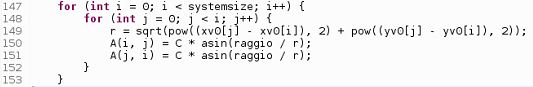
Zur Interpretation der Ergebnisse müssen zuallererst Störfaktoren für die Auswertung und dessen allgemeiner Ablauf zusammengefasst werden. Hierbei handelt es sich wahrscheinlich primär um zwei Gruppen:

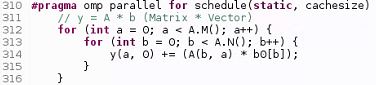
1. Zufällige Zugriffe des Betriebssystems auf Prozessor oder Arbeitsspeicher – dies wäre nur vermeidbar bei einer Ausführung außerhalb eines Betriebssystems, welches im Rahmen der Studienarbeit soweit nicht möglich ist. Die Schwankungen, welche hierdurch auftreten, sollten klein, jedoch nicht vernachlässigbar sein.
2. Zufällige Zugriffe von Benutzeranwendungen auf Prozessor oder Arbeitsspeicher – da auf dem Studiencomputer lediglich TeamViewer parallel aktiv ist und dies einen gleichmäßigen Einfluss auf die Ergebnisse haben, wodurch dies vernachlässigbar sein sollte.

Aus diesen Störfaktoren lässt sich schließen, dass für eine genauere Interpretation der Ergebnisse mit verhältnismäßig hohem Rechenaufwand geeigneter sind, da die Abweichungen sich dort prozentual gesehen weniger bemerkbar machen als Ergebnisse mit wenig Rechenaufwand. Dies ist unter anderem auch der Grund, warum die minimale Laufzeit aus mindestens 100 Werten generiert wurde. Hierdurch wird der Effekt der oben aufgeführten Einflussfaktoren reduziert.

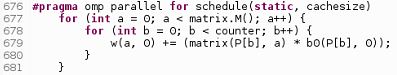
Zu unterscheiden ist zwischen drei Aufgaben, aus welchen Daten zur Visualisierung generiert wurden. „Times1“ stellt die Aufgabe dar, auf der Hauptdiagonalen einer Matrix die Elemente mit Werten zu befüllen. In „Times2“ wird ein Matrix-Vektor-Produkt einer direkten Adressierung gelöst, während bei „Times3“ die Ausführungszeit der indirekten Adressierung in einem Matrix-Vektor-Produkt gelöst wird.



Code für Times1; Auszug aus OMP\_Test, Release vom 21.08.2020



Code für Times2; Auszug aus OMP\_Test, Release vom 21.08.2020



Code für Times3; Auszug aus OMP\_Test, Release vom 21.08.2020

Als allererstes ist zu erkennen, dass bei den Datensätzen „Times1“ und „Times2“ sowohl der Anteil „dynamic“ und „guided“ länger benötigen als „static“ – eine mögliche Erklärung hierbei liegt in der Art wie diese „schedules“ funktionieren. Während bei „static“ davon ausgegangen wird, dass jeder Schleifendurchlauf ungefähr gleich viel Zeit beansprucht und dementsprechend wenig Overhead reserviert wird, ist dies bei „dynamic“ und „guided“ nicht der Fall ist. Da der Overhead bei dieser Aufgabe nicht benötigt wird ist die statische Herangehensweise bei dieser Aufgabe somit klar die effizienteste Variante. Ein erwartetes Ergebnis, da „static-schedule“ für Codeabschnitte mit identischen Iterationen ist, welches hier der Fall ist. „guided“ als leichte Abschwächung von „static“ mit ein wenig Overhead liegt in diesem Fall – wie erwartet – kurz hinter den Ausführungszeiten von „static“. „Dynamic“ liegt am Ende der Messdaten, da „dynamic“ auf nicht identische Iterationen ausgelegt ist und dieser Typ besonders viel Overhead reserviert, welches hier ineffizient ist und dementsprechend das Programm verlangsamt.

Auszug der Messdaten für sup6 zu den einzelnen Problemarten und deren Parallelisierungslaufzeiten; generiert aus BEM bis zum 26.08.2020.

Bei den Visualisierungen der Daten zu wachsenden Problemgrößen ist zu erkennen, dass bei „static“ und „guided“, der Effekt einer Erhöhung der Threadanzahl bedeutend stärker ausfällt als die Vergrößerung des Chunks, während bei „dynamic“ die Vergrößerung des Chunks eine stärkere Rolle spielt als bei den anderen beiden „schedules“. Allgemein ist einfach zu verstehen, dass wenn mehr Aufgaben gleichzeitig abgearbeitet werden können (Hervorgerufen durch eine Erhöhung der Anzahl der Threads) das Programm bedeutend schneller läuft. Eine Erhöhung des Chunks verringert die Zeit, welches das Programm versucht Daten aus dem Arbeitsspeicher abzufragen und in den Prozessor zu kopieren, was vor allem bei „schedules“, zusätzlich wird insbesondere bei „dynamic“ die Zeit, welche zur Vergabe der Aufgaben benötigt wird, verringert, wenn die Chunks größer werden. Die Begründung hierin liegt in der Arbeitsweise eines Prozessors, da dieser bereits bekannte Aufgaben parallel mit abarbeiten kann und weniger Code geladen werden muss, da die Aufgabe sich nur in kleinen Punkten unterscheidet. Hieraus ergibt sich, dass wenn bei „dynamic-schedule“ nicht alle Threads gleichzeitig auf den Arbeitsspeicher zugreifen, die Chunk-Größe eine bedeutend größere Rolle spielen, als dies bei „static“ und „guided“ der Fall ist.

Zur Bestätigung, dass eine Parallelisierung auch erfolgreich funktioniert, wurde eine Skalierungsstudie mithilfe von „strong scaling“ erstellt. Hierbei wurde die Formel verwendet, wobei die Laufzeit mit n Threads darstellt. Die Daten wurden folgendermaßen visualisiert:

Hierdurch wurde die Theorie validiert, dass die Parallelisierung tatsächlich eine Geschwindigkeitserhöhung des Codes erreicht.

Die verschiedenen Maxima der Graphen erklärt sich durch die Problemgröße der einzelnen Aufgaben. Die Aufgabentypen „times1“ und „times2“ sind nicht groß genug, um eine ideale Skalierung zu erreichen und bleiben somit unter dem Optimum. Bei „times3“ erkennt man dann einen schnellen Anstieg bis zur 6-fachen Geschwindigkeit. Dies lässt sich gut durch die Architektur des Prozessors des Computers, auf welchem diese Simulation lief, erklären: Durch Hyperthreading der Prozessoren sind effektiv 6 Prozessoren verfügbar, auf welchen bis zu 12 Threads laufen können. Da jedoch nur 6 physische Prozessoren aktiv sind, bewirkt eine Erhöhung der Threads einen Anstieg der Geschwindigkeit, die Begrenzung durch die Hardware bewirkt jedoch dass dieser nur relativ klein ist.

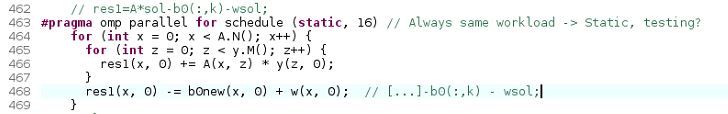
Weiterhin ist zu erkennen, dass „dynamic“ anscheinend stärker ansteigt und ein höheres Maximum hat, also die beiden anderen Typen bei „Times1“ und „Times2“. Der Grund dafür, dass „dynamic“ nicht verwendet wird ist die Tatsache, dass „dynamic“ eine bedeutend höhere Ausführungszeit hat als die anderen beiden – dementsprechend ist zwar das Verhältnis zwischen der Ausführungszeit mit wenigen Threads und mit vielen Threads höher als bei anderen Typen.

Dokumentation des Codes:

Aus den obigen Beobachtungen ergeben sich folgende Zusammenhänge für die Parallelisierung des BEM-Codes:

Allgemeine Fälle

1. Schleifen für Matrix-Vektor-/Vektor-Vektor-Produkte und einfache Zuweisungen: Hierfür wird der „scheduling-type“ „static“ gewählt, da der Rechenaufwand pro Iteration nahezu identisch bleibt. Somit wird kaum Overhead benötigt und die Situation von „Times1“ und „Times2“ lässt sich optimal übertragen.



Beispiel eines einfachen Matrix-Vektor-Produkt; Entnommen aus BEM, Release vom 12.06.2020



Beispiel einer einfachen Zuweisungsschleife; Entnommen aus BEM, Release von 12.06.2020

1. Schleifen mit indirekter Adressierung:

Hierfür wird der „scheduling-type“ „guided“ verwendet, da durch die indirekte Indexierung lediglich ein verhältnismäßig kleiner zeitlicher Unterschied zwischen den einzelnen Threads entsteht. (TODO: Bild von Zeile 265+)

1. „Reductions“ für parallele Schleifen:

Zur Verwendung von benutzerdefinierten Reduktionsdirektiven müssen diese Sonderfälle zuerst definiert werden. Hieraus resultierend mussten zwei Reduktionsdeklarationen erstellt werden (eine für Vektoren mit Ganzzahlen und eine für größere Gleichkommazahlen).



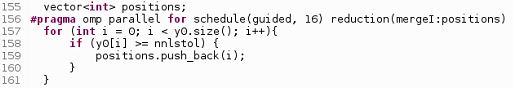
Deklaration für Vektoren mit Integer-Werten; Entnommen aus BEM, Release von 13.06.2020 (Zeilenumbruch zur besseren Lesbarkeit)



Deklaration für Vektoren mit Integer-Werten; Entnommen aus BEM, Release von 13.06.2020 (Zeilenumbruch zur besseren Lesbarkeit)

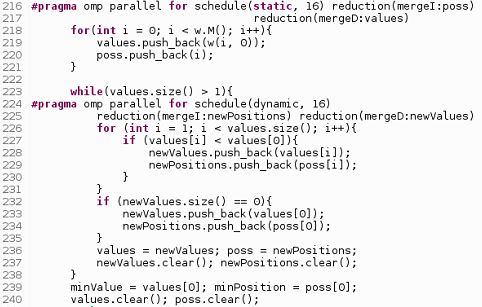
Sonderfälle

1. Find-Befehl: Für einen find-Befehl sollte der „scheduling-type“ entweder „dynamic“ oder „guided“ sein, je nach Anwendungsgebiet. In diesem Fall ergab sich nach Laufzeitmessungen „guided“. Hier vereinfachte eine „reduction“ den Code erheblich, da diese die neuen Indexwerte in der richtigen Reihenfolge aus den einzelnen Threads in den Mainthread reduziert und somit eine Menge Codezeilen und Ausführungszeit spart.



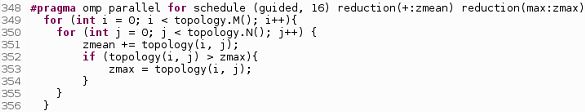
Implementierung eines Find-Befehls äquivalent zur Funktionsweise der Software MatLab; Entnommen aus BEM, Release vom 16.08.2020

1. Suche nach einem Werteminimum mit Positionsangabe: Für die Komplexität dieses Problems musste die Aufgabe in zwei Teile aufgeteilt werden: Einmal in die Befüllung der Vektoren „poss“ und „values“, welches klar einen „scheduling-type“ von „static“ bevorzugt. Weiterhin wird noch die Suche nach dem eigentlichen Werteminimum und dem Index benötigt, bei welchem sich nach Testergebnissen „dynamic“ als am effizientesten herausstellte. Hierbei kann nicht direkt eine „reduction“-Direktive verwendet werden, da es keine effiziente Methode gibt sowohl ein Maximum als auch seinen Index ausgeben zu lassen. In diesem Fall erstellt der Algorithmus zuerst eine Liste aller Werte und legt parallel eine weitere Liste der Indexierung der Werte an (beides in Form eines Vektors). Im Verlauf der Abarbeitung wird eine Schleife gestartet, bei welchem überprüft wird, ob der i-te Index des Vektors größer als der 0-te Index ist. Falls dies der Fall ist, wird der Wert samt seines Indexes in zwei verschiedene Vektoren übertragen. Am Ende der Schleife wird dann der behilfsmäßige Vektor in den eigentlichen Vektor übertragen und geleert. Dies wird solange fortgeführt, bis keine größeren Werte mehr existieren. In diesem Fall werden von beiden Vektoren die 0-ten Einträge verwendet und in globale Variablen übertragen, damit weiterer Code abgearbeitet werden kann.



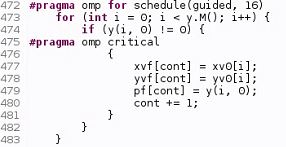
Parallele Suche eines Minimums mit gekoppelter Indexsuche des Minimums, eingefügte Zeilenumbrüche dienen der besseren Lesbarkeit und entfernt Zeilenumbrüche der Platzersparnis; Entnommen aus BEM, Release vom 16.08.2020

1. Summen- und Maximumbildung über „reduction“-Direktive: Für die Bildung einer Summe und der Suche nach einem Maximum (entkoppelt von einer parallelen Suche nach dem dazugehörigen Index), kann eine Parallelisierung mithilfe von „guided“ und zwei „reduction“-Direktiven verwendet werden. Zwar ergaben die Laufzeitmessungen für „static“ und „guided“ ähnliche Werte bei einer kleinen Datenmenge, jedoch wurde in diesem Fall „guided“ für den Allgemeinfall verwendet – aufgrund einer bessern Skalierung bei größer werdenden Problemen.



Beispiel für einen Algorithmus zum Aufsummieren und der Suche eines Maximums der Elemente der Matrix „topology“; Entnommen aus BEM, Release vom 16.08.2020

1. Einfügen in Vektoren mit abhängigem Index: Für die strikte Einfügung von Werten in einen Vektor wird zuallererst ein „scheduling-type“ „guided“ benötigt. Da die Einfügung der Werte strikt erfolgen soll, darf also jederzeit nur ein Thread auf den Codeblock zugreifen, welcher die Zuweisungen ausführt – welches zur Konsequenz hat, dass die Codeperformance zwar zunimmt, aber nicht skaliert wie dies bei der Parallelisierung eines Matrix-Vektor-Produkt der Fall wäre.



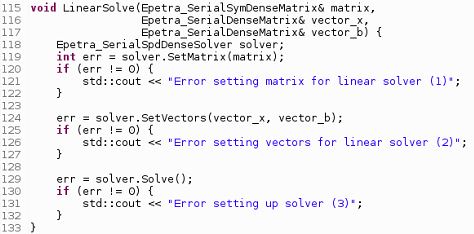
Beispiel für einen Algorithmus für die strikt indexbasierte Einfügung von Werten in einen Vektor; Entnommen aus BEM, Release vom 16.08.2020

1. Nicht parallele Regionen 1: Im Code treten immer wieder Codeabschnitte zwischen Parallelisierungsstrukturen auf, welche in mehreren Zeilen identische Befehle für verschiedene Objekte ausführt (siehe Beispiel 1). Bei der Parallelisierung und anschließenden Laufzeitmessung stellte sich jedoch heraus dass solche Konstrukte einen negativen Einfluss auf die Performance des Programmes haben – die Erstellung und Zuweisung der Aufgaben an die Threads nimmt mehr Rechenaufwand in Anspruch als die eigentliche Aufgabe. Somit ist eine Parallelisierung fragwürdig und dementsprechend nicht zu implementieren.

C:\Users\Henrik Bartsch\AppData\Local\Microsoft\Windows\INetCache\Content.Word\NotParallel_Allocations and Implementations.jpg

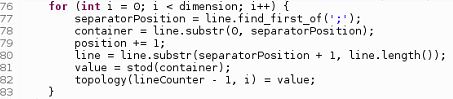
Beispiel 1 für einen nicht parallelisierbaren Codeabschnitt; Entnommen aus BEM, Release vom 16.08.2020

1. Nicht parallele Regionen 2: Bei dem Beispiel „LinearSolve“ des Code’s tritt ein weiteres typisches Parallelisierungsproblem auf: Es kann kein Codeabschnitt parallelisiert werden, welcher nur einmal ausgeführt wird und bei dem in mehreren Zeilen auf ein Objekt zugegriffen wird – ein „merging“ bei solchen nicht standartmäßigen Objekten in C++ schwer durchführbar. In diesem Fall ist dies gar nicht möglich, da die Library „Epetra“ nicht von außen einsehbar ist. Letztendlich ist auch die Parallelisierung dieses Abschnittes nicht von der Library unterstützt. Hieraus resultiert dass in diesem Codeabschnitt keine Parallelisierung implementiert wird. Durch Umstellen auf andere Bibliotheken kann das Lösen des linearen Gleichungssystems zwar parallelisiert werden, jedoch ist dies nicht Teil der Studienarbeit.



Beispiel 2 für einen nicht parallelisierbaren Codeabschnitt; Entnommen aus BEM, Release vom 16.08.2020

1. Nicht parallele Regionen 3: Ähnlich zu dem Beispiel „LinearSolve“ existiert auch in der Funktion zum Einlesen von Matrixwerten eine Schleife. Wie in dem Beispiel zu sehen greifen viele Codezeilen auf die gleiche globale Variable zurück. Mögliche Lösungen wären entweder um die abhängigen Codeteile einen „critical“-Block zu ziehen oder jeweils private Variablen zu definieren, um nicht mehr auf globale Variablen zurückgreifen zu müssen. Bei beiden Lösungen ergab sich jedoch: Die Implementierung dieser Lösung wirkt sich negativ auf die Laufzeit des Programmes aus. Somit wurde eine Implementierung nicht durchgeführt.



Beispiel 3 für einen nicht parallelisierbaren Codeabschnitt; Entnommen aus BEM, Release vom 17.08.2020