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 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.
3. Cache-Befüllung: Beim ersten Durchlauf des Programms ist der Cache des Prozessors noch nicht durch das Programm befüllt worden. Hierdurch kommt es dazu, dass das Programm selbst beim ersten Programmdurchlauf bedeutend verlangsamt wird. Umgangen werden kann dieses Problem durch einmaliges Ausführen des Programms, bevor jegliche Arten von Tests beginnen oder Entleeren des Caches vor jedem Durchlauf des Programmes.

Aus diesen Störfaktoren lässt sich schließen, dass für eine genauere Interpretation der Ergebnisse mit verhältnismäßig hohem Rechenaufwand geeigneter sind, da die Störfaktoren sich dort prozentual gesehen weniger bemerkbar machen als bei Problemen mit wenig Rechenaufwand. Aus diesem Grund, wurde die minimale Laufzeit aus mindestens 100 Werten generiert. 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, die Elemente einer vollbesetzten Matrix mit Werten zu befüllen (siehe Figur 1). In „Times2“ wird ein Matrix-Vektor-Produkt gelöst (siehe Abbildung 2), während bei „Times3“ die Ausführungszeit mit indirekter Adressierung eines Matrix-Vektor-Produkts gemessen wird (siehe Abbildung 3).

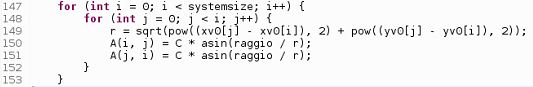


Abbildung 1: Code für Times1

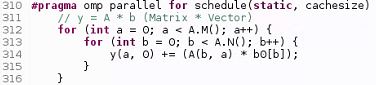


Abbildung 2: Code für Times2

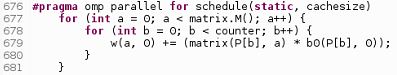


Abbildung 3: Code für Times3

Laufzeitvisualisierungen für verschiedene Problemgrößen:

Als allererstes ist zu erkennen, dass bei den Datensätzen für „Times1“ und „Times2“ die Laufzeit bei Nutzung der „scheduling types“ „dynamic“ und „guided“ länger ist als bei der Nutzung des „scheduling types“ „static“ – eine mögliche Erklärung hierfür liegt in der Funktionsweise dieser „schedules“. Während bei „static“ davon ausgegangen wird, dass jeder Schleifendurchlauf ungefähr gleich viel Zeit beansprucht und dementsprechend wenig zusätzliche Ressourcen benötigt werden, geschieht dies jedoch bei „dynamic“ und „guided“ nicht. Da die zusätzlich angeforderten Ressourcen bei dieser Aufgabe nicht benötigt werden ist die statische Herangehensweise bei dieser Aufgabe somit klar die effizienteste Variante. Ein erwartetes Ergebnis, da der „static-schedule“ für Codeabschnitte mit identischen Iterationenslaufzeiten die effizienteste Option darstellt. „guided“ als leichte Abschwächung von „static“ mit ein paar zusätzlichen Ressourcen, zum Managen von Threads, liegt in diesem Fall – wie erwartet – kurz hinter den Ausführungszeiten von „static“. „Dynamic“ liegt am Ende der Messdaten, da „dynamic“ auf Iterationen von verschiedener Komplexität ausgelegt ist und dieser Typ besonders viele Ressourcen benötigt um die Threads zu managen, welches hier ineffizient ist und dementsprechend das Programm verlangsamt.

Bei den Visualisierungen der Daten zu wachsenden Problemgrößen (für sup5 siehe: Abbildungen 4-12, sup6 siehe: Abbildungen 13-21, sup7 siehe: Abbildungen 22-30) 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“ (für die Visualisierung des Einflusses von Chunk-Änderungen siehe Abbildungen 31-33, Einfluss für Laufzeit bei Variation der Threads siehe Abbildungen 34-36). 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 bekannten 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.

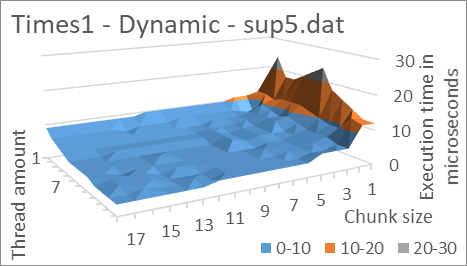
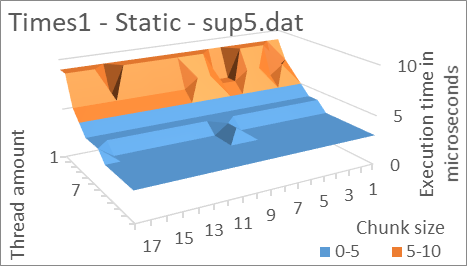


Abbildung 4 Abbildung 5

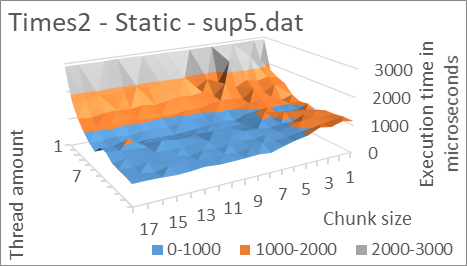
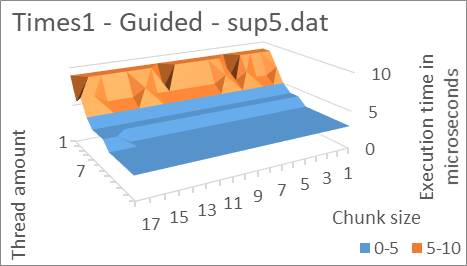


Abbildung 6 Abbildung 7

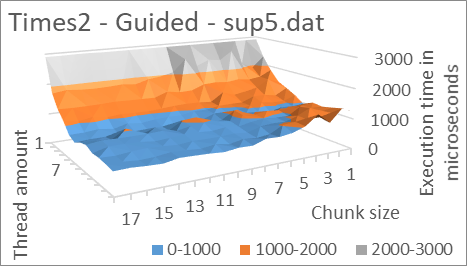
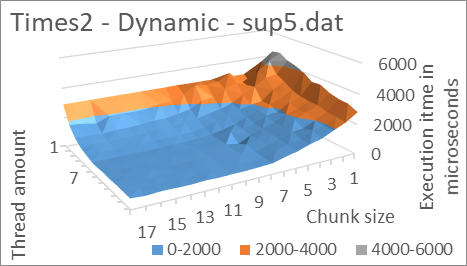


Abbildung 8 Abbildung 9

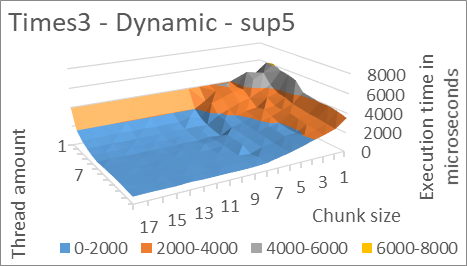
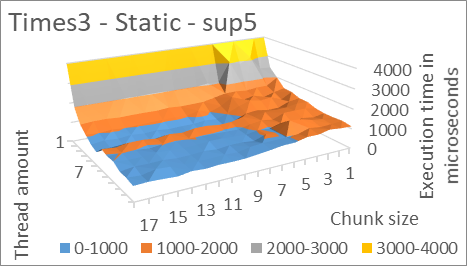
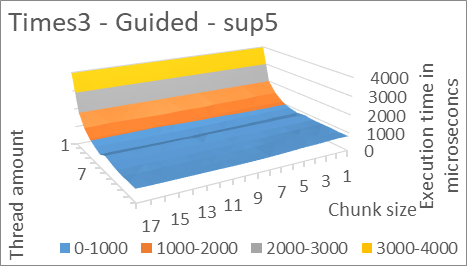


Abbildung 10 Abbildung 11



Auszug der Messdaten für sup5 zu den einzelnen Problemarten und deren Parallelisierungslaufzeiten

Abbildung 12

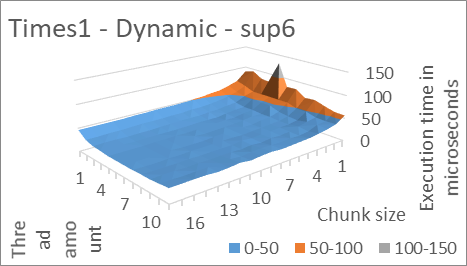
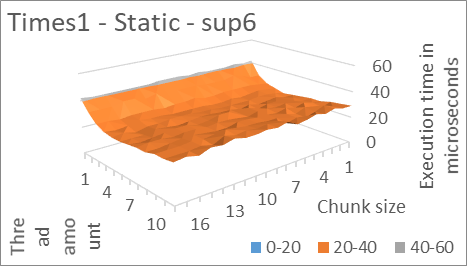


Abbildung 13 Abbildung 14

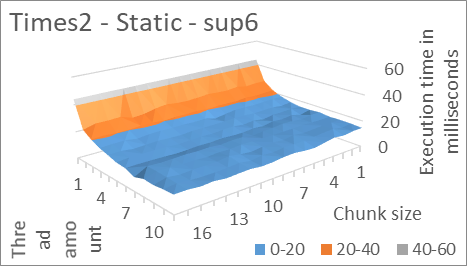
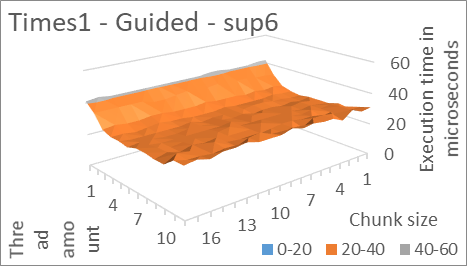


Abbildung 15 Abbildung 16

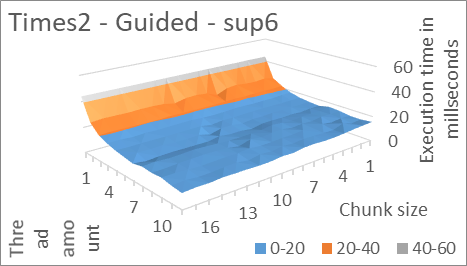
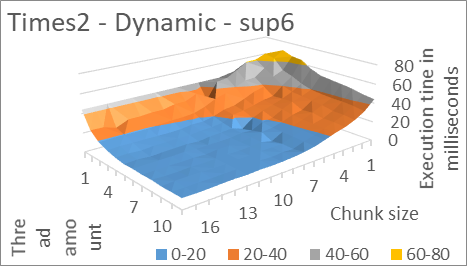
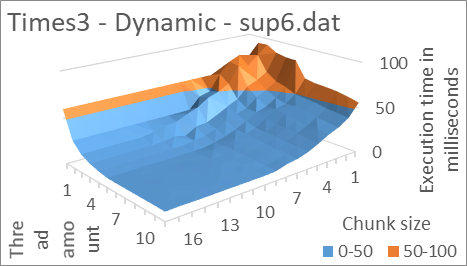
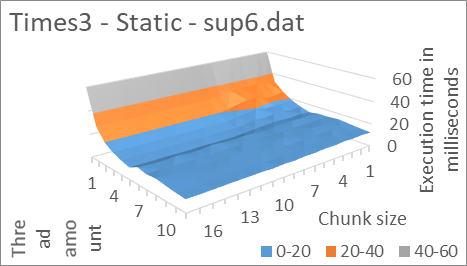


Abbildung 17 Abbildung 18



Auszug der Messdaten für sup6 zu den einzelnen Problemarten und deren Parallelisierungslaufzeiten

Abbildung 19 Abbildung 20

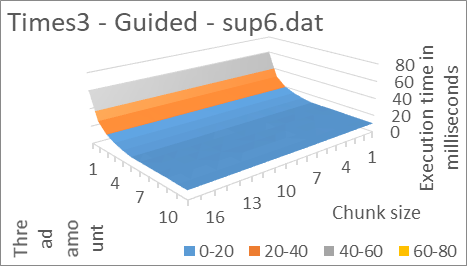


Abbildung 21

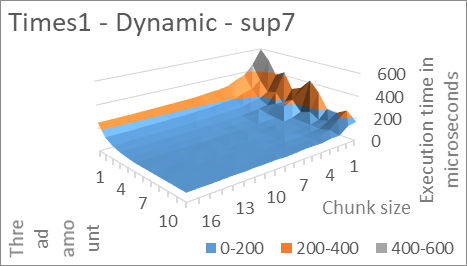
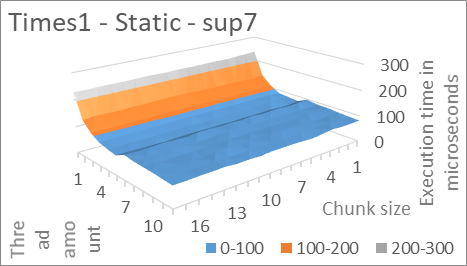


Abbildung 22 Abbildung 23

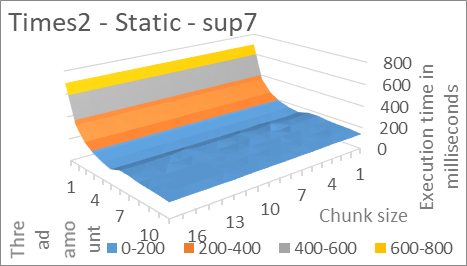
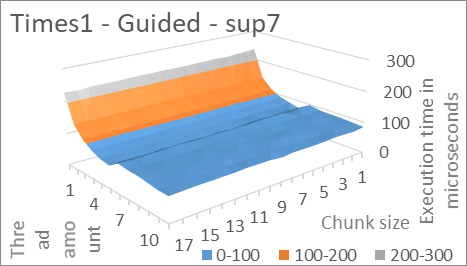


Abbildung 24 Abbildung 25

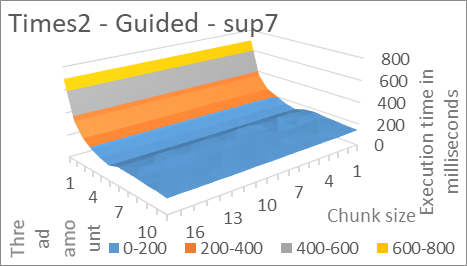
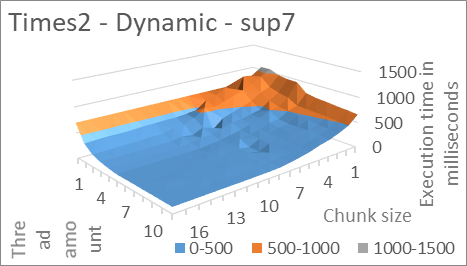


Abbildung 26 Abbildung 27

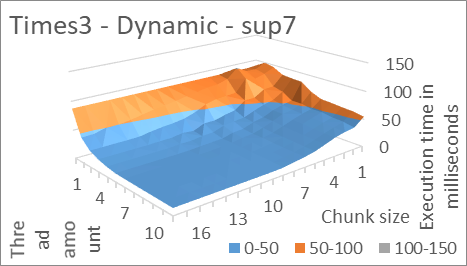
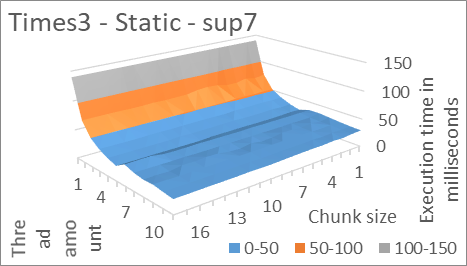
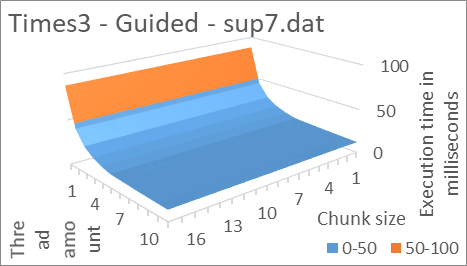


Abbildung 28 Abbildung 29

Abbildung 30

Auszug der Messdaten für sup7 zu den einzelnen Problemarten und deren Parallelisierungslaufzeiten

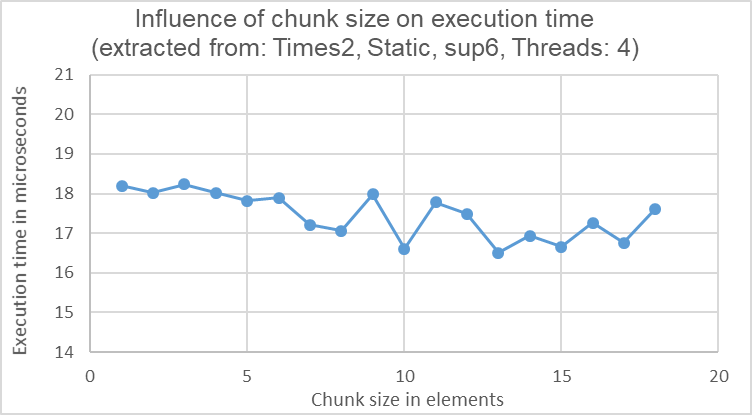


Abbildung 31

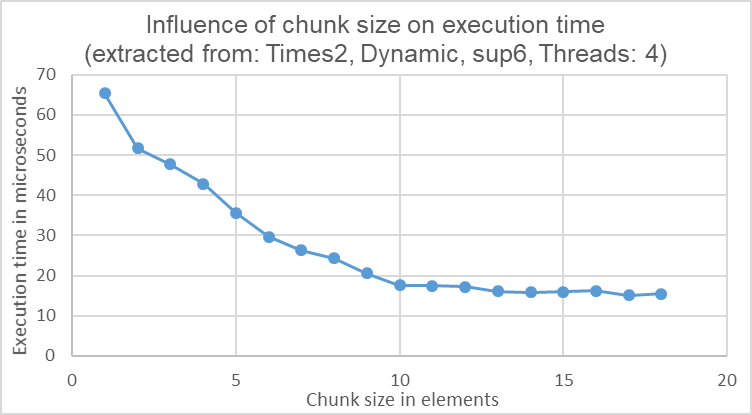


Abbildung 32

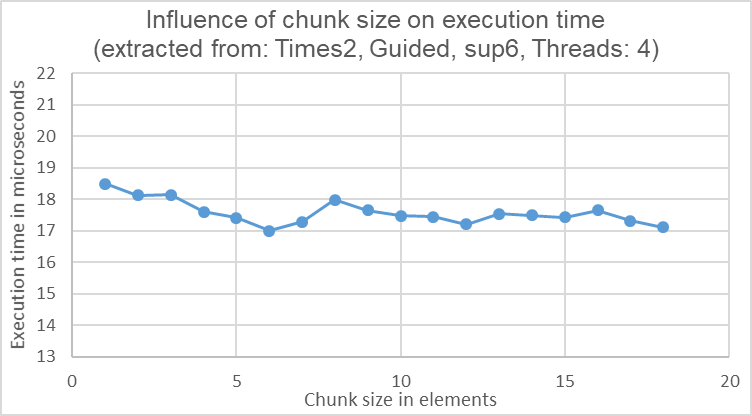


Abbildung 33

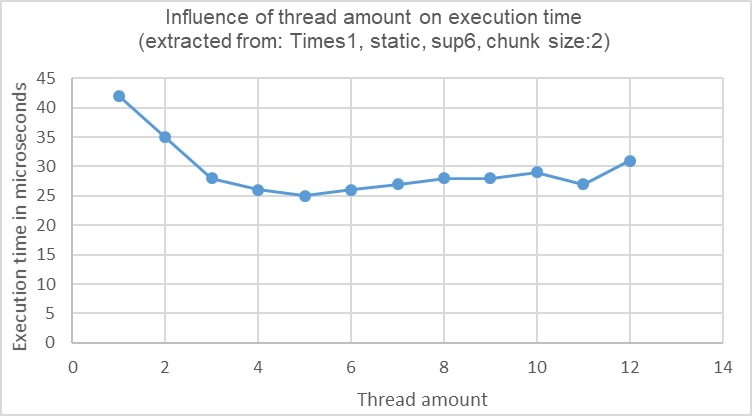


Abbildung 34

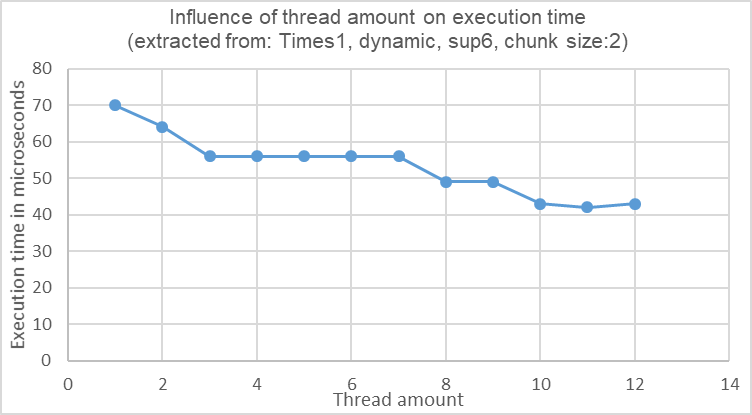


Abbildung 35

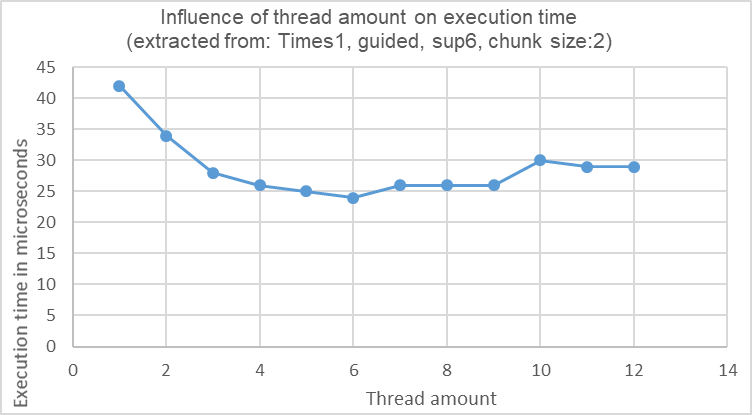


Abbildung 36

Zur Validierung der Parallelisierung, wurde eine „Strong scaling“-Skalierungsstudie erstellt. Hierbei wurde die Formel verwendet, wobei die Laufzeit mit n Threads darstellt. Die Daten wurden folgendermaßen visualisiert:

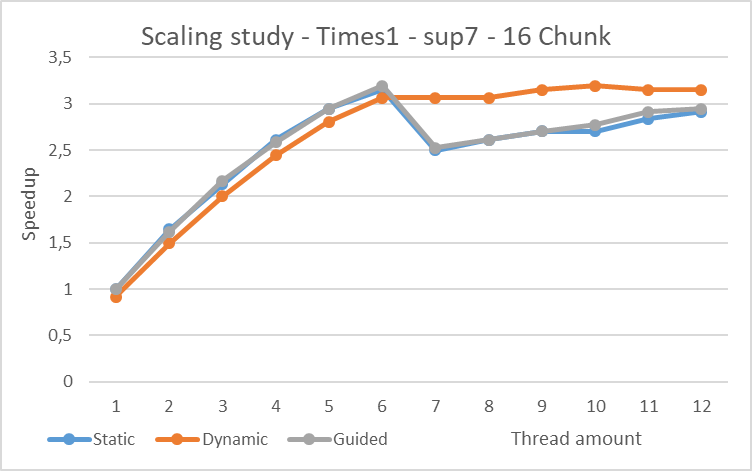


Abbildung 37

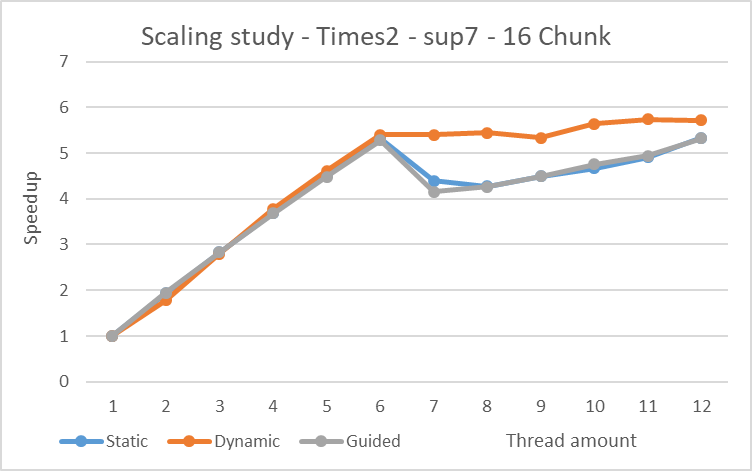


Abbildung 38

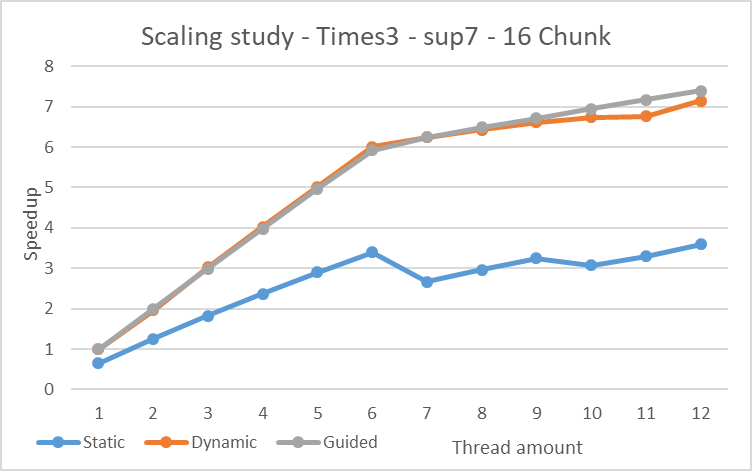


Abbildung 39

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. Der Aufgabentyp „Times1“ (siehe Abbildung 37) ist nicht groß genug, um eine ideale Skalierung zu erreichen und bleibt somit unter dem Optimum. Bei „Times2“ (siehe Abbildung 38) und „Times3“ (siehe Abbildung 39) erkennt man dann einen idealen 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 Hardware-Threads laufen können. Da jedoch nur 6 physische Prozessoren existieren, bewirkt eine Erhöhung der Threads einen Anstieg der Geschwindigkeit, die Begrenzung durch die Hardware bewirkt jedoch, dass dieser nur relativ klein ist.

Nach 6 Threads ist ein weiterer interessanter Effekt zu bemerken: Der Speedup verringert sich. Dies lässt sich durch den Umstand von „Thread placement“ erklären: Sobald 6 Threads erreicht sind, versucht das Betriebssystem die Auslastung der Prozessoren gleichmäßig aufzuteilen. Da aber mehr als 6 Threads auf 6 Prozessoren kommen, springen viele Threads zwischen den Prozessoren. Hierdurch werden u.U. auch Prozesse von ihren Daten getrennt, was zu längeren Zugriffszeiten führt (auch als „Thread affinity“ bekannt). Mit zunehmender Zahl der Threads (Threadanzahl > 6) nimmt dieser Effekt wieder ab, da die Anzahl der weniger ausgelasteten Prozessoren damit abnimmt.

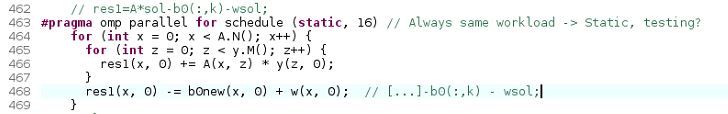
Weiterhin ist zu erkennen, dass „dynamic“ anscheinend stärker ansteigt und ein höheres Maximum hat, also die beiden anderen Typen bei „Times1“ und „Times2“. Der Grund dafür, dass „dynamic“ nicht verwendet wird ist die Tatsache, dass „dynamic“ eine bedeutend höhere Ausführungszeit hat als die anderen 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-Produkts



Beispiel einer einfachen Zuweisungsschleife

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.



Beispiel einer einfachen indirekten Adressierung

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).



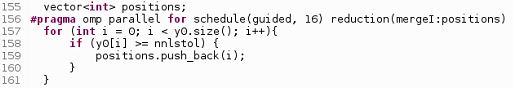
Deklaration für Vektoren mit Integer-Werten (Zeilenumbruch zur besseren Lesbarkeit)



Deklaration für Vektoren mit Integer-Werten (Zeilenumbruch zur besseren Lesbarkeit)

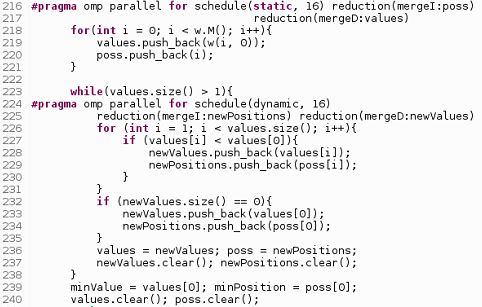
Sonderfälle

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.



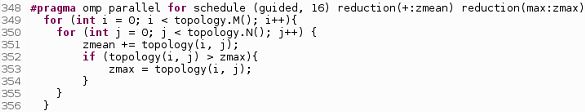
Implementierung eines Find-Befehls äquivalent zur Funktionsweise der Software MatLab

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 primären Vektoren „poss“ und „values“, für welches sich der „scheduling type“ „static“ hervorragend eignet. 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, in welcher 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 temporäre Vektor in den primären Vektor übertragen und geleert. Dies wird solange fortgeführt, bis keine größeren Werte mehr existieren. In diesem Fall werden von beiden primären Vektoren die 0-ten Einträge verwendet und in globale Variablen übertragen, damit weiterer Code abgearbeitet werden kann.



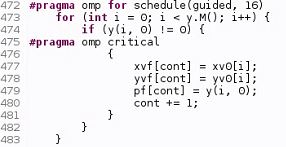
Parallele Suche eines Minimums mit gekoppelter Indexsuche des Minimums, eingefügte Zeilenumbrüche dienen der besseren Lesbarkeit

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“

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 an eine gegebene Position 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.



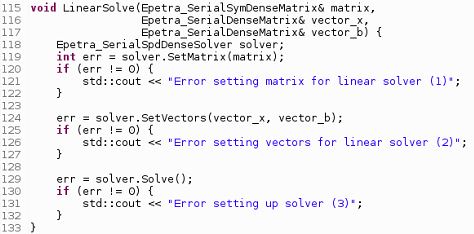
Beispiel für einen Algorithmus für die strikt indexbasierte Einfügung von Werten in einen Vektor

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 implementiert.

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

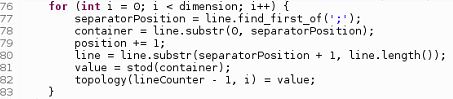
Beispiel 1 für einen nicht parallelisierbaren Codeabschnitt

1. Nicht parallele Regionen 2: 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

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 zu. 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

Skalierungsstudien des Codes:

In der Gesamtuntersuchung des Codes auf dessen Effizienz wurden verschiedene Problemgrößen auf dem HPC-Cluster Charon & Hades laufen lassen und „LinearSolve“ aus den Laufzeiten ausgenommen. Hierbei kamen folgende Ergebnisse heraus:

1. „Strong Scaling“: Um aufzuzeigen, inwiefern das Programm bei zunehmender Problemgröße mitskaliert, wurde bei dieser Skalierungsstudie zwischen den Laufzeiten der einzelnen Problemgrößen statt der „scheduling types“ unterschieden. Für das Problem „sup5.dat“ liegt der Speedup weit entfernt von den Problemen „sup6.dat“ und „sup7.dat“, dies liegt jedoch an der Tatsache, dass die Problemgröße von „sup5.dat“ einfach zu klein ist – was bereits auch in vorherigen Betrachtungen festgestellt wurde. Das Speedup von „sup6.dat“ und „sup7.dat“ sind fast identisch bis zu 8 Threads, danach wird „sup7.dat“ relativ gesehen noch ein wenig schneller als dies bei „sup6.dat“ der Fall ist. Hierdurch zeigt sich: Bei einer größeren Problemgröße scheint das Programm die Parallelisierung ein wenig besser nutzen können.

„Strong Scaling“-Studie von BEM-Code

1. „Weak scaling“: Bei dieser Skalierungsstudie werden die Laufzeiten von Problemgrößen, welche proportional mit ihren Threadzahlen steigen, gemessen. Hintergrund hierbei ist, dass eine vierfache Problemgröße bei der vierfachen Threadanzahl bei idealer Parallelisierung, keinen seriellen Anteilen und linearer Komplexität identische Ausführungszeiten besitzen müsste. In diesem Fall sind also „sup5“ bei einem, „sup6“ bei vier und „sup7“ bei 16 Threads gelaufen und deren Laufzeiten ohne den „LinearSolve“ gemessen. An der Skalierungsstudie erkennt man jedoch das dies nicht der Fall beim Code ist – eine Erklärung findet sich hierfür, dass die Komplexität nicht linear, sondern in diesem Fall zur Potenz zwei () wirkt. Die Komplexität ergibt sich im Code größtenteils aus dem nichtlinearen Lösers sowie aus den Matrix-Vektor-Produkten.

„Weak Scaling“-Studie des BEM-Codes