, sondern leicht versetzt oder nacheinander. So erhalten wir im ungünstigsten Fall, beim Testen unseres Programms, das richtige Ergebnis und erst nach jahrelangem zuverlässigen laufen des Programms tritt der Fehler auf und führt Beispielsweise zum Systemausfall.

3.5 Performancemaße

Um zu bestimmen wie viel Vorteil wir durch die Parallelisierung unserer Algorithmen ehrhalten bzw. mit wie viel Vorteil wir rechnen können, werden wir im Folgenden einige Metriken zur Beschreibung unserer parallelen Algorithmen einführen. Dafür und für alle folgenden Betrachtungen müssen wir ein paar wenige Vereinfachungen treffen, da wir sonst zu tief ins Detail gehen und das eigentliche Ziel der Arbeit verfehlen. Zunächst werden wir davon ausgehen, dass alle unsere Kerne die gleich Rechenleistung haben. Zusätzlich werden wir noch die Zeiten für das Scheduling ignorieren, wobei dies sogar relativ realitätsnah ist, da in vielen Fällen der Aufwand für das Scheduling im Vergleich zum restlichen Rechenaufwand minimal ist. Die Gleichheit der Rechenleistung unserer Kerne hingegen ist eine etwas stärkere Vereinfachung, da wir so nicht betrachten, dass Beispielsweise noch Rechenkapazitäten für das Betriebssystem oder andere Programme reserviert sind und wir somit nicht auf allen Prozessoren immer die volle CPU-Zeit bekommen können.

Beginnen wir nun mit einer ganz grundlegenden Eigenschaft eines Algorithmus, der Ausgeführten Arbeit.

Definition 3.5.1 (Arbeit T_1)**.** *Die Arbeit T_1 beschreibt wie viel Arbeit insgesamt zur Verfügung steht. Was Arbeit in diesem Zusammenhang bedeutet, lässt sich auf vielerlei Weise veranschaulichen man kann sich beispielsweise die Arbeit als Zeiteinheiten die zur Ausführung aller Berechnungen gebraucht werden oder als Gesamtmenge der Berechnungsaufgaben vorstellen. Zusätzlich kann man diese Größe als die Zeit die ein Prozessor zur Ausführung des Algorithmus benötigen würde bzw. die Zeit bei serieller Ausführung bezeichnen.*

Definieren wir nun die Zeitspanne.

Definition 3.5.2 (Zeitspanne T_∞)**.** *Die Zeitspanne beschreibt die minimale Ausführungszeit des Programms. Das Unendlichkeitszeichen kann man hier als Anzahl der Prozessoren Auffassen. So beschreibt unsere Zeitspanne die Zeit die unser Programm auf unendlich vielen Prozessoren brauchen würde. Zum weiteren Verständnis kann man sagen, dass die Zeitspanne die längste Folge voneinander abhängiger Berechnungen ist. Veranschaulicht wird dies zusätzlich im Absatz 3.7.*

Die nächste Definition beschäftigt sich mit der Zeit auf P Prozessoren.

Definition 3.5.3 (Zeit auf P Prozessoren T_P)**.** *Die Zeit auf P Prozessoren gibt uns die Laufzeit die unser Programm bei idealen Scheduling auf P Prozessoren benötigen würde.*

Ein kleines Beispiel zur Veranschaulichung dieser Größe wäre ein Programm welches zwei Funktionen aufruft. Die eine Funktion nimmt eine Integer Variable und führt einen leftshift aus. Die andere Funktion führt mit einer Integer Variable einen rightshift aus. Die beiden Funktionen sollten also in etwa die gleiche Komplexität haben. Anschließend gibt unser Programm die Summe der beiden Variablen aus. Betrachten wir der Einfachheit halber die Summation und den Print als gleichaufwändig wie die beiden Shifts. Dann erhalten wir eine Arbeit T_1 von 4, unsere Zeitspanne ist hier offensichtlich $T_\infty = 3$ da der Print nach dem Summationsbefehel und dieser nach den beiden Shifts ausgeführt werden muss und nur die beiden Shifts parallelisierbar sind.

Damit sieht man leicht, dass unsere Zeit auf P Prozessoren immer größer oder gleich der Zeit auf unendlich vielen Prozessoren sein muss. Also, dass folgendes Lemma, welches auch Zeitspannengesetz genannt wird, gilt.

Lemma 3.5.4 ($T_P \geq T_\infty$)**.** *Die Laufzeit auf P Prozessoren ist immer mindestens genau so groß wie die Laufzeit auf unendlich vielen Prozessoren.*

Beweis. Offensichtlich gilt für jeden Algorithmus, dass sich P Prozessoren immer auf unendlich vielen simulieren lassen, indem man nur P Prozessoren für die Berechnung verwendet. Daraus folgt, dass man auf begrenzter Prozessorenzahl offensichtlich nicht schneller als auf unendlich vielen Prozessoren sein kann. Langsamer hingegen kann man sein, was deutlich wird wenn man unser Beispiel weiter betrachtet. Wir haben für T_n mit $n > 1$ immer den Wert $T_n = 3$, da hier zu Beginn beide Shifts parallel ausgeführt werden können und anschließend eine der n Recheneinheiten die letzten beiden Schritte ausführt. Für $T_P = T_1$ haben wir hingegen den Wert 4, welcher größer ist als T_∞, was unsere Aussage abschließend beweist. □

Überlegt man weiter, kommt man zum Schluss, dass man die Arbeit von 4 Schritten, auf zwei Prozessoren, eigentlich in 2 Schritten schaffen können sollte, indem man einfach jeden Prozessor zwei Schritte ausführen lässt. Diese Überlegung führt uns zum nächsten Lemma, welches auch Arbeitsgesetz genannt wird.

Lemma 3.5.5 ($T_P \geq \frac{T_1}{P}$)**.** *Die Laufzeit auf P Prozessoren kann höchstens so gering sein, wie die Laufzeit die sich ergibt, wenn man die Arbeit T_1 durch die Anzahl P der Prozessoren teilt.*

Beweis. Um dieses Lemma zu Beweisen gilt es zu zeigen, dass es nicht möglich ist, dass T_P kleiner als $\frac{T_1}{P}$ ist. Dies wollen wir hier durch Widerspruch beweisen. Nehmen wir dazu an, dass gilt $T_P < \frac{T_1}{P}$. Aus dieser Gleichung würde folgen, dass $T_P \cdot P < T_1$ ebenfalls gilt. Das wiederum würde bedeuten, dass wir durch Ausführen unseres Algorithmus auf P Prozessoren unsere Gesamtarbeit T_1, also die auszuführenden Berechnungen verringern könnten, was ein Widerspruch zur Definition von T_1 ist. □

Dass die Zeit auf P Prozessoren größer sein kann als $\frac{T_1}{P}$ haben wir bereits mit dem vorangegangenen Beispiel gezeigt (T_n mit $n > 1$ war dort gleich 3 was wiederum in jedem Fall größer ist als $\frac{T_1}{P} = \frac{4}{n}$ mit $n > 1$). Um die Möglichkeit der Gleichheit zu zeigen betrachten wir einfach dasselbe Beispiel ohne die Summation und die Ausgabe. So haben wir nur noch ein T_1 von 2 und können unseren Algorithmus offensichtlich in einem Schritt ausführen, wenn wir beide Berechnungen gleichzeitig ausführen lassen. Für diese Berechnungen verwenden wir nun zwei Recheneinheiten und erhalten so $T_P \geq \frac{T_1}{P} = 1 = \frac{2}{2} = T_P$ was gilt.

Definieren wir abschließend den Beschleunigungsfaktor, die Parallelität, perfekte lineare Beschleunigung und den Spielraum.

Definition 3.5.6 (Beschleunigungsfaktor $\frac{T_1}{T_P}$)**.** *Der Beschleunigungsfaktor ist der Faktor, welcher angibt, um wie viel schneller unser Programm auf P Prozessoren im Vergleich zur benötigten Zeit auf einem Prozessor ist.*

Definition 3.5.7 (Parallelität $\frac{T_1}{T_\infty}$)**.** *Die Parallelität beschreibt so zu sagen den maximalen Beschleunigungsfaktor. Hat man diesen Wert erreicht kann man die Berechnung nicht weiter durch Hinzufügen weiterer Recheneinheiten beschleunigen. Der Berechnete Wert gibt außerdem an wie viele Prozessoren man für eine hohe Beschleunigung braucht (die Anzahl der Prozessoren für die maximale Beschleunigung lässt sich mit der Parallelität nicht zuverlässig herausfinden).*

Hat man zum Beispiel 5 Berechnungen zu tun also eine Arbeit von $T_1 = 5$ und eine Zeitspanne von $T_\infty = 2$ so erhält man eine Parallelität von $2,5$, das bedeutet für uns, dass wir mit 3 Prozessoren bereits unsere maximal mögliche Beschleunigung erreichen können. (Falls sich die 5 Arbeitsschritte günstig verteilen und nicht die letzten 4 Schritte vom Ersten abhängen, in diesem Fall würden wir dann 4 Recheneinheiten benötigen) Bei nur zwei Prozessoren hingegen hätten wir in jedem Fall noch ein Beschleunigungspotential, für diese Aussage ist es egal wie die Abhängigkeiten zwischen den einzelnen Berechnungen sind.

Definition 3.5.8 (perfekte lineare Beschleunigung)**.** *Eine sogenannte Perfekte lineare Beschleunigung kann man erreichen, wenn die Parallelität größer als die Anzahl an Prozessoren, welche wir verwenden, ist. (Auch hier müssen die Abhängigkeiten zwischen den einzelnen Berechnungen günstig sein) Perfekte lineare Beschleunigung bedeutet dabei, dass wir jeden Prozessor zur gesamten Laufzeit voll auslasten können, also, dass unser Beschleunigungsfaktor gleich der Anzahl an Prozessoren ist.*

Definition 3.5.9 (Spielraum $\frac{T_1}{T_\infty}/P$)**.** *Der Spielraum Teilt die Parallelität durch die Anzahl der Prozessoren und gibt uns so einen Faktor, welcher angibt um wie viel unsere Parallelität unsere Prozessoren Zahl übersteigt bzw. wie viel geringer sie im Vergleich zur Prozessoren Zahl ist. Aus dem Spielraum kann man leicht ablesen ob das Programm mit mehr Prozessoren noch beschleunigt werden kann. Hier gilt, ist der Spielraum kleiner 1 so limitiert die Parallelität unseres Programms die Beschleunigung, ist der Spielraum größer 1 so ist die Prozessoren Zahl der limitierende Faktor. Zur Veranschaulichung dieser Größe kann man einfach das zur Parallelität genannte Beispiel betrachten.*

3.6 Scheduler

In diesem Abschnitt gehen wir noch einmal genauer auf den Scheduler ein, welcher ein wichtiger Bestandteil der meisten Concurrency Plattformen ist (zumindest derer die dem Entwickler das Scheduling abnehmen). Die Aufgabe des Schedulers ist wie bereits angesprochen den Workload ideal auf unsere Threads zu verteilen und diese in einer günstigen Reihenfolge auszuführen. (bzw. in der Praxis vom Betriebssystem ausführen zu lassen) Im Allgemeinen ist die Aufgabe des Scheduling sehr anspruchsvoll, da der Scheduler live arbeiten muss, dass bedeutet er weiß nicht wann ein neuer Thread auftaucht und wie anspruchsvoll dieser sein wird, dennoch hat er die Aufgabe sie sinnvoll zu verteilen. Zudem muss zusätzlich noch verteilt gearbeitet werden, dass wiederum bedeutet, dass jeder Thread der wiederum neue Threads spawnt das Scheduling Problem für sich selbst erneut lösen muss und für einen optimalen Ablauf theoretisch eine Kommunikation zwischen diesen einzelnen Schedulern nötig ist.

Da die Analyse dieser Art von Schedulern enorm schwer ist reicht es für unsere Zwecke einen unverteilten Scheduler zu betrachten. Das bedeutet, dass unserer Scheduler zu jedem Zeitpunkt den globalen Zustand unserer Berechnung kennt. Somit weiß er wie viele Threads existieren und welchen Workload diese beinhalten.

Nun kommen wir zu der eigentlich spannenden Frage. Gegeben sei eine Menge von Threads, welche sich ständig updatet, bzw. welche ständig wächst, wie können wir nun in möglichst kurzer Zeit eine möglichst sinnvolle Aufteilung dieser Threads auf unsere Recheneinheiten finden. Da es enorm schwierig sein kann, die bestmögliche Antwort auf diese Frage zu finden verwendet man dafür häufig einen Greedy Scheduler. Diesen wollen wir hier etwas genauer betrachten. Ein Greedy Algorithmus ist im Allgemeinen ein Algorithmus der basierend auf dem aktuellen Zustand die Entscheidung trifft, die ihm dem Ziel am nächsten bringt, ohne dabei den weiteren Verlauf der Berechnung zu betrachten. Auf unser Problem bezogen bedeutet dies, dass der Scheduler so viele Threads auf die Recheneinheiten verteilt wie möglich, hat man also 4 Kerne mit jeweils einem Thread so kann der Scheduler zu jedem Zeitschritt maximal 4 Threads verteilen und macht dies auch. Ein solcher Zeitschritt, bei dem jeder verfügbare Thread eine Aufgabe erhält, wird als vollständiger Schritt bezeichnet. Ist ein Thread gerade belegt oder hat der Scheduler nicht genügend Threads zu verteilen, so teilt er so viele zu wie möglich. (unvollständiger Schritt) Durch dieses Vorgehen erreicht der Scheduler zwar nicht immer die bestmögliche Ausführungsreihenfolge, jedoch erhält man durch dieses Verfahren eine Variante die mindestens halb so gut wie die Bestmögliche ist.

Um dies zu zeigen müssen wir zunächst zeigen, dass folgendes Lemma, für einen Greedy Scheduler, gilt.

Lemma 3.6.1 ($T_p \leq \frac{T_1}{P} + T_\infty$)**.** *Beim Verwenden eines Greedy Schedulers ist die Zeit, die auf P Prozessoren benötigt wird, kleiner oder gleich der Zeitspanne addiert mit der minimal auf P Prozessoren benötigten Zeit.*

Dafür möchte ich zunächst eine Beweisidee geben. Der vollständige Beweis ist relativ lang und kann im Referenzwerk unter 27.1 nachgelesen werden.

Beweisidee: Die Anzahl der vollständigen Schritte muss kleiner oder gleich $\lfloor \frac{T_1}{P} \rfloor$ sein, da sonst die Prozessoren mehr Arbeit erledigen würden, als gesamt zur Verfügung steht. Betrachten wir beispielsweise 4 Recheneinheiten und eine Arbeit von $T_1 = 20$. Hätte man nun mehr als $\frac{20}{4}$ vollständige Schritte so würde man offensichtlich mehr als 20 Arbeitsschritte verteilen, da gilt $n \cdot 4$ mit $n > \frac{20}{4}$ ist größer als 20. Die Anzahl der unvollständigen Schritte hingegen muss kleiner oder gleich T_∞ sein, da jeder unvollständige Schritt definitiv eine Berechnung aus unserer längsten Folge voneinander abhängiger Berechnungen (als welche wir T_∞ laut unserer Definition auch verstehen können) ausführt. Dies liegt daran, dass ein unvollständiger Schritt nur dann gemacht wird, wenn nicht mehr genügend voneinander unabhängige Berechnungen zur Verfügung stehen. Führt man die Betrachtungen der vollständigen und unvollständigen Schritte, welche offensichtlich die Gesamtheit unserer Schritte ausmachen, zusammen und stellt zunächst fest, dass unsere Zeit auf P Prozessoren T_P gleich der Zahl der vollständigen Schritte addiert mit den unvollständigen Schritten sein muss. Dann ergibt sich, da wir sowohl für die vollständigen Schritte, als auch für die unvollständigen Schritte eine obere Schranke angegeben haben, dass $T_P \leq \frac{T_1}{P} + T_\infty$ gilt.

Mit Lemma 3.6.1 können wir nun unsere Aussage, dass ein Greedy Scheduler immer eine Variante findet, welche mindestens halb so gut, wie die Bestmögliche, ist, beweisen.

Beweis.

$$T_p \leq \frac{T_1}{P} + T_\infty \tag{1}$$

$$\leq 2 \cdot max(\frac{T_1}{P}, T_\infty) \tag{2}$$

$$\leq 2 \cdot T_P^* \tag{3}$$

□

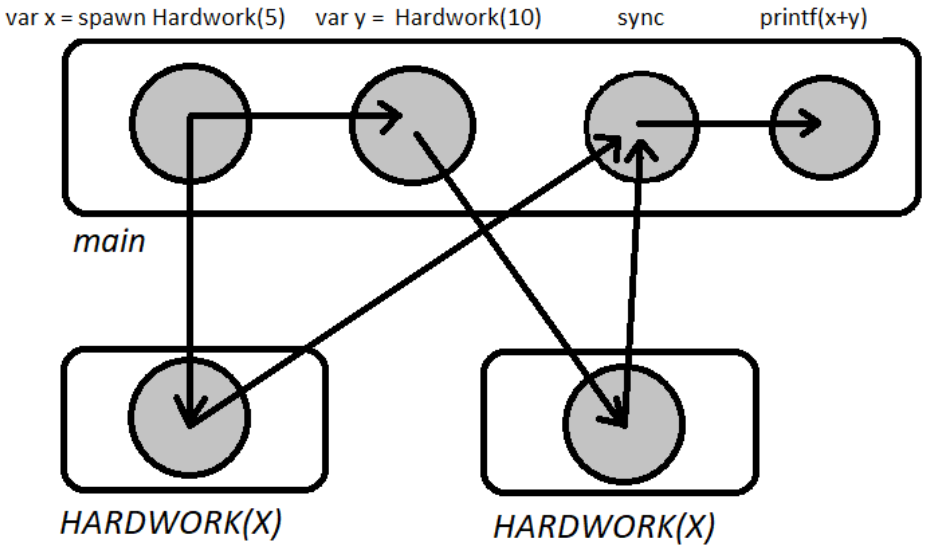
T_P^* ist hierbei die optimale Laufzeit die von einem beliebigen Scheduler erreicht werden kann. Aus Gleichung (1) folgt in (2), dass das doppelte des Maximums aus den beiden addierten Werten ebenfalls größer oder gleich der Zeit auf P Prozessoren sein muss. Und da zusätzlich leicht erkennbar ist, dass T_P^* niemals kleiner als die Zeitspanne T_∞ oder kleiner als die minimal auf P Prozessoren benötigte Zeit $\frac{T_1}{P}$ sein kann gilt somit auch $T_p \leq 2 \cdot T_P^*$ (3), was zu beweisen war.

So haben wir mit unserem Greedy Scheduler nun eine Variante zum Scheduling beschrieben, welche sehr effizient ist und zudem auch im schlechtesten Fall eine vertretbare Lösung für unser Problem findet. Achtet man beim Programmieren darauf, dass die

Parallelität deutlich größer ist als die Anzahl der Prozessoren so findet man mit dem Greedy Scheduler sogar zuverlässig weitaus schnellere Lösungen als den worst case. Dies liegt daran, dass wir bei hoher Parallelität immer viele Threads zur Verfügung haben und der Scheduler in den meisten Schritten genug Threads hat, um alle Recheneinheiten mit Arbeit zu versorgen. (Optimale Beschleunigung also die Zeit $\frac{T_1}{P}$ erreicht man dann, wenn in jedem Schritt immer alle Recheneinheiten mit Aufgaben versorgt werden können)

3.7 Modell

Zur genaueren Analyse eines bestimmten Algorithmus kann man diesen als gerichteten azyklischen Berechnungsgraphen darstellen. Hierbei beschreiben die Knoten Anweisungen und die Kanten Abhängigkeiten zwischen diesen, wobei eine Kante von U nach V bedeutet, dass u vor v ausgeführt werden muss. Eine Funktion mit verschiedenen Anweisungen kann man darstellen, indem man um eine Gruppe von Knoten einen Kasten zieht. Zusätzlich gilt, dass man Berechnungsfäden ohne Kontrollanweisungen (also ohne sync, spawn oder parallel) zu einem Knoten zusammenfassen kann. Der längste Pfad vom Startknoten zum Zielknoten in unserem Graphen entspricht genau der Zeitspanne T_∞, da es die längste Abfolge von voneinander abhängigen Anweisungen ist, die in unserem Algorithmus auftritt. Ablesen kann man die Zeitspanne jedoch nur, wenn jeder Knoten die gleiche Arbeit leistet, also man nicht in einigen Knoten mehrere Arbeitsschritte zusammengefasst hat und andere Knoten nur aus einem Arbeitsschritt bestehen. (ein Arbeitsschritt dauert hier genau eine Zeiteinheit) Start und Zielknoten lassen sich relativ leicht bestimmten. Um den Startknoten zu finden muss man lediglich den Knoten finden, welcher keine eingehende Kante hat. Der Zielknoten hingegen ist der, welcher keine ausgehende Kante hat. Aus der Struktur unseres Graphen kann man dann relativ leicht die Kontrollanweisungen ablesen, wobei beispielsweise ein Knoten, auf den mehrere Kanten zeigen ein sync Befehl bzw. ein Return Befehl sein muss und ein Knoten von dem mehrere Kanten abgehen ein spawn Befehl bzw. parallel Befehl sein muss. Um aus unserem Graphen Parallelität abzulesen, kann man einfach Prüfen ob es zwischen zwei Knoten einen gerichteten Pfad gibt also ob man von Knoten 1 aus irgendwie Knoten 2 erreicht oder andersherum, ist dies der Fall so muss der Knoten, von dem aus man den anderen erreicht, offensichtlich vor diesem Ausgeführt werden und die beiden Knoten sind nicht parallel ausführbar. Hat man hingegen Knoten die sich nicht verbinden lassen dann kann man diese bedenkenlos parallel ausführen, da zwischen ihnen keine Abhängigkeit besteht. Den Berechnungsgraphen für die Funktion $HARDWORK(x)$ aus Absatz 3.3 könnte man wie folgt darstellen.



Genaueres zur Modellierung von Algorithmen mithilfe von gerichteten azyklischen Berechnungsgraphen kann im Referenzwerk unter 27.1 nachgelesen werden, des weiteren wird das Vorgehen dort noch einmal an einem sehr viel umfangreicheren Beispiel erklärt.

4 Ergebnisse

4.1 Einführung

Im Folgenden werden wir die gelernten Grundlagen nun einsetzen um 3 verschiedene Arten von Matrixmultiplikation zu parallelisieren und sie anschließend untereinander und mit ihrer seriellen Version vergleichen. Ich habe mich an dieser Stelle für die Matrixmultiplikation entschieden, da sie für zahlreiche Anwendungen eine enorm wichtige Rolle spielt und sogar so gut parallelisierbar ist, dass man sie auf hochparallele Grafikkarten (GPU's) auslagert anstatt sie ausschließlich mit CPU's zu berechnen.

Diese GPU's lassen sich, im Falle von NVIDIA GPU's, mit der eigenen Concurrency Plattform CUDA Programmieren und können bei hochparallelen Anwendungen die Geschwindigkeit eines CPU's bei weitem übersteigen. Dies wird dadurch ermöglicht, dass die GPU aus einem Vielfachen an Recheneinheiten einer CPU besteht. Nun stellt sich die Frage warum eine GPU nicht einfach eine CPU mit vielen Kernen ist, hierbei liegt die Antwort in der Art der Kerne, CPU-Kerne sind für alle möglichen Aufgabengebiete ausgelegt und haben beispielsweise umfangreiche Branchprediction, GPU Kerne hingegen sind eher minimalistisch aufgebaut und können nur sehr einfache Rechnungen ausführen.

Die Grobe Funktionsweise besteht hier also darin, dass die CPU Berechnungen die sehr gut parallelisierbar sind so vorbereitet, dass die GPU diese nur noch ausführen muss um die Werte anschließend an den CPU zurückzugeben.

4.2 Theorie

Betrachten wir nun die Matrixmultiplikation. Dazu starten wir mit dem ersten unserer 3 Möglichen Algorithmen.

4.2.1 geschachtelte Schleife

Dieser Algorithmus ist an das Vorgehen angelehnt, welches die meisten von uns in der Schule gelernt haben. Hierbei betrachtet man jedes Feld der Zielmatrix einzeln für sich und berechnet es indem man dem folgenden Algorithmus anwendet.

```
Eingabe → nxn Matrizen A, B;
n = A.zeilen;
C = neue nxn Matrix;
for i = 1 to n do
    for j = 1 to n do
        C_{i,j} = 0;
        for k = 1 to n do
            C_{i,j} = C_{i,j} + A_{i,k} · B_{k,j};
        end for
    end for
end for
return C;
```

$\Longrightarrow$

```
Eingabe → nxn Matrizen A, B;
n = A.zeilen;
C = neue nxn Matrix;
parallel for i = 1 to n do
    parallel for j = 1 to n do
        C_{i,j} = 0;
        for new k = 1 to n do
            C_{i,j} = C_{i,j} + A_{i,k} · B_{k,j};
        end for
    end for
end for
return C;
```

Wie im vorangegangenen Satz erwähnt wird hier jede Zeile unserer Zielmatrix für sich, also unabhängig von allen anderen Zellen berechnet. Wie wir bereits gelernt haben bedeutet das für uns, dass wir hier die Möglichkeit haben unsere Berechnung zu parallelisieren. Dies geschieht wie oben gezeigt, indem man das Schlüsselwort "parallel" so in den Code einfügt, dass jedes Feld der Zielmatrix, also jede Ausführung der innersten For-Loop, parallel zu allen anderen ausgeführt wird. Zusätzlich benötigen wir hier noch ein bisher noch nicht eingeführtes Schlüsselwort. Das Schlüsselwort "new" wird hier verwendet um zu kennzeichnen, dass wir für jeden erstellten Thread eine eigenen Variable für k benötigen und diese nicht unter den Threads teilen möchten. Würden wir dieses Schlüsselwort hier weglassen würde sich jeder unserer Threads immer in der selben Schleifeniteration befinden wie alle anderen, auch wenn die Threads nicht gleichzeitig gestartet wurden. Dies liegt daran, dass die Threads dann alle auf denselben Speicherort zugreifen, um herauszufinden in welcher Iteration sie sich befinden.

Kommen wir nun zu Quantifizierung unseres Algorithmus mithilfe der eingeführten Metriken.

Arbeit: $T_1 = n^3$

Die Arbeit ergibt sich hier aus den einzelnen Berechnungen, die ausgeführt werden müssen, zunächst einmal haben wir n^2 Werte in der Zielmatrix zu berechnen für jeden dieser Werte brauchen wir jeweils n Rechenoperationen, so kommen wir also auf eine Gesamtkomplexität von n^3

Zeitspanne: $T_\infty = n$

Die Zeitspanne liegt hierbei bei n da wir, wie bereits erwähnt, jeden der n^2 Einträge der Zielmatrix parallel berechnen können und eine solche Berechnung n Schritte braucht.

Parallelität: $\frac{T_1}{T_\infty} = \frac{n^3}{n} = n^2$

Aus dem Wert unserer Parallelität folgt in diesem Fall, dass wir unsere maximale Beschleunigung erreichen, wenn wir n^2 Recheneinheiten zur Verfügung haben. Also genau dann, wenn für jede Zelle der Zielmatrix genau ein Prozessor zur Verfügung steht.

Mithilfe dieser Metriken und den Gesetzen bzgl. Laufzeit, Spielraum und Beschleunigungsfaktor können wir nun abschätzen wie sich unser Algorithmus auf verschiedenen Prozessoren verhalten würde. Einige Beispiel und Vergleiche mit der Realität finden sich im Abschnitt Praxisbeispiel 4.3.

4.2.2 Divide and Conquer

Bei dieser Art der Matrix Multiplikation zerlegt man die gegebenen nxn Matrizen (A, B) in jeweils 4 gleichgroße Teilmatrizen.

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \text{ x } \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Diese zerlegt man wiederum ebenfalls in 4 Teile, dies führt man solange rekursiv fort bis man nur noch 1x1 Matrizen für A_{xy} und B_{ij} hat. An diesem Punkt stoppt die Rekursion, da die Teilmatrizen sich nicht weiter zerlegen lassen. (man keine weite Matrixmultiplikation mehr ausführen muss um das Zwischenergebnis zu berechnen) Man erhält für C_{vw} einen Wert durch das einfache Multiplizieren und addieren der entsprechenden Werte A_{xy} und B_{ij}, nach folgender Rechenvorschrift.

$$\begin{aligned} C_{11} &= A_{11} \times B_{11} + A_{12} \times B_{21} \\ C_{12} &= A_{11} \times B_{12} + A_{12} \times B_{22} \\ C_{21} &= A_{21} \times B_{11} + A_{22} \times B_{21} \\ C_{22} &= A_{21} \times B_{12} + A_{22} \times B_{22} \end{aligned}$$

Von hier an müssen nur noch die Ergebnisse in rekursiv aufgerufenen Gleichungen eingesetzt werden, bis man zur Wurzel gelangt und das fertige Ergebnis erhält.

Alternativ kann man die Addition erst am Ende durchführen also:

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1}B_{1,1} & A_{1,1}B_{12} \\ A_{2,1}B_{1,1} & A_{2,1}B_{12} \end{bmatrix} + \begin{bmatrix} A_{1,2}B_{2,1} & A_{1,2}B_{22} \\ A_{2,2}B_{2,1} & A_{2,2}B_{22} \end{bmatrix}$$

das Vorgehen zur Rekursion ist hierbei analog.

Das beschriebene Vorgehen lässt sich nun folgendermaßen als Pseudocodealgorithmus umsetzen

$DivideAndConquer(A, B)$

Eingabe $\rightarrow nxn$ Matrizen A, B;
$n = A.zeilen$;
if $n == 1$ **then**
    $C_{11} = A_{11} \cdot B_{11}$
else
    Teile die Matrizen A, B wie oben veranschaulicht in jeweils 4 gleich große Teilmatrizen $(A_{11}, A_{12}, ..)$

    spawn $T_{11} = DivideAndConquer(A_{11}, B_{11})$
    spawn $T_{12} = DivideAndConquer(A_{11}, B_{12})$
    spawn $T_{21} = DivideAndConquer(A_{21}, B_{11})$
    spawn $T_{22} = DivideAndConquer(A_{21}, B_{12})$
    spawn $R_{11} = DivideAndConquer(A_{12}, B_{21})$
    spawn $R_{12} = DivideAndConquer(A_{12}, B_{22})$
    spawn $R_{21} = DivideAndConquer(A_{22}, B_{21})$
    $R_{22} = DivideAndConquer(A_{22}, B_{22})$
    sync

    return T + R
end if

Wir haben hier die Idee des Divide and Conquer Algorithmus umgesetzt indem wir zunächst mit einem If-Statement den Basisfall abgedeckt haben. Tritt der Basisfall nicht ein so gehen wir in unseren Else-Zweig, welcher die Funktionsweise der oben beschriebenen Rekursion umsetzt indem er die Matrizen in vier Teile teilt und dann die nötigen Berechnungen rekursiv und parallel als neue Divide and Conquer Matrix Multiplikationen aufruft. Sind alle dieser Aufrufe zu einem Ergebnis gekommen (sync Befehl),

so können wir die beiden entstandenen nxn Matrizen T und R zu unserem Gesamtergebnis zusammenaddieren und ausgeben. Für die Addition habe ich hier keine weitere Funktion angegeben, da sie im Vergleich mit den Multiplikationen eine sehr geringe Komplexität hat. Man könnte die Addition zweier Matrizen jedoch durch eine doppelte Schleife umsetzen. Wobei beide dieser Schleifen parallele Schleifen sein können, man kann also gleichzeitig alle Felder der Matrix betrachten und mit ihrem entsprechenden Gegenstück in der zu addierenden Matrix addieren.

Auch diesen Algorithmus möchten wir nun Quantifizieren.

Arbeit: $T_1 = n^3$

Die Arbeit ergibt sich auch hier aus den einzelnen Berechnungen die ausgeführt werden müssen. Da ein ausführlicher Beweis der Komplexität hier zu weit führen würde gebe ich im Folgenden eine Beweisidee.

Beweisidee: Stellen wir zunächst die Vermutung auf, dass unsere Arbeit gleich der Arbeit des drei Schleifen Algorithmus, also n^3 ist. Wie wir aus unseren Rechenvorschriften sehen können brauchen wir für den Basisfall, bei dem sich die Matrizen A und B nicht noch weiter zerlegen lassen, genau 8 Multiplikationen und eine Addition. Für die Komplexitätsberechnung ist diese Addition jedoch nicht relevant, somit haben wir für den genannten Basisfall bei dem wir zwei 2x2 Matrizen multiplizieren eine Komplexität von n^3, was unserer Vermutung entspricht. Der Einfachheit halber fahren wir hier mit der doppelten Matrix Größe also 4x4 Matrizen fort, wir sollten also auf 64 Multiplikationen kommen. Dies ist tatsächlich der Fall, da wir für die Berechnung unserer 4x4 Zielmatrix, laut Rechenvorschrift acht Multiplikationen von 2x2 Matrizen durchführen müssen, welche wie im Schritt 1 gezeigt jeweils 8 Multiplikationen für ihre Berechnung benötigen. Also haben $8 \cdot 8$ Multiplikationen was unserer Erwartung entspricht. Dieses vorgehen kann man nun induktiv weiterführen und so zeigen, dass unsere Komplexität für alle n^2 Matrizen gilt.

Um zu zeigen, dass die Komplexität von n^3 für alle Matrizen Größen gilt kann man einen Induktionsbeweis führen. Dafür muss man sich zunächst vor Auge führen, wie die Divide and Conquer Matrixmultiplikation für Matrizen ungrader Größe Funktioniert, was hier jedoch zu weit führt.

Zeitspanne: $T_\infty = ld^2(n)$

Der konkrete Beweis hierfür führt ebenfalls zu weit, daher folgt eine kurze Beweisidee.

Beweisidee: Zunächst halten wir fest, dass das Zerlegen der Matrizen in vier gleichgroße Teile, wie es in unserem Algorithmus zu Beginn jedes Else-Zweig Durchlaufs durchgeführt wird, in von der Eingabe unabhängiger Zeit möglich ist. Daher wird dies durch die benötigte Zeitspanne der Addition der berechneten Teilmatrizen T und R, am Ende unserers Else-Statements, dominiert. Die Addition benötigt eine Zeit von $ld(n)$. Zu der

Zeit, welche wir für die Addition benötigen, kommt zusätzlich noch die Zeit, die für die Ausführung der 8 rekursiven Aufrufe unserer Funktion benötigt wird. Da alle unsere 8 Funktionsaufrufe parallel ausgeführt werden können, ist diese Zeit genau die Zeit, die ein einzelner Funktionsaufruf benötigt. Aufgrund dessen, dass dieser Funktionsaufruf mit zwei Matrizen halber Ordnung, als Parameter, geschieht, kommen wir zu folgender Rekursionsgleichung $DivideAndConquer_\infty(n) = DivideAndConquer_\infty(\frac{n}{2}) + ld(n)$. Löst man diese Rekursionsgleichung dann auf, so erhält man $ld^2(n)$.

Parallelität: $\frac{T_1}{T_\infty} = \frac{n^3}{ld^2(n)}$

Dadurch, dass wir die gleiche Arbeit wie in unserem dreifach Schleifen Algorithmus haben, die Zeitspanne jedoch (bei Matrix Größen größer 16x16) geringer ist erhalten wir im Allgemeinen eine höhere Parallelität.

Auch hier können wir wieder unsere gelernten Metriken und Gesetze anwenden und basierend darauf Abschätzungen für unser Zielsystem treffen. Der wohl wichtigste Schluss ist in diesem Fall, dass durch unsere höhere Parallelität, welche wir ab einer Matrixgröße von nxn mit $n > 16$ erreichen (da ab dieser Größe $n^2 > ld^2(n)$), bei entsprechenden Matrizen mehr Prozessoren mit Arbeit versorgt werden können als in unserem dreifach Schleifen Algorithmus. Das Bedeutet, dass wir beispielsweise bei einer 20x20 Matrixmultiplikation mit unserem ersten Algorithmus 400 ($\frac{20^3}{20}$) Recheneinheiten sinnvoll mit Arbeit versorgen könnten und mit unserem Divide and Conquer Algorithmus etwa 428($\frac{20^3}{ld^2(20)}$). Das bedeutet in der Praxis, dass unser zweiter Algorithmus auf System mit sehr vielen Recheneinheiten eine schnellere Laufzeit haben wird als unser erster Algorithmus. Da eine so große Anzahl an CPU-Kernen vermutlich in der Praxis eher selten zu finden ist und die Parallelen Berechnungen hier wieder sehr einfach sind, wird man den Performancevorteil des Divide and Conquer Algorithmus im Vergleich mit der dreifachen Schleife vermutlich erst bei der Verwendung von Grafikkarten feststellen.

4.2.3 Strassen Matrix Multiplikation

Die Matrix Multiplikation nach Strassen wollen wir hier nur kurz anreißen, da eine genauere Betrachtung auch in diesem Fall wieder zu weit führen würde. Die Idee der Matrix Multiplikation nach Strassen ist es die Arbeit zu Verringern. Dies schafft man indem man ähnlich vorgeht wie beim Divide and Conquer Algorithmus also die zu Multiplizierenden Matrizen zunächst solange vierteilt, bis man den Basisfall erreicht. An dieser Stelle setzt Strassen an und multipliziert die beiden 2x2 Matrizen mit 7 statt 8 Multiplikationen, was sich bei sehr großen Matrizen deutlich im Wert der Arbeit widerspiegelt. Wie die Strassen Matrix Multiplikation 2x2 Matrizen in 7 statt 8 Schritten multipliziert kann beispielsweise im Referenzwerk im Abschnitt 4.2 nachgelesen werden. Dort findet man auch die Berechnung zur Komplexität, also zur Arbeit des Algorithmus, und weitere Informationen. Im Folgenden gebe ich die Werte für unsere drei Metriken nur an um sie anschließend mit unseren anderen beiden Algorithmen vergleichen zu können.

Arbeit: $T_1 = n^{ld(7)}$

Zeitspanne: $T_\infty = ld^2(n)$

Parallelität: $\frac{T_1}{T_\infty} = \frac{n^{ld(7)}}{ld^2(n)}$

Durch das Verwenden des Strassen Algorithmus haben wir es geschafft unsere Arbeit zu reduzieren, was in der Praxis bedeutet, dass wir insgesamt weniger Berechnungen ausführen müssen als mit den vorangegangenen Algorithmen. Auch die Zeitspanne ist genau so gering wie beim Divide and Conquer Algorithmus, was den Strassen Algorithmus gemessen an unseren drei Metriken zum theoretisch schnellsten Algorithmus macht.

4.3 Praxisbespiel

Um unsere Vermutungen, welche basierend auf den Metriken und Gesetzen aufgestellt wurden testen zu können und um aufzuzeigen, wie sich dabei gemachte Vereinfachungen auf das tatsächliche Ergebnis in Form von Laufzeitunterschieden auswirken habe ich für diesen Teil der Ausarbeitung ein kleines Python3 Programm geschrieben. In diesem Programm kann man die Matrixmultiplikation mit dem dreifach Schleifen Algorithmus, auf verschiedenen Anzahlen von Prozessorkernen testen, sofern man auf seinem System mehrere Kerne zur Verfügung hat.

Das Programm kann in der Kommandozeile gestartet werden, wobei es keine Startparameter hat. Alle nötigen Eingaben zum Testen auf dem eigenen System werden im Programm erklärt. Zur Parallelisierung habe ich die Python Bibliothek multiprocessing verwendet, diese Bibliothek können wir als die verwendete Concurrency Plattform sehen. Mit dieser Bibliothek hatte ich ähnlich wie bei der beispielhaft in Absatz 3.3 eingeführten Concurrency Plattform die Möglichkeit alle Aufgaben, bis auf das spezifizieren der parallel auszuführenden Berechnungen, der zugrunde liegenden Plattform, also in diesem Falle der multiprocessing Bibliothek, zu überlassen. Die genauere Verwendungsweise und weitere interessante Informationen zum Python3 multiprocessing können in der entsprechenden Dokumentation auf der Python3-docs Website gefunden werden.

Link zum Programm: github.com/richard-riedel/Parallele-Algorithmen

Kommen wir nun zu den Erkenntnissen, welche ich aus einigen Tests auf meinem System (Intel i7 4770) gewinnen konnte. Zunächst einmal habe ich festgestellt, dass die serielle Implementierung der dreifachen Schleife wie in 4.2.1 dargestellt, deutlich langsamer ist als das Ausführen des, mithilfe der Python Bibliothek parallelisierten, Algorithmus. Dies liegt daran, dass die multiprocessing Bibliothek, noch einige Optimierungen während der Ausführung, der Berechnung durchführt und nicht, wie der serielle Algorithmus einfach n^3 Schleifendurchläufe hintereinander ausführt. Auffällig hierbei ist jedoch, dass bei sehr kleinen Matrizen, beispielsweise der Größe $n = 50$, der serielle Algorithmus schneller ist als der parallele auf einem Kern. Grund dafür ist der von der Concurrency Plattform erzeugte Overhead, welcher bei einer einfachen Schleifenverschachtelung wegfällt. Dies

zeigt, dass wir auch bei gleicher Arbeit und gleicher Prozessorzahl mit verschiedenen Vorgehensweisen verschiedene Ergebnisse erreichen können und eine so einfache Betrachtung wie wir sie zunächst vorgenommen haben nicht ausreicht, um eine korrekte Erwartung für die Laufzeit auf einem bestimmten System formulieren zu können.

Diese Erkenntniss zeigt sich auch bei Betrachtung der Beschleunigung. Um diese Betrachtung durchführen zu können müssen wir jedoch zunächst die Parallelität betrachten. Da ich in allen meinen Versuchen mit sehr großen Matrizen gearbeitet habe ($50 \leq n \leq 1000$) erhalten wir bei allen Versuchen eine Parallelität, die die Prozessorzahl bei weitem übersteigt (schon bei 50x50 Matrizen erreichen wir eine Parallelität von $\frac{50^3}{50} = 2500$). Daher liegt die Vermutung nahe, dass wir eine perfekte Beschleunigung erreichen. Die praktischen Tests haben jedoch gezeigt, dass dies nicht der Fall ist, zwar reduzieren sich die Laufzeiten bei einer Erhöhung der Prozessorzahl, vor allem bei großen Matrizen, wie zu erwarten deutlich, jedoch eben nicht linear. Gründe dafür müssen wir wieder in von uns bisher nicht betrachteten Bereichen suchen. Beispielsweise kann das temporäre übertackten einzelner Kerne, was von modernen CPU's vor allem im mobilen Bereich heutzutage standartmäßig eingesetzt wird, dazu führen, dass unser CPU all seine Kerne nacheinander durchwechselt und solang übertacktet bis sie zu heiß werden. Diese Möglichkeit hat der CPU bei einer parallelen Ausführung auf allen Kernen natürlich nicht, wodurch wir beispielsweise die erwartete Verdopplung beim rechnen auf 8 statt 4 Kernen nicht erreichen. Ein anderer Grund für unseren geringeren Beschleunigungsfaktor kann sein, dass für eine Ausführung auf mehreren CPU's natürlich auch mehr Kommunikations-, Scheduling- und Speicherzugriffs-Overhead entsteht.

Anzumerken ist auch, dass sich im Systemmonitor deutlich beobachten lässt, wie das Scheduling des Betriebssystems den Python Task während der Laufzeit des Programms zwischen den einzelnen CPU-Kernen hin und her schiebt um die Last besser auf die gesamte CPU zu verteilen. Dies ist eine Aufgabe die man bei der statischen Programmierung von Threads selbst übernehmen müsste. (Verwendet man alle zur Verfügung stehenden Kerne tritt dieser Effekt offensichtlich nicht auf)

5 Zusammenfassung

Zu Beginn der Arbeit wurde kurz erläutert, warum die parallele Programmierung immer mehr an Relevanz für wissenschaftliches Arbeiten und industrielle Anwendungen gewinnt. Festzuhalten gilt hier, dass die Rechengeschwindigkeit die, beim Verwenden serieller Programme, zur Verfügung steht sich seit etwa 2005 nur noch langsam erhöht und man deshalb bei vielen aktuellen Anwendungen mit parallelen Algorithmen arbeiten muss. Im Anschluss haben wir uns die Frage gestellt, wie wir diese Parallelisierung unserer Algorithmen umsetzen können. Dabei haben wir zahlreiche Schwierigkeiten und Probleme aufgezeigt, welche es zu lösen gilt. Da die Lösung dieser Probleme nun für viele Anwendungsgebiete ähnlich ist, wurden zahlreiche Concurrency Plattformen entwickelt, um uns die Arbeit zu erleichtern. Genauer haben wir anschließend die Klasse

des dynamic Multithreading betrachtet, welche sich dadurch auszeichnet, dass man hier einen Algorithmus parallelisieren kann, indem man, in bereits bestehenden sequenziellen Code, einige einfache Schlüsselwörter einfügt. Um beurteilen zu können wie viel Beschleunigung erreicht wurde oder wie viel Potential für Beschleunigung noch offen ist, haben wir anschließend einige Performancemaße eingeführt und ein Modell zu Darstellung der parallelen Algorithmen gegeben. Um die "Magie" der Concurrency Plattformen nicht unberührt zulassen, haben wir uns in einem Abschnitt noch mit einem sehr wichtigen Bestandteil dieser beschäftigt, dem Scheduling. Hierzu wurde kurz das Wesen von Greedy Algorithmen beschrieben um anschließend einen solchen Algorithmus auf unser Problem anzuwenden, dadurch konnten wir eine Problemlösung finden die selbst im schlimmsten Fall immer noch halb so gut, wie die ideale Lösung, ist. Um alle aus den Grundlagen gewonnenen Erkenntnisse umzusetzen haben wir im Teil 4 der Ausarbeitung drei verschiedene Arten der Matrix Multiplikation parallelisiert und verglichen. Zusätzlich wurde noch ein Python Codebeispiel gegeben, welches die theoretisch gewonnenen Erkenntnisse über die Matrix Multiplikationen noch einmal praktisch beweist, veranschaulicht und schwächen unserer vereinfachten Betrachtung aufzeigt. Was den tatsächlichen Umfang des Themas, welcher unsere kurze Betrachtung bei weitem übersteigt, noch einmal verdeutlicht.

6 Quellen

- Referenzwerk: Deutsche Übersetzung von Prof. Dr. rer. nat. habil. Paul Molitor (2013) des Werks: Prof. Dr. Thomas H. Cormen, Prof. Dr. Charles E. Leiserson, Prof. Dr. Ronald Rivest, Prof. Dr. Clifford Stein (2009) Introduction to Algorithms, Third Edition. Cambridge, Massachusetts/ London, England: The MIT Press

- John L. Hennessy, David A. Patterson (September 2011) Computer Architecture, Fifth Edition: A Quantitative Approach.