Parallele Implementierung und Analyse eines expliziten Adams-Verfahrens

Konrad Ley

Bayreuth Reports on Parallel and Distributed Systems

No. 2, December 2010

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Telefon: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Parallele Implementierung und Analyse eines expliziten Adams-Verfahrens

Bachelorarbeit

Universität Bayreuth

Lehrstuhl für Angewandte Informatik 2

Betreuer: Prof. Dr. Th. Rauber

Konrad Ley

Wonsees, 21.11.2010

Zusammenfassung

Das Adams-Bashforth-Verfahren ist ein numerisches Verfahren zur Lösung von gewöhnlichen Differentialgleichungen.

In dieser Arbeit werden mehrere Implementierungsvarianten des Adams-Bashforth-Verfahrens vorgestellt, verglichen und analysiert. Zunächst arbeiten die Implementierungen sequentiell. Später werden die sequentiellen Implementierungen für den Einsatz auf einem Parallelrechner erweitert.

Dabei wird besonderen Wert auf die Ausnutzung der Speicherhierarchie durch eine geschickte Organisation der Berechnungsreihenfolge gelegt. Außerdem wird bei der Synchronisation darauf geachtet, dass die Implementierungen auf Mehrkernprozessoren mit einer Shared-Memory-Architektur gut skalieren.

Ziel ist die Ausführungszeit des Adams-Bashforth-Verfahrens zu minimieren.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

In den Naturwissenschaften erfordern viele Probleme die Lösung einer Differentialgleichung.

Im Alltag ist es dabei häufig uninteressant unter großem Aufwand eine exakte analytische Lösung zu bestimmen. Es reicht häufig vollkommen aus eine hinreichend genaue numerische Lösung zu finden.

Einen Ingenieur interessieren zum Beispiel nicht die genauen Belastungen, die an einem Bauteil auftreten, sondern nur ob das Bauteil den Belastungen stand hält.

Es gibt auch Differentialgleichungen, die gar keine analytische Lösung haben. Bei ihnen ist man auf numerische Verfahren angewiesen.

Die Navier-Stokes-Gleichungen aus der Strömungsmechanik, die das Verhalten von newtonschen Flüssigkeiten beschreiben, sind zum Beispiel nur unter speziellen Randbedingungen analytisch lösbar. Man muss hier also im Allgemeinen auf die Hilfe von Computern zurückgreifen.

Ein spezielles numerisches Verfahren zur Lösung von gewöhnlichen Differentialgleichungen ist das Adams-Bashforth-Verfahren.

Es ist für bestimmte Differentialgleichungen besonders geeignet. Zum Beispiel wenn Funktionsauswertungen der rechten Seite der Differentialgleichung sehr aufwendig sind. Für einfache Differentialgleichungen sind dagegen Einschrittverfahren zu empfehlen. Siehe Kapitel 1.3.

Wir werden untersuchen ob und wie sich eine direkte Umsetzung der Rechenvorschrift des Adams-Bashforth-Verfahrens noch verbessern lässt, so dass sie die Möglichkeiten der Hardware besser ausnutzt. In dieser Arbeit werden dazu mehrere Vorschläge gemacht.

1.2 Verwandte wissenschaftliche Arbeiten

Da sich unter anderem der Lehrstuhl für Angewandte Informatik 2 an der Universität Bayreuth schon länger mit der Implementierung von Lösungsverfahren für gewöhnliche Differentialgleichung befasst, gibt es schon verwandte wissenschaftliche Arbeiten.

Im Besonderen ist hier die Dissertation mit dem Thema „Effiziente Implementierung eingebetteter Runge-Kutta-Verfahren durch Ausnutzung der Speicherzugriffslokalität“ [1] zu nennen. Mit dem sogenannten „Pipelining-Algorithmus“ wird in ihr eine besonders effiziente Berechnungsreihenfolge für Runge-Kutta-Verfahren vorgeschlagen. Der „Pipelining-Algorithmus“ wird in dieser Arbeit auf das Adams-Bashforth-Verfahren übertragen und analysiert.

1.3 Das Adams-Bashforth-Verfahren

Eine gewöhnliche Differentialgleichung hat die Form:

$$\frac{d}{dt} y(t) = f(t, y(t)) \tag{1}$$

Sie setzt also die Ableitung der gesuchten Funktion $y:\mathbb{R}\to\mathbb{R}^n$ mit der Funktion selbst in Beziehung.
Im Allgemeinen hat so eine Gleichung unendlich viele Lösungen. Um eine eindeutige Lösung zu erhalten legt man noch eine Bedingung fest:

$$y(t_0)=y_0 \tag{2}$$

Das Problem, eine Lösungsfunktion $y(t)$ für die Gleichung (1) unter der Bedingung (2) zu finden, nennt man Anfangswertproblem.
Zunächst lässt sich das Problem lösen, indem man auf beiden Seiten der Gleichung (1) vom Anfangswert t_0 bis zum interessierenden Zeitpunkt t_e integriert.

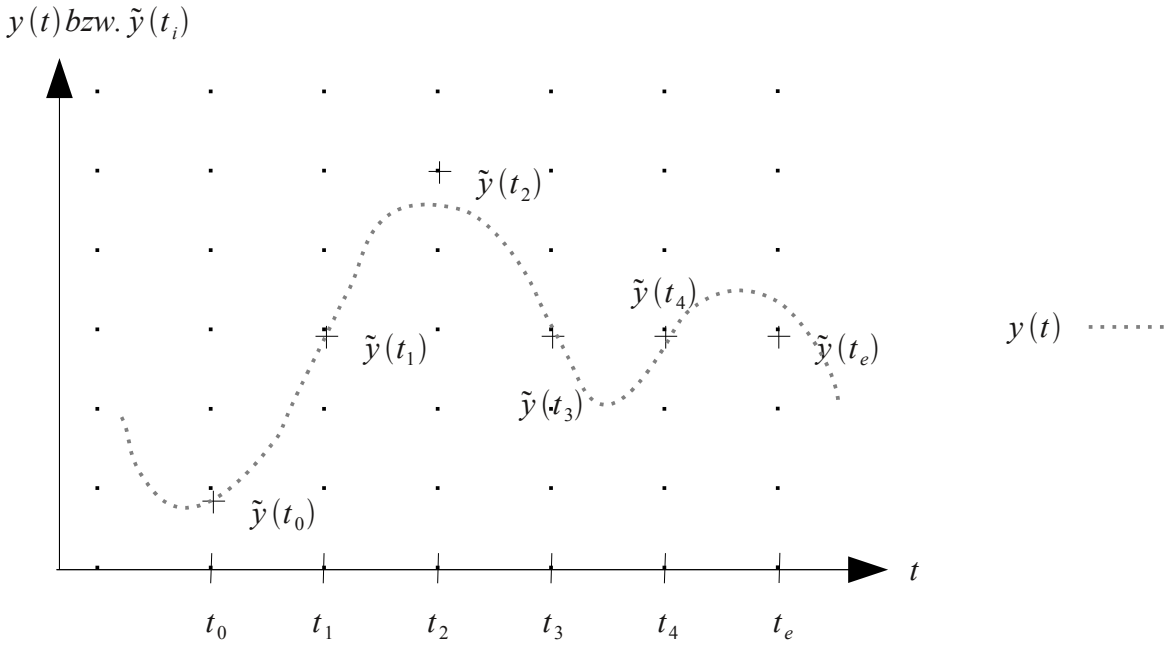
$$\int_{t_0}^{t_e}\dot{y}(\tau)\,d\tau=\int_{t_0}^{t_e}f(\tau,y(\tau))\,d\tau$$

$$y(t_e)-y(t_0)=\int_{t_0}^{t_e}f(\tau,y(\tau))\,d\tau$$

$$y(t_e)-y_0=\int_{t_0}^{t_e}f(\tau,y(\tau))\,d\tau$$

$$y(t_e)=y_0+\int_{t_0}^{t_e}f(\tau,y(\tau))\,d\tau \tag{3}$$

Das Anfangswertproblem besteht darin, das Integral $\int_{t_0}^{t_e}f(\tau,y(\tau))\,d\tau$ zu lösen, wobei zu beachten ist, dass die Lösungsfunktion $y(t)$ im Integranden enthalten ist.
Eine analytische Lösung auf dem gewöhnlichen Weg ist daher nicht möglich.
Es ist aber möglich, numerisch zu integrieren, d.h. das Integral zu approximieren. Man berechnet ausgehend vom Anfangswert $y(t_0)=y_0$ schrittweise die folgenden Werte $\tilde{y}(t_1),\tilde{y}(t_2),\ldots,\tilde{y}(t_e)$ und nutzt in jedem Schritt die vorangegangenen schon berechneten Werte.
Um ein Integral näherungsweise zu berechnen, gibt es verschiedene Ideen und Möglichkeiten, die zu unterschiedlichen Lösungsverfahren führen.
Alle Lösungsverfahren errechnen approximierte Werte $\tilde{y}(t_i)$ der Lösungsfunktion $y(t)$ zu diskreten Zeitpunkten $t_0,t_1,\ldots,t_e$. Wir verwenden einen konstanten Abstand h ,genannt Schrittweite, zwischen den diskreten Zeitpunkten. D.h. wir verwenden die Zeitpunkte $t_i=t_0+i*h$ mit Index i von 0 bis $\left\lfloor\frac{t_e-t_0}{h}\right\rfloor$. Man kann sich das wie folgt vorstellen:



Die Gitterfunktion $\tilde{y}(t_{i+1})$ wird sequentiell, angefangen beim Zeitpunkt t_0 bis zum interessierenden Zeitpunkt t_e, in sogenannten Zeitschritten berechnet. Bei einem k-Schrittverfahren wird in einem Zeitschritt aus k der schon berechneten Näherungswerte $\tilde{y}(t_{i-j})$ mit $0 \le j < k$ bzw. dem Anfangswert $y(t_0)$ ein neuer approximierter Lösungsfunktionswert $\tilde{y}(t_{i+1})$ ermittelt.
Die hierfür nötige Berechnungsvorschrift für den i-ten Zeitschritt des expliziten Adams-Bashforth-Verfahren lautet:

$$\tilde{y}(t_{i+1}) = \tilde{y}(t_i) + h \sum_{j=0}^{k-1} b_j f(t_{i-j}, \tilde{y}(t_{i-j})) \tag{4}$$

Die Gewichtungsfaktoren b_j können nach der folgenden Formel berechnet werden:

$$b_j = \frac{(-1)^j}{j!(s-j)!} \int_0^1 \prod_{i=-1, i \neq j}^{k-1} (u+i)\, du\,,\ j = 0, \ldots, k-1 \tag{5}$$

k kann beliebig gewählt werden. Eine größeres k erhöht den Aufwand, aber auch die Genauigkeit der approximierten Werte. Typische Werte für k sind z.B. $3, 4, \ldots, 6$.
Auf die Herleitung der Berechnungsvorschrift (4) und (5) wird an dieser Stelle verzichtet, da sie nicht Gegenstand der Arbeit und für das Verständnis dieser Arbeit nicht notwendig ist. Sie kann in entsprechender Literatur nachgelesen werden. [2]
Um nach dem Adams-Bashforth-Verfahren eine näherungsweise Lösung $\tilde{y}(t_{i+1})$ zu bestimmen, muss man also zur Lösung des vorhergehenden Zeitschritts $\tilde{y}(t_i)$ k Funktionsauswertungen der rechten Seite der Differentialgleichung $f(t_{i-j}, \tilde{y}(t_{i-j}))$ gewichtet mit b_j aufsummieren und mit der Schrittweite h multipliziert hinzuaddieren.
Um z.B. für $k = 3$, $t_0 = 3{,}4$ und $h = 0{,}1$ den angenäherten Funktionswert $\tilde{y}(13{,}1)$ an der Stelle $t_e = 13{,}1$ auszurechnen, sind folgende Berechnungen zu vollziehen:

Zeitschritt i	Zeitpunkt t_i	Berechnung
2	3,6	$\tilde{y}(t_3)=\tilde{y}(t_2)+h*(b_0*f(t_2,\tilde{y}(t_2))+b_1*f(t_1,\tilde{y}(t_1))+b_2*f(t_0,\tilde{y}(t_0)))$
3	3,7	$\tilde{y}(t_4)=\tilde{y}(t_3)+h*(b_0*f(t_3,\tilde{y}(t_3))+b_1*f(t_2,\tilde{y}(t_2))+b_2*f(t_1,\tilde{y}(t_1)))$
4	3,8	$\tilde{y}(t_5)=\tilde{y}(t_4)+h*(b_0*f(t_4,\tilde{y}(t_4))+b_1*f(t_3,\tilde{y}(t_3))+b_2*f(t_2,\tilde{y}(t_2)))$
⋮	⋮	⋮
96	13	$\tilde{y}(t_{97})=\tilde{y}(t_{96})+h*(b_0*f(t_{96},\tilde{y}(t_{96}))+b_1*f(t_{95},\tilde{y}(t_{95}))+b_2*f(t_{94},\tilde{y}(t_{94})))$

Man nennt ein solches Verfahren auch Mehrschrittverfahren, weil in einem Zeitschritt mit $\tilde{y}(t_{i-j})$ auf k vorangegangene Schritte zurückgegriffen wird. Wo hingegen die Einschrittverfahren, z.B. das Runge-Kutta-Verfahren, jeweils nur einen Anfangswert für jeden Zeitschritt benötigen.
Weil bei einem Anfangswertproblem nur der eine Wert $y(t_0)$ gegeben ist, müssen bei Mehrschrittverfahren zu Beginn die ersten $k-1$ Werte anderweitig, zum Beispiel durch Einschrittverfahren, berechnet werden.
Außerdem ist eine Adaption der Schrittweite h zur Laufzeit schwieriger, als bei einem Einschrittverfahren. Denn will man bei einem Mehrschrittverfahren die Schrittweite verändern und mit einer kleineren Schrittweite weiter rechnen, werden neue Funktionswerte zwischen den schon berechneten gebraucht. Diese können zum Beispiel durch Interpolation bestimmt werden. Soll die Schrittweite h vergrößert werden, so benötigt man bereits berechnete Werte, die eventuell nicht mehr im Speicher sind.
Eine solche Schrittweitenkontrolle dient dazu, eine vorher festgelegte Genauigkeit des Ergebnisses sicherzustellen. Außerdem kann sie die Geschwindigkeit des Verfahrens erhöhen, wenn die Differentialgleichung über weite Teile des Integrationsintervalls große Schrittweiten zulässt.
Wie wir gesehen haben, bedeutet die Programmierung einer Schrittweitenkontrolle allerdings auch einen erheblichen Mehraufwand, weshalb hier darauf verzichtet wurde und mit einer konstanten Schrittweite gearbeitet wird.
Es ist trotzdem möglich, wenn man eine entsprechend kleine Schrittweite h wählt, ein hinreichend genaues Ergebnis zu erzielen.
Der große Vorteil des Adams-Bashforth-Verfahrens gegenüber Einschrittverfahren ist, dass pro Zeitschritt, unabhängig von k , nur eine neue Funktionsauswertung nötig ist, weshalb es bei Differentialgleichungen, bei denen Auswertungen der rechten Seiten einen großen Rechenaufwand bedeuten, den Einschrittverfahren überlegen ist.

1.4 Zielsetzung und Gliederung

Das Ziel dieser Arbeit ist es, die Laufzeit des Adams-Bashforth-Verfahrens auf aktuellen Computern mit Single- und Multicore-Prozessoren zu minimieren. Dies soll erreicht werden, indem die Wartezeiten der CPUs, die durch Synchronisation und Speicherzugriffe entstehen, so kurz wie möglich gehalten werden.

Die Wartezeit aufgrund von Speicherzugriffen kann zum Beispiel durch bessere Ausnutzung des Caches verkürzt werden. Das kann man durch Schleifenoptimierungen erreichen.

Zuerst werden im Kapitel 2 nacheinander verschiedene, teilweise aufeinander aufbauende sequentielle Implementierungen vorgestellt. Die Implementierung werden in Pseudo-Code beschrieben. Um die Vor- und Nachteile der verschiedenen Implementierung

herauszufinden, werden anschließend Messungen auf realen Rechnern gemacht. Der Pseudo-Code wurde hierfür in C übertragen. Für die Messungen wird der „Brüsselator“, eine partielle Differentialgleichung, verwendet (siehe Anhang).

Danach werden im Kapitel 3 die sequentiellen Varianten für den Einsatz auf einem Parallelrechner erweitert und ebenfalls deren Laufzeiten gemessen.

2 Sequentielle Implementierungen

In Pseudo-Code sieht das Adams-Bashforth-Verfahren, wenn man davon ausgeht, dass die k Anfangswerte $\tilde{y}(t_0),\ldots,\tilde{y}(t_{k-1})$ bereits berechnet sind, wie folgt aus:

```
// Schleife über alle Zeitschritte
for(i:=k-1; i<H/h-1; i:=i+1)
{
  ỹ(t_{i+1}):=0;
  // Schleife zur Aufsummation der k Funktionsauswertungen
  for(j:=0; j<k; j:=j+1)
  {
    ỹ(t_{i+1}) += b_j * f(t_{i-j}, ỹ(t_{i-j}));
  }
  ỹ(t_{i+1}):=ỹ(t_i)+h*ỹ(t_{i+1});
}
```

Pseudocode 1: Allgemeines Adams-Bashforth-Verfahren

Wir werden für die Implementierung des Adams-Bashforth-Verfahrens also mindestens drei verschachtelte Schleifen brauchen. Eine, die über alle Zeitschritte des Integrationsintervalls H iteriert (i-Schleife). Eine Zweite, die das gewichtete Aufsummieren der Funktionsauswertungen $f(t_{i-j},\tilde{y}(t_{i-j}))$ übernimmt (j-Schleife). Außerdem, weil das Differentialgleichungssystem mehrere Komponenten besitzt, noch eine Dritte, die über alle n Komponenten der Differentialgleichung iteriert.

Die Frage ist nun, wie man die Schleifen anordnen kann. Da im Allgemeinen zur Auswertung von $f(t_{i-j},\tilde{y}(t_{i-j}))$ alle Komponenten des vorhergehenden Zeitschritts nötig sind, muss die Schleife über alle Zeitschritte die äußerste sein. Wir werden später sehen, dass es für spezielle Differentialgleichungen möglich ist, diese Regel etwas aufzuweichen. Zwischen der „j-Schleife“, die die Funktionsauswertung aufsummiert, und der Schleife über alle Komponenten bestehen jedoch keine Abhängigkeiten, weshalb sie vertauscht werden können.

2.1 Implementierung ohne gespeicherte Funktionsauswertungen

Um diesen Algorithmus auf einem Computer umzusetzen, müssen einige Daten gespeichert werden.

Zunächst müssen die letzten k approximierten Vektoren $\tilde{y}(t_{i-j})$ mit $j=0,\ldots,k-1$ und der neu berechnete Vektor $\tilde{y}(t_{i+1})$ gespeichert werden. Hierzu wird ein zweidimensionales Array y_old der Größe

$$(k+1)*n \tag{6}$$

verwendet.

Der neue Vektor $\tilde{y}(t_{i+1})$ wird zyklisch mit jedem Zeitschritt eine Position höher im Array abgespeichert. Das erspart das Verschieben von Daten bzw. das Löschen und neu Allozieren von Speicher.

Wir schauen uns ein Beispiel für $k=3$; $t_0=3{,}4$ und $h=0{,}1$ an. Gesucht ist $\tilde{y}(13{,}1)=\tilde{y}(t_{97})$ d.h. das Integrationsintervall hat die Länge $H=9{,}7$ (neu berechneter Vektor ist grau eingefärbt) :

Zeitschritt i	***y_old[0]***	***y_old[1]***	***y_old[2]***	***y_old[3]***
2	$\tilde{y}(t_0)$	$\tilde{y}(t_1)$	$\tilde{y}(t_2)$	$\tilde{y}(t_3)$
3	$\tilde{y}(t_4)$	$\tilde{y}(t_1)$	$\tilde{y}(t_2)$	$\tilde{y}(t_3)$
4	$\tilde{y}(t_4)$	$\tilde{y}(t_5)$	$\tilde{y}(t_2)$	$\tilde{y}(t_3)$
5	$\tilde{y}(t_4)$	$\tilde{y}(t_5)$	$\tilde{y}(t_6)$	$\tilde{y}(t_3)$
6	$\tilde{y}(t_4)$	$\tilde{y}(t_5)$	$\tilde{y}(t_6)$	$\tilde{y}(t_7)$
7	$\tilde{y}(t_8)$	$\tilde{y}(t_5)$	$\tilde{y}(t_6)$	$\tilde{y}(t_7)$
⋮			⋮	
96	$\tilde{y}(t_{95})$	$\tilde{y}(t_{96})$	$\tilde{y}(t_{97})$	$\tilde{y}(t_{94})$

Der Pseudo-Code des „ogF“-Adams-Bashforth-Algorithmus (ohne gespeicherte Funktionsauswertungen) sieht wie folgt aus:

```
for(i:=k-1; i<H/h-1; i:=i+1)
{
  for(l:=0; l<n; l:=l+1)
  {
    y_old[(i+1) mod (k+1)][l]=b_0*f_l(t_0+i*h, y_old[i mod (k+1)]);
    for(j:=1; j<k; j:=j+1)
    {
      y_old[(i+1) mod (k+1)][l] +=b_j*f_l(t_0+(i-j)*h, y_old[(i-j) mod (k+1)]);
    }
    y_old[(i+1) mod (k+1)][l]:=y_old[i mod (k+1)][l]+h*y_old[(i+1) mod (k+1)][l];
  }
}
```

Pseudocode 2: Implementierung „ogF“ ohne gespeicherte Funktionsauswertungen

2.2 Implementierung mit gespeicherten Funktionsauswertungen

Wenn man sich die Berechnungsvorschrift (4) anschaut, stellt man fest, dass wie eingangs schon erwähnt, die rechte Seite der Differentialgleichung $f(y(t),t)$ pro Zeitschritt nur einmal ausgewertet werden muss, da sich die Funktionsauswertungen zweier aufeinander folgender Zeitschritte überschneiden.

Für $k=3$ werden im ersten Zeitschritt ($i=2$) zum Beispiel folgende Funktionsauswertungen gemacht: $f(t_0,\tilde{y}(t_0))$, $f(t_1,\tilde{y}(t_1))$ und $f(t_2,\tilde{y}(t_2))$. Im Zweiten: $f(t_1,\tilde{y}(t_1))$, $f(t_2,\tilde{y}(t_2))$ und $f(t_3,\tilde{y}(t_3))$. Wenn man die Funktionsauswertungen aus der ersten Iteration speichert, muss für die Zweite nur $f(t_3,\tilde{y}(t_3))$ neu berechnet werden.

Das Speichern der alten Funktionsauswertungen erhöht den Speicherbedarf nicht, da die Werte der Lösungsfunktion $\tilde{y}(t_{i-j})$, die als Argumente für die Funktionsauswertungen $f(t_{i-j},\tilde{y}(t_{i-j}))$ gebraucht wurden, nun nicht mehr gespeichert werden müssen. Man braucht also für die Berechnungsvorschrift (4) die approximative Lösung des letzten Zeitschritts $\tilde{y}(t_i)$ und die $k-1$ Funktionsauswertungen $f(t_{i-j},\tilde{y}(t_{i-j}))$ aus den letzten Zeitschritten, die wieder verwendet werden. Zu guter Letzt natürlich auch einen Speicher für den neuen Vektor $\tilde{y}(t_{i+1})$. Man braucht also bei der Implementierung „mgF“ (mit gespeicherten Funktionsauswertungen).

$$(k+1)*n \tag{7}$$

Speicherplätze und damit genauso viele wie bei „ogF“ (6).

Im Array *storedFunctionValues* für die Werte $f(t_{i+j},\tilde{y}(t_{i+j}))$ wird wieder zyklisch die zeitlich letzte von der neuen Funktionsauswertung überschrieben, wie bei der Implementierung „ogF“ die Werte der Lösungsfunktion $\tilde{y}(t_i)$.

Implementierung „mgF“ in Pseudo-Code:

```
for(i:=k-1; i<H/h-1; i:=i+1)
{
  for(l:=0; l<n; l:=l+1)
  {
    y_old[(i+1) mod 2][l]:=b_{k-1}*storedFunctionValues[i mod (k-1)][l]
    for(j:=1; j<k-1; j:=j+1)
    {
      y_old[(i+1) mod 2][l] += b_j*storedFunctionValues[(i-j) mod (k-1)][l];
    }
    storedFunctionValues[i mod (k-1)][l]:=f_l(t_0+i*h, y_old[i mod 2]);
    y_old[(i+1) mod 2][l] += b_0*storedFunctionValues[i mod (k-1)][l];
    y_old[(i+1) mod 2][l]:=y_old[i mod 2][l]+h*y_old[(i+1) mod 2][l];
  }
}
```

Pseudocode 3: Implementierung "mgF" mit gespeicherten Funktionsauswertungen

2.3 Vergleich der Laufzeiten von „mgF“ und „ogF“

Bei der Implementierung „mgF“ werden

$$n*k*s \tag{8}$$

Auswertungen von Komponentenfunktionen f_l mit $l \in \{0;1;2;\ldots;n-1\}$ gemacht. Wobei $s=\frac{H}{h}-k$ die Anzahl der Zeitschritte ist.

In der Implementierung „mgF“ wird jedoch pro Zeitschritt und Komponente nur eine Funktionsauswertung gemacht, d.h. es sind

$$n*s \tag{9}$$

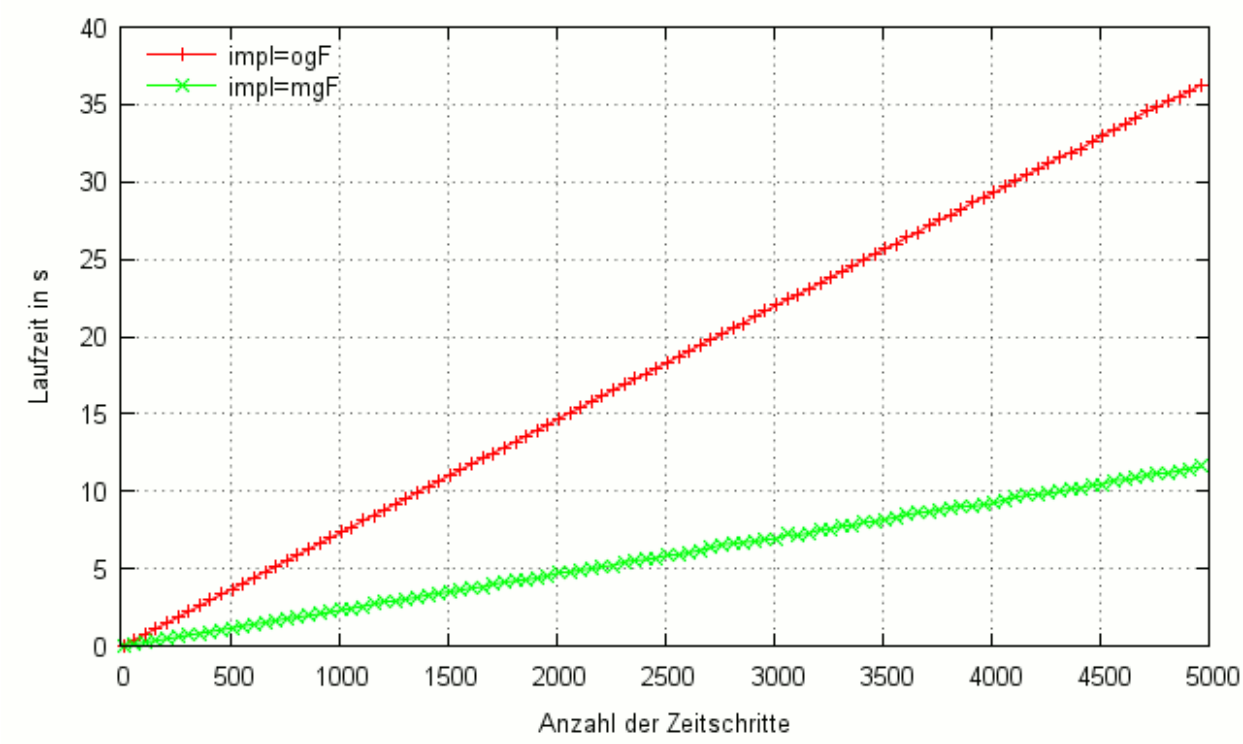
Komponentenfunktionsauswertungen.

Ist die Anzahl der Funktionsauswertungen geringer, bedeutet das, dass weniger Speicherzugriffe und Rechenoperationen nötig sind.

Misst man die Laufzeit in Abhängigkeit von der Anzahl der Zeitschritte s, so ist für Implementierung „mgF“ eine flachere Gerade zu erwarten.

Im Folgenden verwenden wir für alle Messungen das Bruss2D-Mix-Problem (siehe Anhang) und für das Adams-Bashforth-Verfahren $k=6$.

In Messung 1 messen wir für $n=20000$ Komponenten für Integrationsintervalle verschiedener Längen auf Rechner 1 die Gesamtlaufzeiten. (Nähere Angaben zu den verwendeten Rechnern befinden sich im Anhang)



Messreihe 1: Gesamtlaufzeiten der Implementierungen „ogF“ und „mgF“ in Abhängigkeit von der Anzahl der berechneten Zeitschritte auf Rechner 1 bei n=20000 Komponenten.

Man sieht also in der Messreihe 1, dass die Laufzeit der Implementierung „mgF“ immer kürzer ist, als die der Implementierung „ogF“.

Außerdem wird uns in Zukunft noch eine andere Größe interessieren: Die Laufzeit pro Zeitschritt und Komponente, im Folgenden auch normierte Laufzeit genannt. Sie ist deshalb interessant, weil sie bei unserem Testproblem Bruss2D-Mix unabhängig von der Anzahl der Komponenten n gleich groß sein sollte, da in jedem Zeitschritt pro Komponente die gleiche Anzahl an Rechenoperationen und Speicherzugriffen gemacht werden. (siehe Anhang)

2.4 Implementierung mit loop-interchange

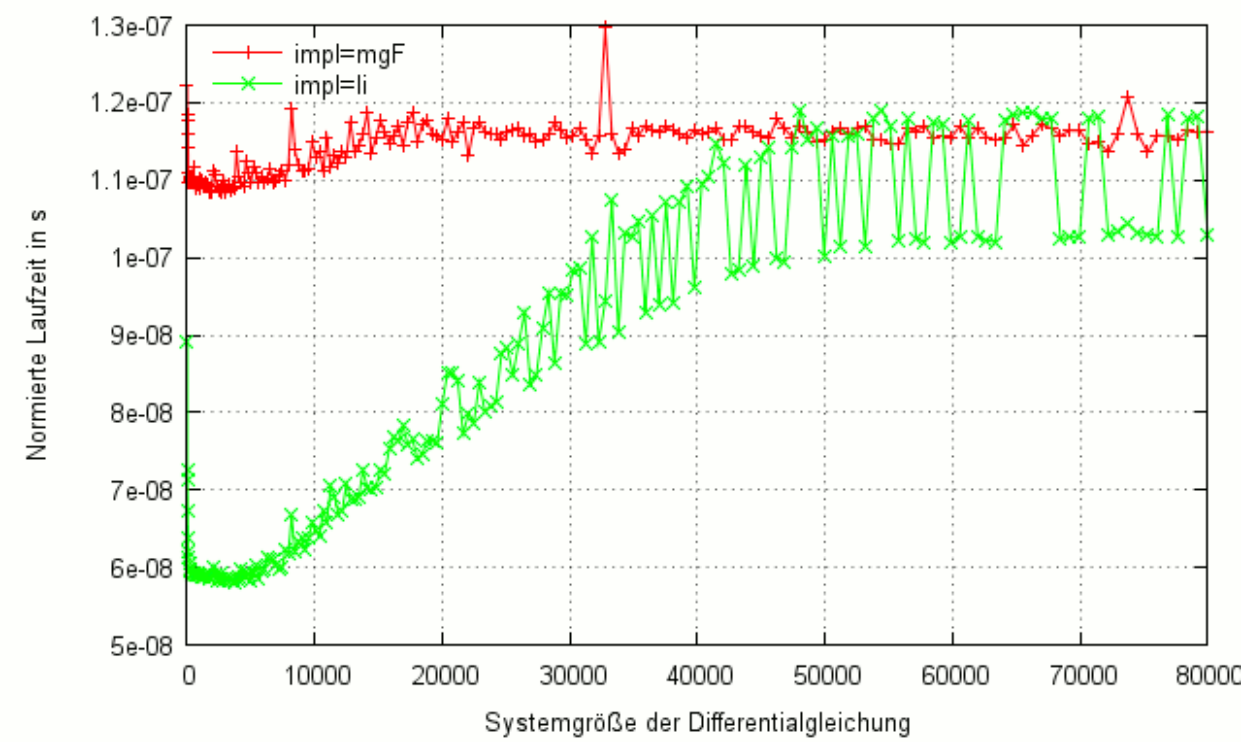
Wie in der Einleitung zum Kapitel 2 schon beschrieben, können wir die Schleife über alle Komponenten und die Schleife über die gespeicherten Funktionsauswertungen vertauschen. Diese neue Variante nennen wir „li“ (loop-interchange).

```
for(i:=k-1;i<H/h-1;i:=i+1)
{
  for(l:=0;l<n;l:=l+1)
    y_old[(i+1)mod 2][l]:=b_{k-1}*stored_function_values[i mod (k-1)][l]

  for(j:=1;j<k-1;j:=j+1)
  {
    for(l:=0;l<n;l:=l+1)
      y_old[(i+1)mod 2][l] += b_j*stored_function_values[(i-j)mod(k-1)][l];
  }
  for(l:=0;l<n;l:=l+1)
    stored_function_values[i mod (k-1)][l]:=f_l(t_0+i*h, y_old[i mod 2]);

  for(l:=0;l<n;l:=l+1)
    y_old[(i+1)mod 2][l] += b_0*stored_function_values[i mod (k-1)][l];

  for(l:=0;l<n;l:=l+1)
    y_old[(i+1)mod 2][l]:=y_old[i mod 2][l]+h*y_old[(i+1)mod 2][l];
  }
}
```

Pseudocode 4: Implementierung „li" mit loop-interchange

2.5 Vergleich der Laufzeiten von „mgF“ und „li“

Wir vergleichen die normierten Laufzeiten der beiden Implementierung „mgF“ und „li“ für verschieden große Differentialgleichungssysteme n.



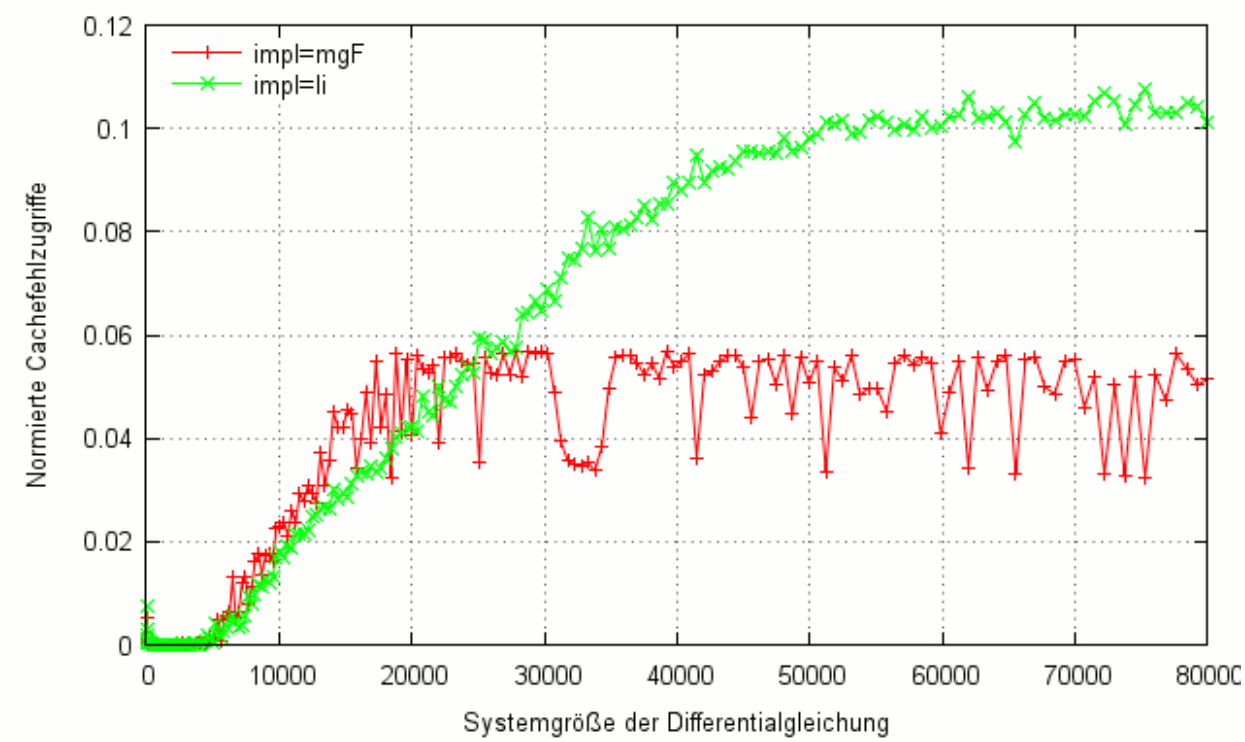
Messreihe 2: Normierte Laufzeiten der Implementierung „mgF“ und „li“ für verschieden große Differentialgleichungssysteme auf Rechner 1bei einem Integrationsintervall H=1 und Schrittweite h=0.001.

Wir sehen, dass die Variante „li“ bei kleinen Differentialgleichungssystemen schneller ist.

Die Frage ist nun, woran das liegt.

Ein Grund für den Laufzeitanstieg von Variante „li“ für $n<9000$ erschließt sich, wenn man sich die Systemdaten des Rechners 1 ansieht, auf dem die Messungen durchgeführt wurden. Der verwendete Pentium 4 Prozessor verfügt über eine L2-Cachegröße von 512 KByte. Bei $n=9000$ Komponenten, haben die Daten, auf die der Adams-Bashforth-Algorithmus für $k=6$ zugreift, nach (7) eine Größe von ungefähr $(n*(k+1))*8\text{Byte}=(9000*7)*8\text{Byte}\approx 492\,KByte$.(Wir rechnen hier mit Double-Werten der Größe 8Byte). Das bedeutet ab $n=9000$ passen nicht mehr alle verwendeten Daten in den Cache. Die dadurch entstehenden Zugriffe auf den Hauptspeicher sind langsamer und die Laufzeit des Programms steigt an.

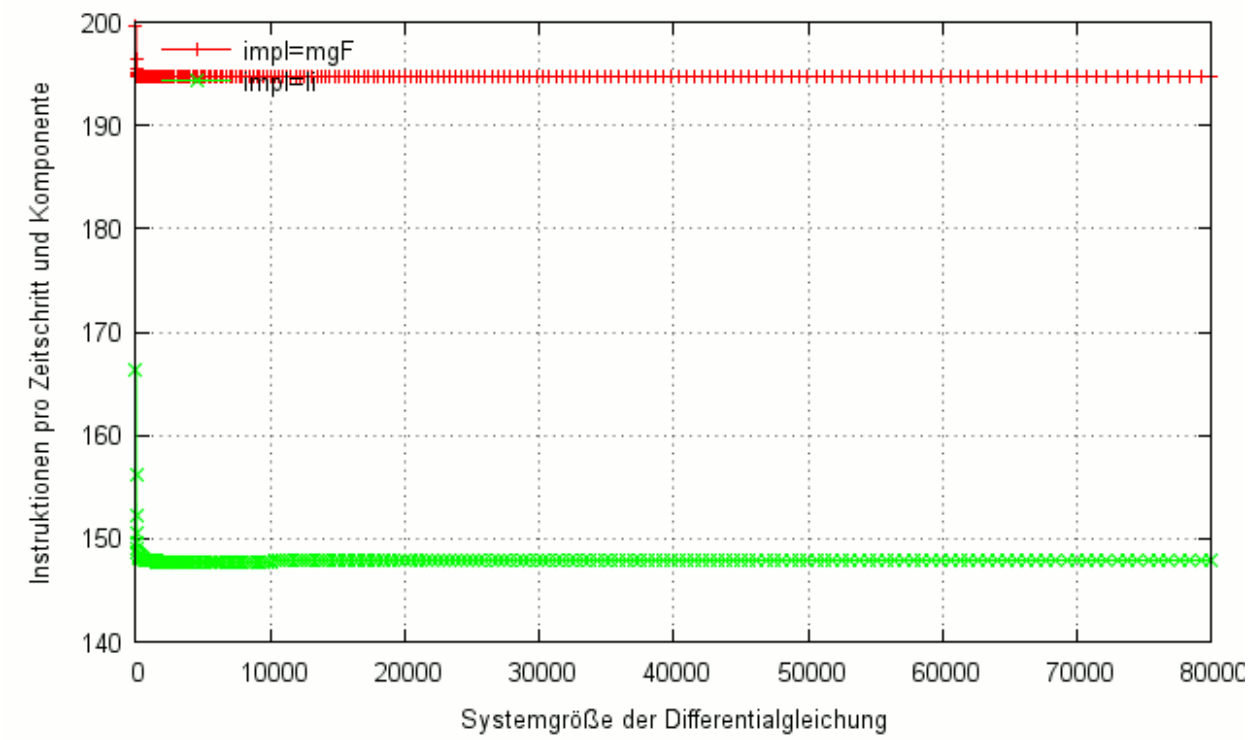
Das sieht man auch, wenn man mit den Hardware Performance Countern die Cachefehlzugriffe beim Zugriff auf den L2-Cache misst.



Messreihe 3: Normierte Cachefehlzugriffe der Implementierung „mgF“ und „li“ in Abhängigkeit von der Anzahl der Komponenten der Differentialgleichung auf Rechner 1 bei einem Integrationsintervall von H=1 und Schrittweite h=0.001

Man sieht in Messreihe3, dass tatsächlich ab ca. $n=9000$ die Cachefehlzugriffe pro Komponente für Implementierung „li“ ansteigen. Leider erklärt das nur den Laufzeitanstieg der Implementierung „li“ und nicht, warum Implementierung „li“ schneller ist als „mgF“. Die Implementierung „mgF“ und „li“ verursachen z.B. für $n<9000$ beide kaum Cachefehlzugriffe, trotzdem ist Implementierung „li“ schneller.

Um das zu erklären, schauen wir uns in Messreihe 4 die Anzahl der Instruktionen an, die beide Implementierungen machen:



Messreihe 4: Instruktionen pro Zeitschritt und Komponente für die Implementierung „mgF“ und „li“ auf Rechner 1 bei verschiedenen Differentialgleichungsgrößen.bei einem Integrationsintervall von H=1 und Schrittweite h=0.001

Nach Messreihe 4 werden bei der Implementierung „mgF“ mehr Instruktionen ausgeführt als bei „li“ und das bedeutet, dass bei gleicher L2-Cacheausnutzung zur Abarbeitung von „mgF“ mehr Zeit benötigt wird.

2.6 Implementierung mit Loop-tiling

Wir wollen nun die Speicherzugriffe der Implementierung „li“ verbessern.

Für kleine Differentialgleichung scheint „li“ schon optimal zu sein, denn dort sind die normierten Cachefehler bei Messreihe 3 schon nahezu Null. Danach steigt die Anzahl der Cachefehlzugriffe mit n an. Wenn wir uns den Pseudo-Code 4 der Implementierung „li“ anschauen, stellen wir fest, dass mit größerem n die j-Schleifen über größere Arrays iterieren und das ist, hinsichtlich der Cachefehler, ungünstig, denn dann passen nicht mehr alle Elemente des Arrays, die in der Schleife verwendet werden, in den Cache. Bei einem Schleifendurchlauf müssen also mehrmals Daten in den Cache nachgeladen werden. Weil Implementierung „li“ mehrere solcher Schleifen enthält, erklärt das, warum Implementierung „mgF“ bei großen Differentialgleichungen weniger Cachefehlzugriffe erzeugt, weil dort nur einmal über alle n Elemente der Arrays iteriert wird.

Mit der Variante „lt“ wollen wir einen optimalen Mittelweg zwischen den Varianten „li“ und „mgF“ finden.

Die optimale Strategie wäre, die Daten einmal in Cache zu laden und anschließend die Daten solange wiederzuverwenden, bis alle Rechnungen auf ihnen abgeschlossen sind. Danach laden wir die nächsten Daten in den Cache.

Da in den j-Schleifen teilweise auf den selben Daten gearbeitet wird, lassen wir also die j-Schleifen nur $blocksize$ Iterationen ausführen. $blocksize$ wählen wir so, dass noch alle Elemente, die in allen j-Schleifen gebraucht werden, in den Cache passen. Dadurch

erreichen wir, dass die Daten solange im Cache bleiben, bis alle Rechnungen auf ihnen ausgeführt wurden.

Die neuen Daten laden wir in den Cache, indem wir mit einer äußeren Schleife $blocksize$ Iterationen weiter springen. Diese Technik nennt sich loop-tiling.

Auf unseren Adams-Bashforth-Algorithmus angewendet sieht das wie folgt aus:

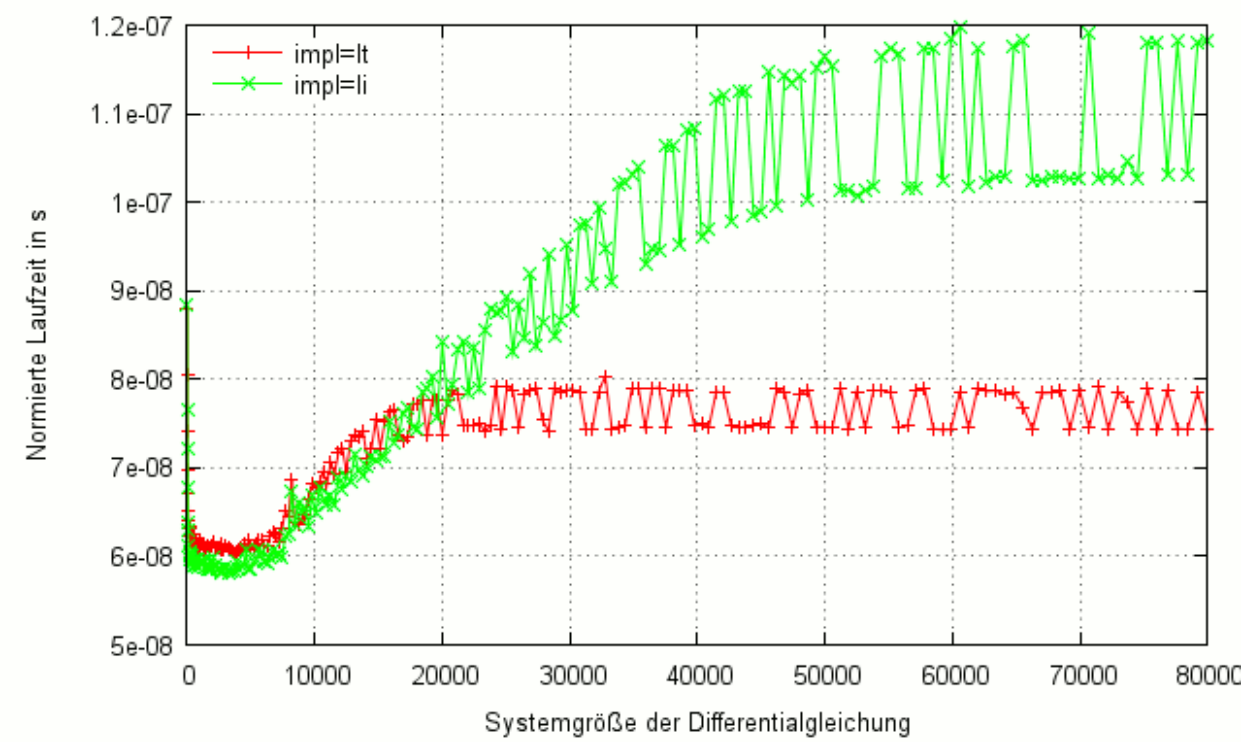
```
for(i:=k-1; i<H/h-1; i:=i+1)
{
  endOfBlock:=min(l+blocksize, n);

  for(l:=0; l<n; l:=l+blocksize)
  {
    for(m:=l; m<endOfBlock; m:=m+1)
      y_old[(i+1) mod 2][m]:=b_{k-1}*stored_function_values[i mod (k-1)][m];

    for(j:=1; j<k-1; j:=j+1)
    {
      for(m:=l; m<endOfBlock; m:=m+1)
        y_old[(i+1) mod 2][m] += b_j*stored_function_values[(i-j) mod (k-1)][m];
    }
    for(m:=l; m<endOfBlock; m:=m+1)
      stored_function_values[i mod (k-1)][m]:=f_m(t_0+i*h, y_old[i mod 2]);

    for(m:=l; m<endOfBlock; m:=m+1)
      y_old[(i+1) mod 2][m] += b_0*stored_function_value[i mod (k-1)][m];

  for(m:=l; m<endOfBlock; m:=m+1)
    y_old[(i+1) mod 2][m]:=y_old[i mod 2][m]+h*y_old[(i+1) mod 2][m];
  }
}
```

Pseudocode 5: Implementierung "lt" mit loop-tiling

Der Speicherbedarf der Implementierung „lt“ (7) ändert sich gegenüber der Implementierung „li“ nicht.

2.7 Vergleich der Laufzeiten von „li“ und „lt“

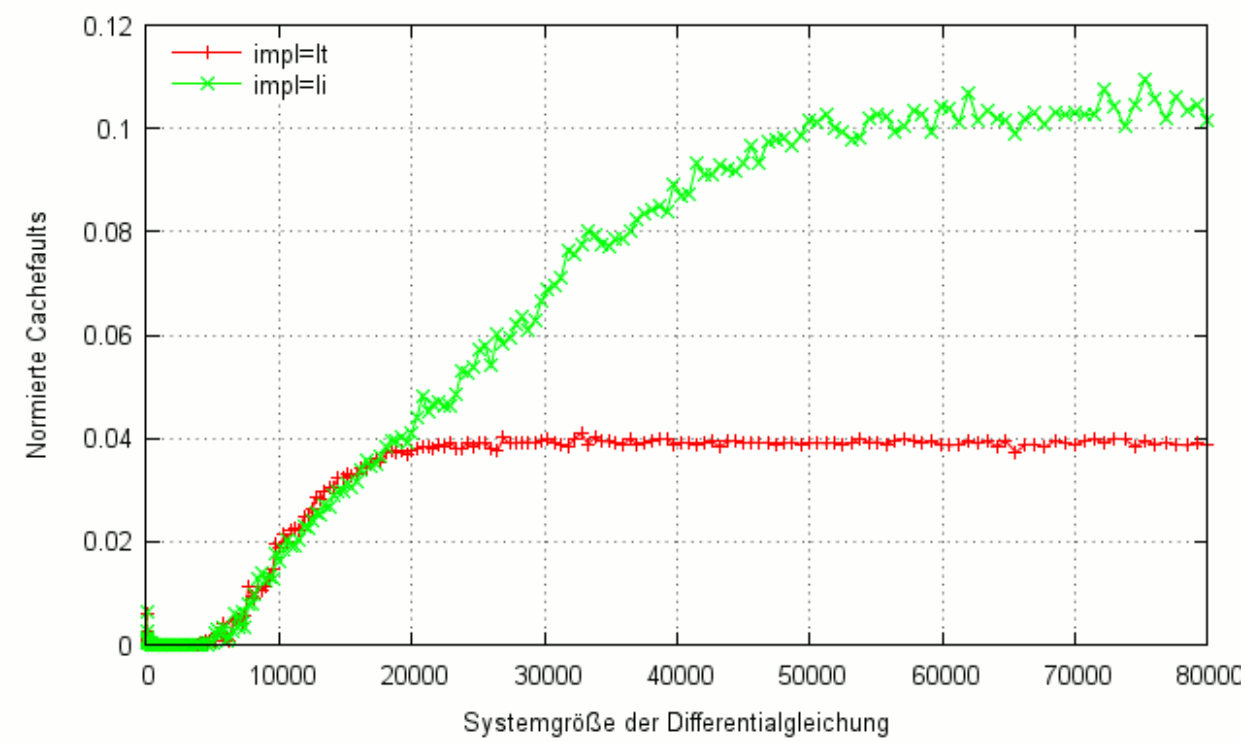
Wir wollen nun überprüfen ob das loop-tiling die Ausführungsgeschwindigkeit erhöhen konnte. Wir wählen für unsere Blockgröße $blocksize = 9000$.



Messreihe 5: Vergleich der normierten Laufzeiten der Implementierungen „lt“ und „li“ in Abhängigkeit von der Größe der Differentialgleichung auf Rechner 1 bei einem Integrationsintervall von H=1 und Schrittweite h=0.001

Man stellt fest, dass die Implementierung „lt“ mit loop-tiling ab einer Systemgröße von $n = 20000$ schneller ist.

Dass das loop-tiling wirklich die Anzahl der Cachefehlzugriffe verringert, sehen wir in folgender Messreihe 6:



Messreihe 6: Vergleich der normierten Cachefehlzugriffe beim Zugriff auf den L2-Cache der Implementierungen „lt“ und „li“ in Abhängigkeit von der Größe der Differentialgleichung auf Rechner 1 bei einem Integrationsintervall von H=1und Schrittweite h=0.001

Die Frage ist nun, ob $blocksize = 9000$ schon die optimale Wahl ist, oder ob man mit einer anderen Blockgröße noch schneller sein kann.

Theoretisch ist die Blockgröße dann optimal gewählt, wenn die Daten eines Blockes genau in den Cache passen. Ist die Blockgröße kleiner, wird der Cache nicht voll genutzt. Ist sie größer, passen nicht mehr alle Daten des Blocks in den Cache.

Wir wollen anhand des Pseudo-Codes für Implementierung „lt“ die Größe eines Blockes , der in einer Iteration der „l-Scheife“ benötigt wird, bestimmen. Wir verwenden dort

- den Vektor b der Größe k
- $blocksize$ Werte des Vektors $y\_old[(i+1) mod\ 2]$
- $blocksize + 2 * d(f)$ Werte des Vektors $y\_old[i\ mod\ 2]$ ($d(f)$ ist die Zugriffsdistanz siehe Anhang zu Bruss2D-MIX)
- $blocksize * (k-1)$ Werte von $stored\_function\_values$

Insgesamt also:

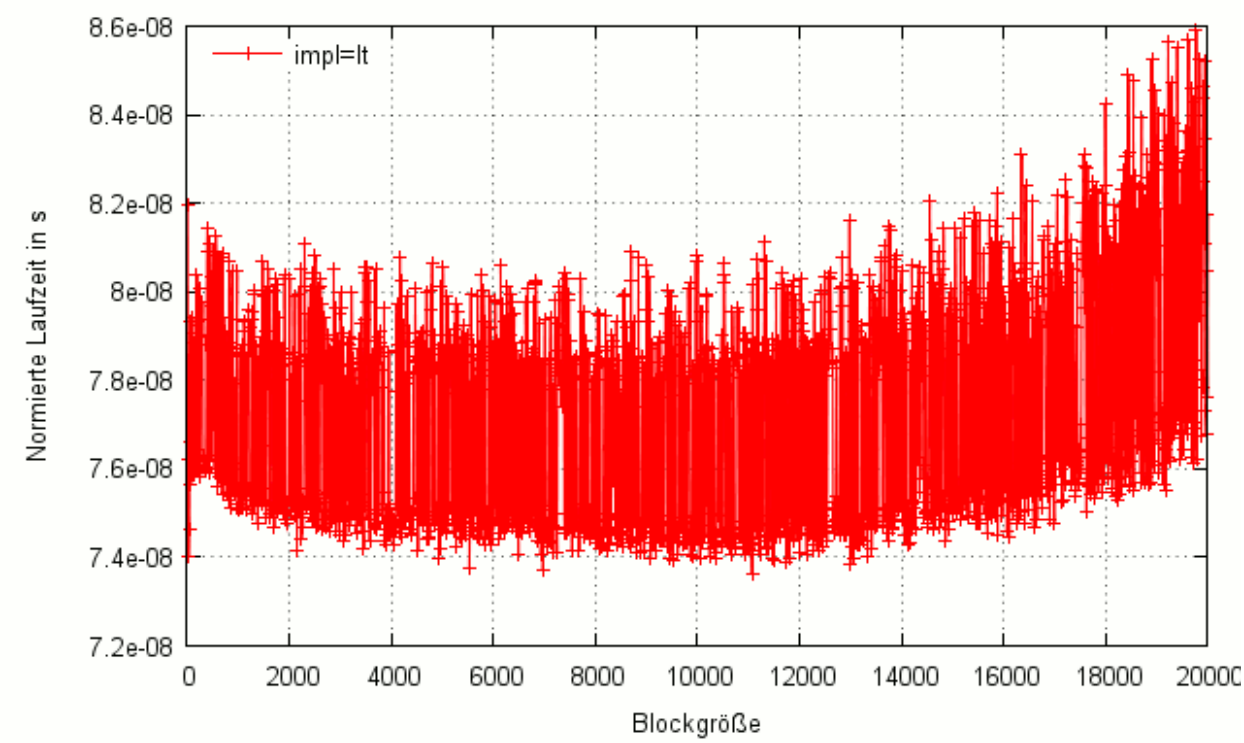
$$k + blocksize + blocksize + 2 * d(f) + blocksize * (k-1) = \\ k + 2 * d(f) + blocksize * (k+1) \qquad (10)$$

Werte.
Die größte optimale Blockgröße berechnet sich also nach der Formel:

$$cachesize = (k + 2 * d(f) + blocksize * (k+1)) * sizeof(datatype)$$

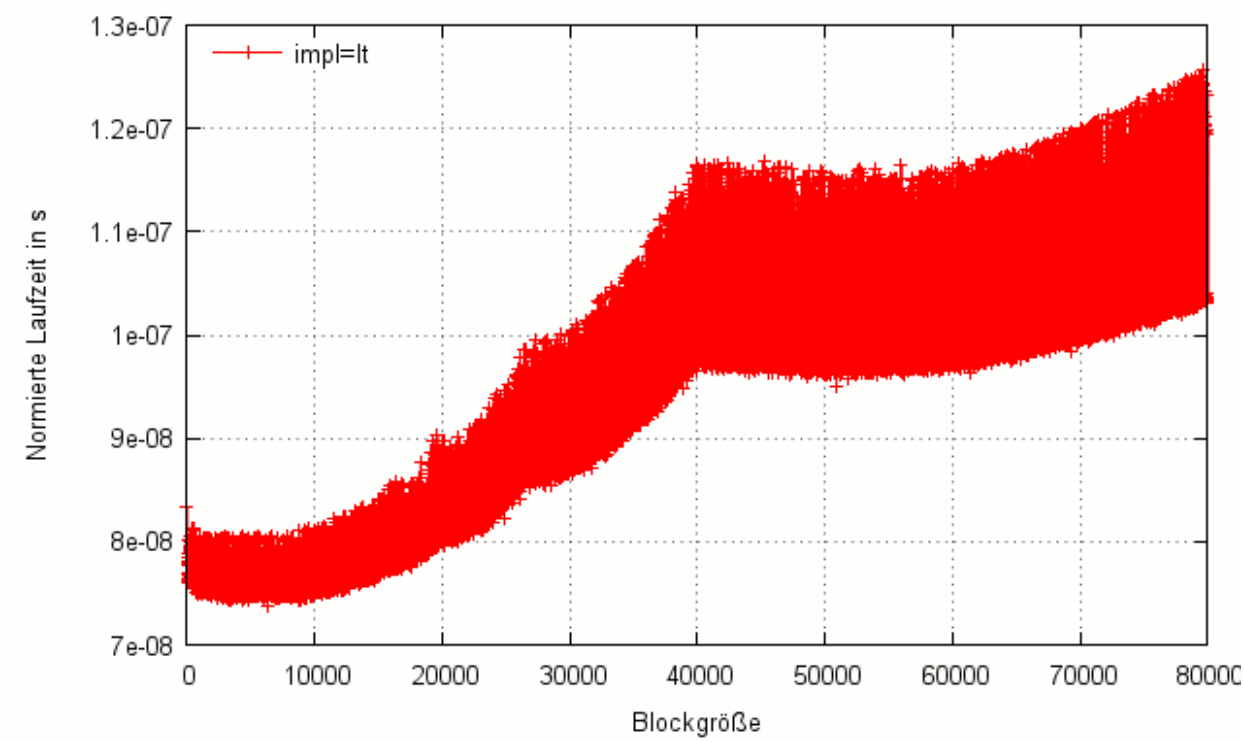
$$blocksize = \frac{\frac{cachesize}{sizeof(datatype)} - (k + 2 * d(f))}{k+1} \qquad (11)$$

Wir schauen uns dazu ein paar Messdaten an:



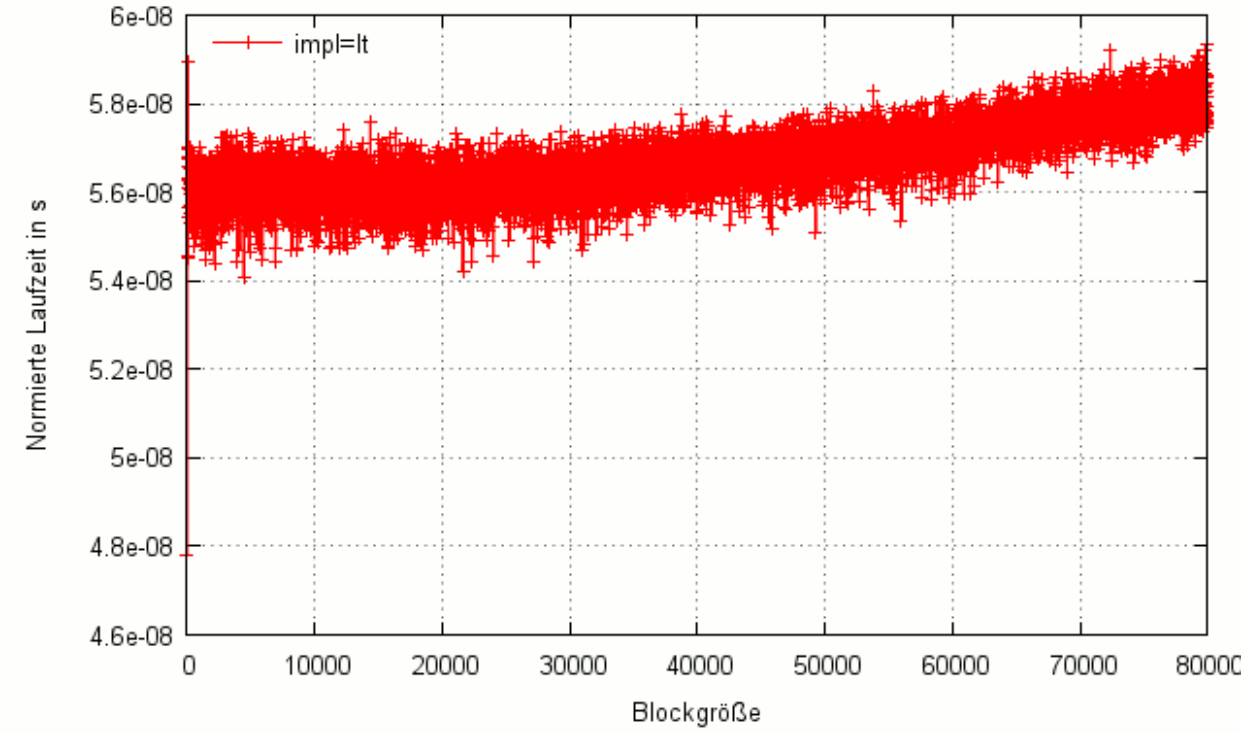
Messreihe 7: Normierte Laufzeit in Abhängigkeit von der Blockgröße für n=20000, d(f)=100, k=6, cachesize=512KByte auf Rechner 1.

Wir sehen ab ca. 10000 einen Anstieg der Laufzeit. Wenn wir nachrechnen, ergibt sich nach (11) für die größte optimale Blockgröße ein Wert von ca. 9333.
Deutlicher werden die Laufzeitunterschiede für größere n :



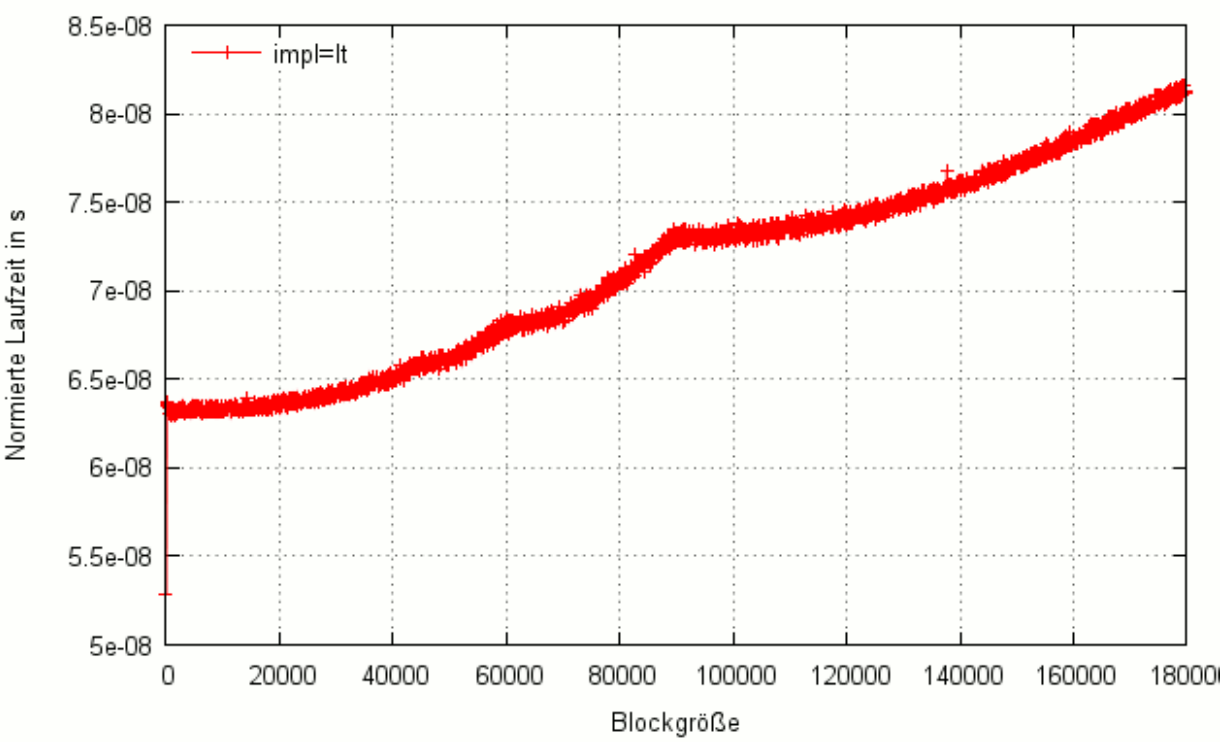
Messreihe 8: Normierte Laufzeit in Abhängigkeit von der Blockgröße für n=80000, d(f)=200, k=6, cachesize=512KByte auf Rechner 1.

Die theoretisch größte optimale Blockgröße für Messreihe 7 wäre 9304.



Messreihe 9: Normierte Laufzeit in Abhängigkeit von der Blockgröße für n=80000, d(f)=200, k=6, cachesize=3072 MB auf Rechner 2

Die theoretisch größte optimale Blockgröße für Messreihe 8 wäre 56115.



Messreihe 10: Normierte Laufzeit in Abhängigkeit von der Blockgröße für n=180000, d(f)=300, k=6, cachesize=3072 MB auf Rechner 2

Die theoretisch größte optimale Blockgröße für Messreihe 9 wäre 57522.
An den Messreihen 9 und 10 sieht man, dass die Formel (11) nur als obere Abschätzung zu gebrauchen ist. In der Praxis ist die Blockgröße oft deutlich kleiner zu wählen.

2.8 Implementierung mit Pipelining

Mit dieser Variante wird der „Pipelining-Algorithmus“ aus der Dissertation „Effiziente Implementierung eingebetteter Runge-Kutta-Verfahren durch Ausnutzung der Speicherzugriffslokalität“[1] auf das Adams-Bashforth-Verfahren übertragen.

Die Idee dabei ist, dass es für die Speicherzugriffslokalität am besten wäre, wenn die Differentialgleichung komponentenweise berechnet werden könnte. Dann würde der Algorithmus sofort im nächsten Berechnungsschritt wieder auf die zuletzt berechneten Daten zugreifen und die Daten wären mit hoher Wahrscheinlichkeit noch im Cache. Wie bereits erwähnt, ist das im Allgemeinen nicht möglich, weil bei der Berechnung einer neuen Approximation $\tilde{y}(t_{i+1})$ die Komponentenfunktionen $f_l(t, \tilde{y})$ auf alle Komponenten des Vektors des vorherigen Zeitschritts $\tilde{y}(t_i)$ zugreifen kann. Man ist also gezwungen, zuerst alle Komponenten des neuen Vektors auszurechnen, bevor man mit dem nächsten Zeitschritt fortfahren kann.

Die nachfolgende Grafik illustriert die bisherige Berechnungsreihenfolge des Programms; dabei ist $k=3$ und $n=12$. Vom Adams-Bashforth-Verfahren werden 5 Zeitschritte berechnet. Die ersten 3 Zeitschritte sind vorher berechnete Anfangswerte.

	$\tilde{y}_0$	$\tilde{y}_1$	$\tilde{y}_2$	$\tilde{y}_3$	$\tilde{y}_4$	$\tilde{y}_5$	$\tilde{y}_6$	$\tilde{y}_7$	$\tilde{y}_8$	$\tilde{y}_9$	$\tilde{y}_{10}$	$\tilde{y}_{11}$
t_0												
t_1												
t_2												
t_3	1	2	3	4	5	6	7	8	9	10	11	12
t_4	13	14	15	16	17	18	19	20	21	22	23	24
t_5	25	26	27	28	29	30	31	32	33	34	35	36
t_6	37	38	39	40	41	42	43	44	45	46	47	48
t_8	49	50	51	52	53	54	55	56	57	58	59	60

Es gibt allerdings Differentialgleichungen mit sogenannter beschränkter Zugriffsdistanz. Bei ihnen greifen die Komponentenfunktionen $f_l(t, \tilde{y})$ nur auf eine Untermenge $\{\tilde{y}_{j-b}, \tilde{y}_{j-b+1}, \ldots, \tilde{y}_j, \ldots, \tilde{y}_{j+b-1}, \tilde{y}_{j+b}\}$ der Komponenten des Vektors $\tilde{y}$ zu. b nennt man auch die Zugriffsdistanz der Komponentenfunktion geschrieben als $d(f_l)$.

$d(f)$ hingegen ist die Zugriffsdistanz von f . Definitionsgemäß ist sie die maximal auftretende Zugriffsdistanz aller Komponentenfunktionen f_l .

Ist $d(f)$ bekannt, so ist es möglich eine andere Berechnungsreihenfolge mit höherer Speicherzugrifflokalität zu konstruieren, das sogenannte „Pipeline-Schema“.

Es besteht aus drei Phasen. In der ersten Initalisierungsphase wird ein oberes linkes Dreieck berechnet. In der Sweep-Phase werden die Elemente in einem diagonalen Schema bis zur letzten Komponente berechnet. In der abschließenden Finalisierungsphase wird das übrig gebliebene untere rechte Dreieck berechnet.

Dieses Schema wird nicht auf die Elemente direkt angewendet, sondern auf Blöcke, die mehrere aufeinander folgende Komponenten eines Zeitschritts zusammenfassen.

Die Blockgröße $blocksize$ zusammen mit der Pipelinelänge $pipesize$ muss so gewählt werden, dass der Cache möglichst gut ausgenutzt wird. Um die Abhängigkeiten durch die Zugriffsdistanz zu berücksichtigen, wählen wir eine Blockgröße von mindestens $d(f)$.

Das folgende Beispiel arbeitet mit einer Pipelinelänge von $pipesize = 4$, bei $k = 3$.

Block	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
t_0																									
t_1																									
t_2																									
t_3	1	2	4	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63	67	71	75	79	83	87	91
t_4	3	5	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	95
t_5	6	9	13	17	21	25	29	33	37	41	45	49	53	57	61	65	69	73	77	81	85	89	93	96	98
t_6	10	14	18	22	26	30	34	38	42	46	50	54	58	62	66	70	74	78	82	86	90	94	97	99	100
t_7	1	2	4	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63	67	71	75	79	83	87	91
t_8	3	5	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	95
t_9	6	9	13	17	21	25	29	33	37	41	45	49	53	57	61	65	69	73	77	81	85	89	93	96	98
t_{10}	10	14	18	22	26	30	34	38	42	46	50	54	58	62	66	70	74	78	82	86	90	94	97	99	100
t_{11}	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48
t_{12}	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	50

Initalisierungsphase Sweep-phase Finalisierungsphase

Der zugehörige Pseudo-Code sieht wie folgt aus:

```
// Berechnungen für einen Block
bkstage(l_0, i) {
  l_e := min(l_0 + blocksize, n);
  for (m := l_0; m < l_e; m := m + 1)
    y_old[(i+1) mod 2][m] := b_{k-1} * stored_function_value[i mod k][m];

  for (j := 1; j < k - 1; j := j + 1)
  {
    for (m := l_0; m < l_e; m := m + 1)
      y_old[(i+1) mod 2][m] += b_j * stored_function_values[(i - j) mod k][m];
  }
  for (m := l_0; m < l_e; m := m + 1)
    stored_function_values[i mod k][m] := f_l(t_0 + i * h, y_old[i mod 2]);

  for (m := l_0; m < l_e; m := m + 1)
    y_old[(i+1) mod 2][m] := b_0 * stored_function_value[i mod k][m];

  for (m := l_0; m < l_e; m := m + 1)
    y_old[(i+1) mod 2][m] = y_old[i mod 2][m] + h * y_old[(i+1) mod 2][m];
}
finalisationstart = ⌊ n / blocksize ⌋ * blocksize;
if (finalisationstart == n) finalisationstart = n - blocksize;
for (i := k - 1; i < H/h - 1; i := endOfPipe) {

  endOfPipe := min(i + pipesize, H/h - 1);
  pipesize := endOfPipe - i;

  // Initialisierungsphase
  for (j := blocksize; j ≤ pipesize * blocksize; j := j + blocksize) {
    l_0 := j, m := i;
    while (l_0 > 0) {
      l_0 := l_0 - blocksize;
      bkstage(l_0, m);
      m := m + 1;
    }
  }
  // Sweepphase
  for (l_0 := pipesize * blocksize; l_0 < n - blocksize; l_0 := l_0 + (pipesize + 1) * blocksize) {
    for (m := i; m < endOfPipe; m := m + 1) {
      bkstage(l_0, m);
      l_0 := l_0 - blocksize;
    }
  }
  // Finalisierungsphase
  for (j := i; j < endOfPipe; j := j + 1)
    for (m := j, l_0 := finalisationstart; m < endOfPipe; m := m + 1, l_0 := l_0 - blocksize)
      bkstage(l_0, m);
}
```

Pseudocode 6: Implementierung "Pipe" mit Pipelining

2.9 Vergleich der Laufzeiten von „lt“ und „Pipe“

Wir wollen nun untersuchen, ob die Pipelining-Variante „Pipe“ gegenüber der loop-tiling Variante „lt“ Vorteile hinsichtlich der Ausführungsgeschwindigkeit hat.

Dazu müssen wir zunächst die Blockgröße $blocksize$ und die Länge der Pipeline $pipesize$ wählen. Wir gehen dabei ähnlich vor wie bei der Wahl der Blockgröße bei der loop-tiling Variante „lt“. Wir schauen, wie viele Daten zur Berechnung der Diagonalen beim Pipelining-Schema abhängig von der Blockgröße und Pipelinelänge gebraucht werden. Anschließend setzen wir die Blockgröße und Pipelinelänge dann so, dass diese Daten in den Cache passen.

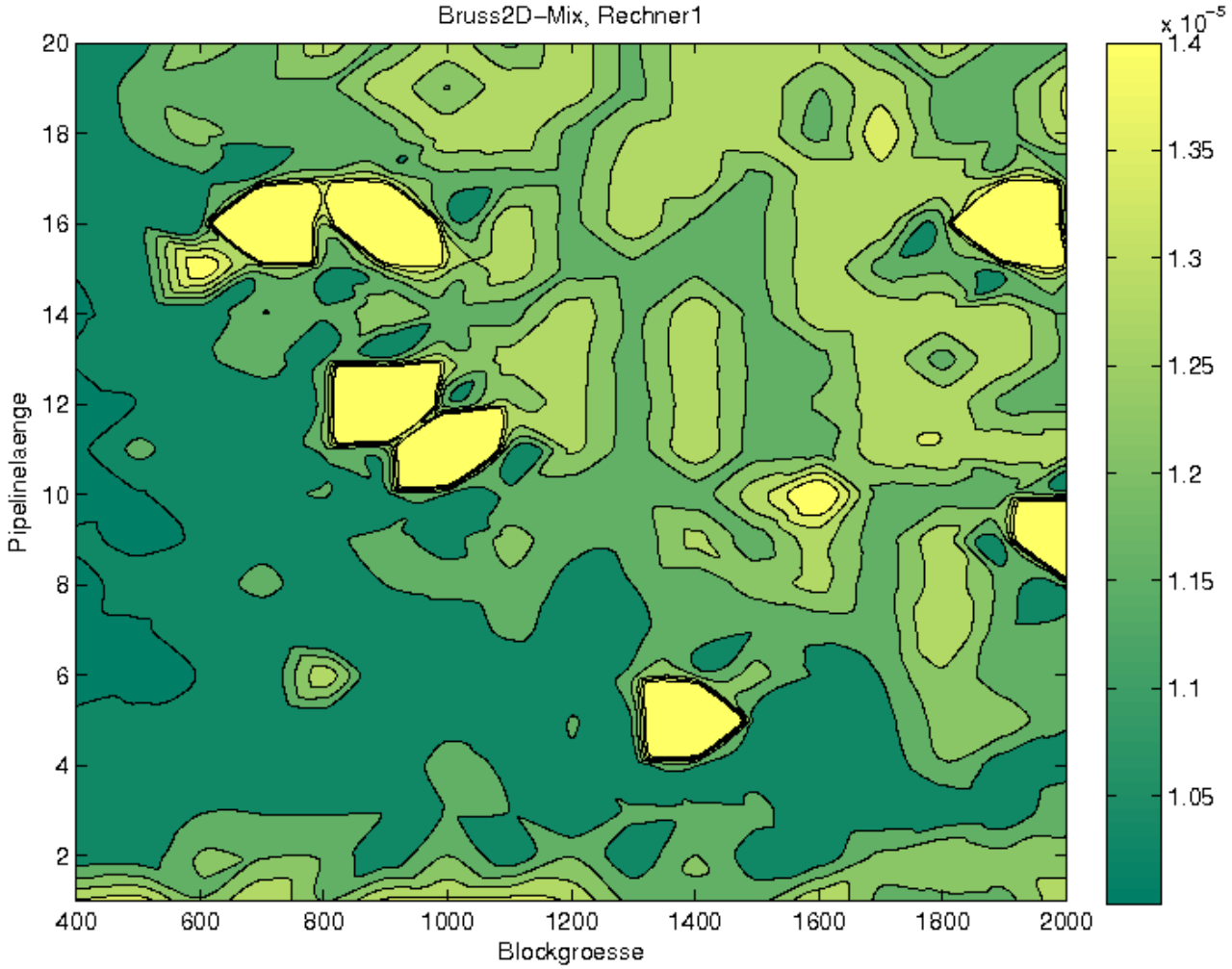
Da bei der Berechnung eines Blockes beim Pipelining genau das gleiche gemacht wird wie beim loop-tiling, können wir die Formel (10) verwenden, um den Speicherbedarf dafür zu bestimmen. Eine Diagonale des Pipeline-Schemas hat $pipesize$ Blöcke, d.h. zur Berechnung einer Diagonalen werden

$$pipesize*(k+2*d(f)+blocksize*(k+1)) \tag{12}$$

Werte verwendet.
Das heißt Blöckgröße und Pipelinelänge müssen so gewählt werden, dass

$$cachesize = pipesize*(k+2*d(f)+blocksize*(k+1))*sizeof(datatype) \tag{13}$$

gilt.
Wie am Anfang dieses Kapitels schon erwähnt, wäre es am besten, man könnte die Differentialgleichung komponentenweise berechnen. Dieser Rechenreihenfolge kommen wir am nächsten, wenn wir die Blockgröße möglichst klein wählen. Um diese These zu stützen, machen wir eine Messreihe 11, in der wir die normierten Laufzeiten für verschiedene Blockgröße und Pipelinelängen betrachten.



Messreihe 11: Normierte Laufzeit der Implementierung "Pipe" für verschiedene Pipelinelängen und Blockgrößen auf Rechner 1 (cachesize=512KByte) mit n=80000 Komponenten bei einem Integrationsintervall von H=1 und einer Schrittweite h=0.001

Wir sehen, dass sich eine kleine Blockgröße günstig auf die Laufzeit auswirkt. Die kleinstmögliche Blockgröße ist $2*d(f)$.
Damit können wir für (13) schreiben:

$$cachesize = pipesize*(k+2*d(f)+2*d(f)*(k+1))*sizeof(datatype) \tag{14}$$

Eine Faustregel für die Pipelinelänge wäre also:

$$pipesize = \frac{cachesize}{(k+2*d(f)+2*d(f)*(k+1))*sizeof(datatype)} \tag{15}$$

Für unsere Messreihe 11 ergäbe das eine Pipelinelänge von 20. Wir sehen aber, dass man in der Realität besser eine deutlich geringere Pipelinelänge benutzen. Eine Pipelinelänge von 11 wäre nach Messreihe 11 bei einer Blockgröße von 400 eine gute Wahl.

3 Parallele Implementierungen

Wir wollen nun die sequentiellen Implementierungen für die Ausführung auf einem Parallelrechner mit Shared-memory anpassen.

3.1 Synchronisation mittels „Barrier“

Zunächst muss man überlegen, welche Berechnungen überhaupt parallel ausgeführt werden können. Man stellt fest, dass bei Differentialgleichungen ohne beschränkte Zugriffsdistanz nur die Komponenten eines Zeitschrittes von mehreren Threads gleichzeitig ausgerechnet werden können, da im nächsten Zeitschritt alle Komponenten des vorherigen Zeitschritts bekannt sein müssen. Bevor begonnen wird, die Komponenten eines Zeitschritts auszurechnen, muss also darauf gewartet werden, dass alle Threads mit den Berechnungen des vorigen Zeitschritts fertig sind. Einen solchen Synchronisationspunkt nennt man „Barrier“.

Auf diese Weise können die sequentiellen Implementierungen „mgF“ und „lt“ synchronisiert werden. Jeder der $threads$ Threads besitzt eine Nummer $mythreadid$ aus dem Intervall $0,1,\ldots,threads-1$,

Parallele Implementierung „mgF“:

$$
\begin{array}{l}
for(i:=k-1;\ i<\frac{H}{h}-1;\ i:=i+1) \\
\{ \\
\quad for(l:=\left\lfloor \frac{mythreadid*n}{threads} \right\rfloor;\ l<\left\lfloor \frac{(mythreadid+1)*n}{threads} \right\rfloor;\ l:=l+1) \\
\quad \{ \\
\quad\quad y\_old[(i+1)\ mod\ 2][l]:=b_{k-1}*stored\_function\_values[i\ mod\ (k-1)][l]; \\
\quad\quad for(j:=1;\ j<k-1;\ j:=j+1) \\
\quad\quad \{ \\
\quad\quad\quad y\_old[(i+1)\ mod\ 2][l]\ +=\ b_j*stored\_function\_values[(i-j)\ mod\ (k-1)][l]; \\
\quad\quad \} \\
\quad\quad stored\_function\_values[i\ mod\ (k-1)][l]:=f_l(t_0+i*h,\ y\_old[i\ mod\ 2]); \\
\quad\quad y\_old[(i+1)\ mod\ 2][l]\ +=\ b_0*stored\_function\_values[i\ mod\ (k-1)][l]; \\
\quad\quad y\_old[(i+1)\ mod\ 2][l]:=y\_old[i\ mod\ 2][l]+h*y\_old[(i+1)\ mod\ 2][l]; \\
\quad\quad barrier(); \\
\quad \} \\
\}
\end{array}
$$

Pseudocode 7: parallele Variante der Implementierung "mgF" mit Barrier

Parallele Implementierung „li“:

$$start = \left\lfloor \frac{mythreadid * n}{threads} \right\rfloor;$$
$$nextstart = \left\lfloor \frac{(mythreadid + 1) * n}{threads} \right\rfloor;$$
$$for(i := k-1;\ i < \frac{H}{h} - 1;\ i := i+1)$$
{
  $for(l := start;\ l < nextstart;\ l := l+1)$
    $y\_old[(i+1) \bmod 2][l] := b_{k-1} * stored\_function\_values[i \bmod (k-1)][l];$

  $for(j := 1;\ j < k-1;\ j := j+1)$
  {
    $for(l := start;\ l < nextstart;\ l := l+1)$
      $y\_old[(i+1) \bmod 2][l] \mathrel{+}= b_j * stored\_function\_values[(i-j) \bmod (k-1)][l];$
  }
  $for(l := start;\ l < nextstart;\ l := l+1)$
    $stored\_function\_values[i \bmod (k-1)][l] := f_l(t_0 + i*h,\ y\_old[i \bmod 2]);$

  $for(l := start;\ l < nextstart;\ l := l+1)$
    $y\_old[(i+1) \bmod 2][l] \mathrel{+}= b_0 * stored\_function\_values[i \bmod (k-1)][l];$

  $for(l := start;\ l < nextstart;\ l := l+1)$
    $y\_old[(i+1) \bmod 2][l] := y\_old[i \bmod 2][l] + h * y\_old[(i+1) \bmod 2][l];$

  $barrier();$
}

Pseudocode 8: parallele Variante der Implementierung „li" mit Barrier

Parallele Implementierung „It“:

```
for(i:=k-1; i<H/h-1; i:=i+1)
{
  start:=⌊threadid*n / threads⌋;
  nextstart:=⌊(threadid+1)*n / threads⌋;
  for(l:=start; l<nextstart; l:=l+blocksize)
  {
    endOfBlock=min(l+blocksize, nextstart);

    for(m:=l; m<endOfBlock; m:=m+1)
      y_old[(i+1)][m]:=b_{k-1}*stored_function_value[(i-1) mod (k-1)][m];

    for(j:=1; j<k-1; j:=j+1)
    {
      for(m:=l; m<endOfBlock; m:=m+1)
        y_old[(i+1) mod 2][m] += b_j*stored_function_values[(i-j) mod (k-1)][m];
    }

    for(m:=l; m<endOfBlock; m:=m+1)
      stored_function_values[i mod (k-1)][m]:=f_l(t_0+i*h, y_old[i mod 2]);

    for(m:=l; m<endOfBlock; m:=m+1)
      y_old[(i+1) mod 2][m] += b_0*stored_function_value[i mod (k-1)][m];

    for(m:=l; m<endOfBlock; m:=m+1)
      y_old[(i+1) mod 2][m]:=y_old[i mod 2][m]+h*y_old[(i+1) mod 2][m];

    barrier();
  }
}
```

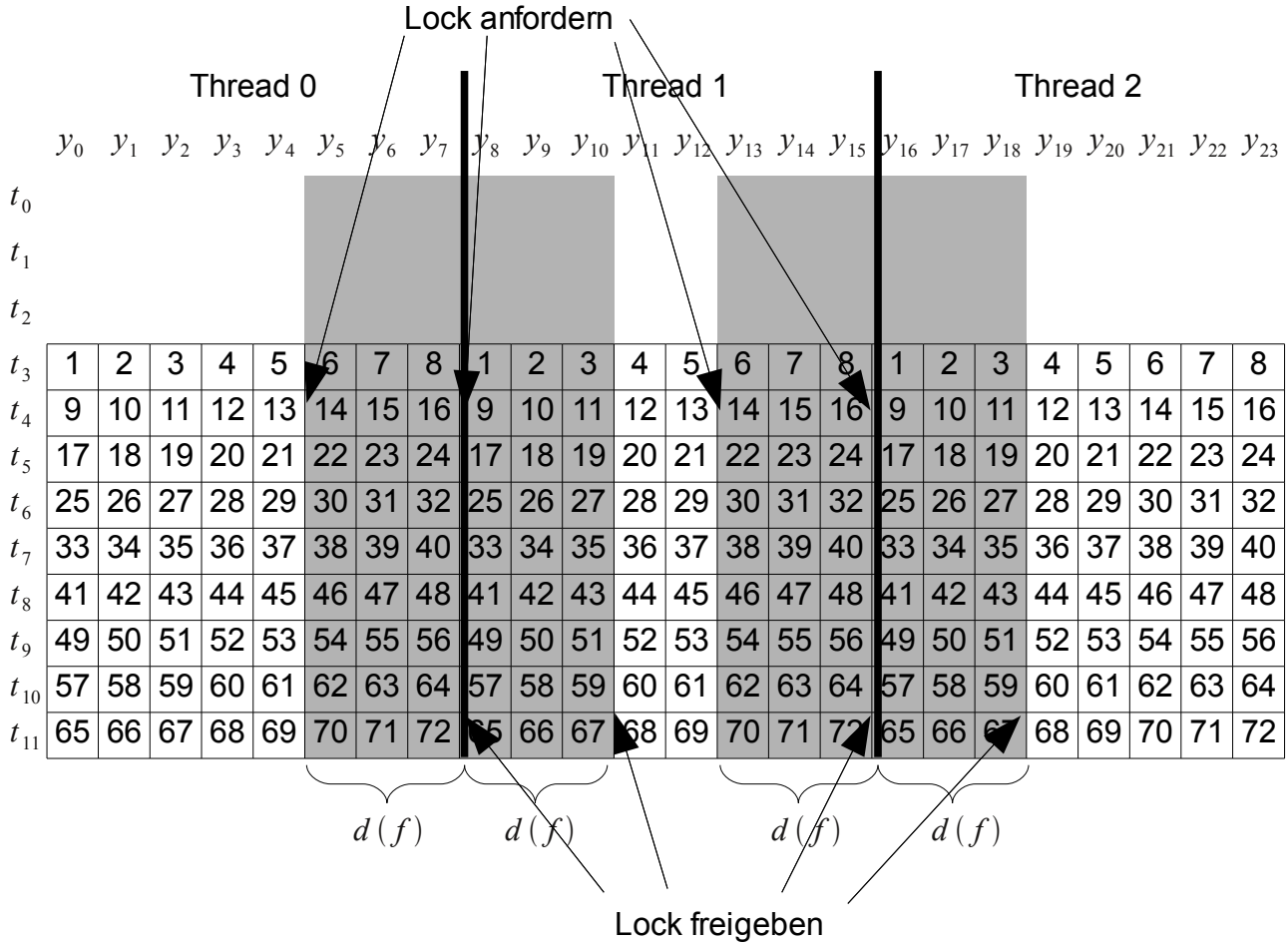
Pseudocode 9: parallele Variante der Implementierung "lt" mit Barrier

3.2 Synchronisation mittels „Locks“

Bei Differentialgleichungen mit beschränkter Zugriffsdistanz ist es nicht unbedingt notwendig, darauf zu warten, dass die Threads alle Komponenten des Zeitschritts berechnet haben. Stattdessen kann, sobald alle Komponenten der Zugriffsdistanz ausgerechnet wurden, schon mit der Berechnung des nächsten Zeitschrittes begonnen werden.

Zur Umsetzung dieser Idee verwenden wir für jeden Thread und Zeitschritt jeweils zwei Locks. Die Locks sind am Anfang alle gesperrt. Braucht ein Thread Werte, die ein anderer Thread berechnen soll, so fordert er das entsprechende Lock an. Natürlich muss der andere Thread das Lock auch frei geben, sobald er die Werte berechnet hat.

In der nachfolgenden Graphik sind die Werte wieder nach der Reihenfolge, in der sie berechnet werden, durchnummeriert. Zusätzlich sind die Zeitpunkte, zu denen Locks angefordert und entsperrt werden, eingezeichnet. Wir gehen davon aus, dass jeder Thread mindestens $2*d(f)$ Komponenten zu berechnen hat.



Die auf der Implementierung „It“ aufbauende Implementierung „LockIt“ sieht dann wie folgt aus:

```
for(i:=k-1; i<H/h-1; i:=i+1)
{
  start:=⌊threadid*n/threads⌋;
  nextstart:=⌊(threadid+1)*n/threads⌋;
  for(l:=start; l<nextstart; l:=l+blocksize)
  {
    endOfBlock:=min(l+blocksize, nextstart);
    if(i>k-1)
    {
      if(mythreadid>0 ∧ l==start)
        getRightLockFromLeftNeighbour(i-1);
      if(mythreadid<threads-1 ∧ l≤nextstart-d(f) ∧
      nextstart-d(f)<endOfBlock)
        getLeftLockFromRightNeighbour(i-1);
    }
    for(m:=l; m<endOfBlock; m:=m+1)
      y_old[(i+1)][m]:=b_{k-1}*stored_function_value[i mod (k-1)][m];

    for(j:=1; j<k-1; j:=j+1)
    {
      for(m:=l; m<endOfBlock; m:=m+1)
        y_old[(i+1) mod 2][m] += b_j*stored_function_values[(i-j) mod (k-1)][m];
    }

    for(m:=l; m<endOfBlock; m:=m+1)
      stored_function_values[i mod (k-1)][m]:=f_l(t_0+i*h, y_old[i mod 2]);

    for(m:=l; m<endOfBlock; m:=m+1)
      y_old[(i+1) mod 2][m] += b_0*stored_function_value[i mod (k-1)][m];

    for(m:=l; m<endOfBlock; m:=m+1)
      y_old[(i+1) mod 2][m]:=y_old[i mod 2][m]+h*y_old[(i+1) mod 2][m];

    if(i<H/h-2)
    {
      if(mythreadid>0 ∧ l≤start+d(f) ∧ start+d(f)<endOfBlock)
        unlockLeftMutex(i);
      if(mythreadid<threads-1 ∧ end==nextstart)
        unlockRightMutex(i);
    }
  }
}
```

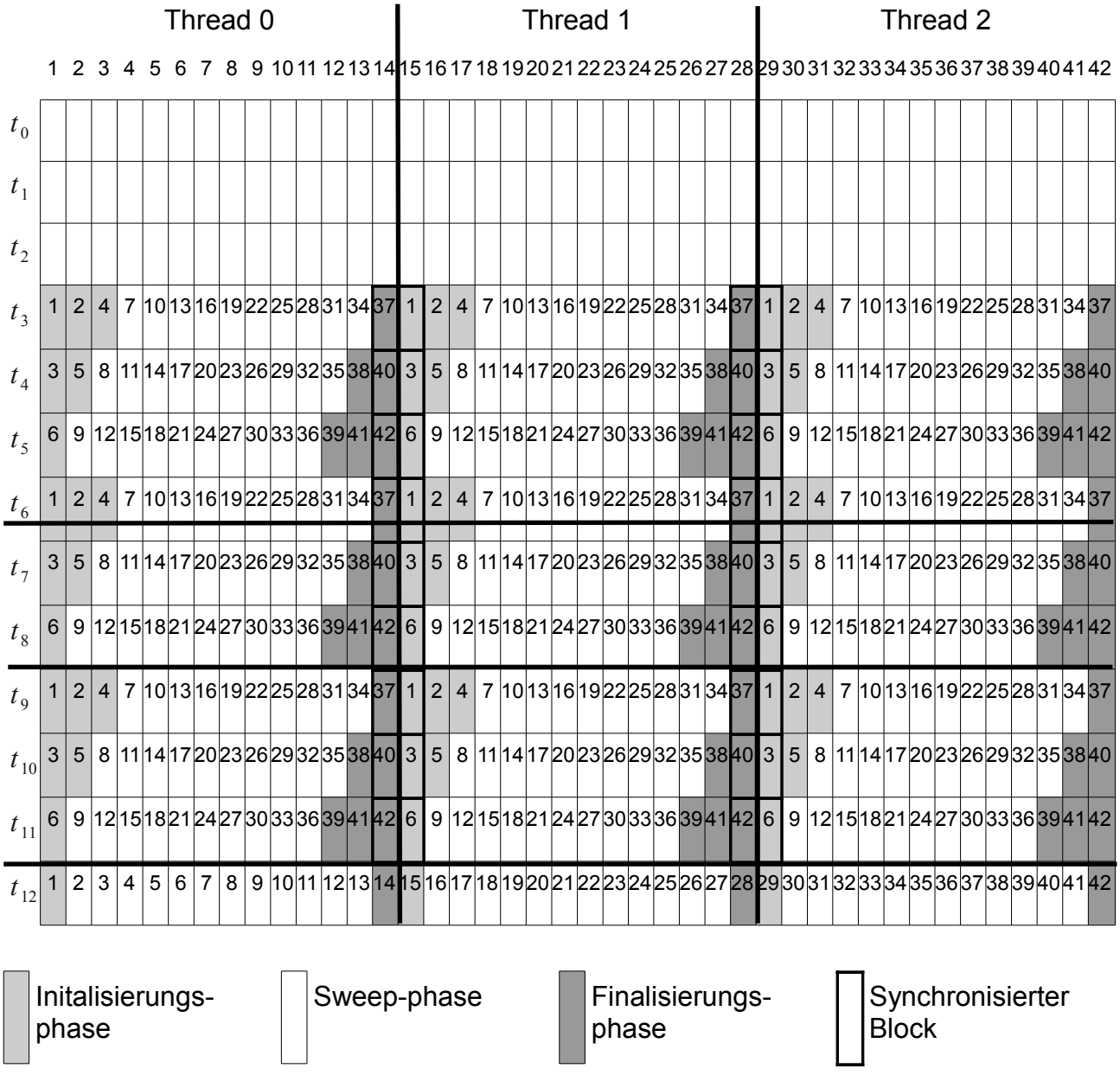
Pseudocode 10: parallele Variante "Locklt" von "lt" mit Locks

Mit Locks lässt sich auch die Implementierung „Pipe“ synchronisieren.

Es gilt weiterhin, dass zur Berechnung einer Komponente die Komponenten der Zugriffsdistanz des vorhergehenden Zeitschritts nötig sind. Da die Blöcke bei der Variante „Pipe“ schon mindestens die Größe der Zugriffsdistanz haben, ist zur Berechnung eines Blockes der entsprechende Block des vorhergehenden Zeitschritts und dessen benachbarte Blöcke nötig.

Für die Berechnung des ersten bzw. letzten Blockes eines Threads in einem Zeitschritt bedeutet das, dass ein Block vom benachbarten Thread gebraucht wird. Um zu gewährleisten, dass dieser Block schon berechnet wurde, verwenden wir, ähnlich wie bei der vorhergehenden Variante „LockIt“, Locks. Die Locks sind am Anfang gesperrt. Sobald der zugehörige Block berechnet wurde, wird das Lock freigegeben. Wenn ein Thread einen Block braucht, den ein anderer Thread berechnen soll, muss er das entsprechende Lock anfordern.

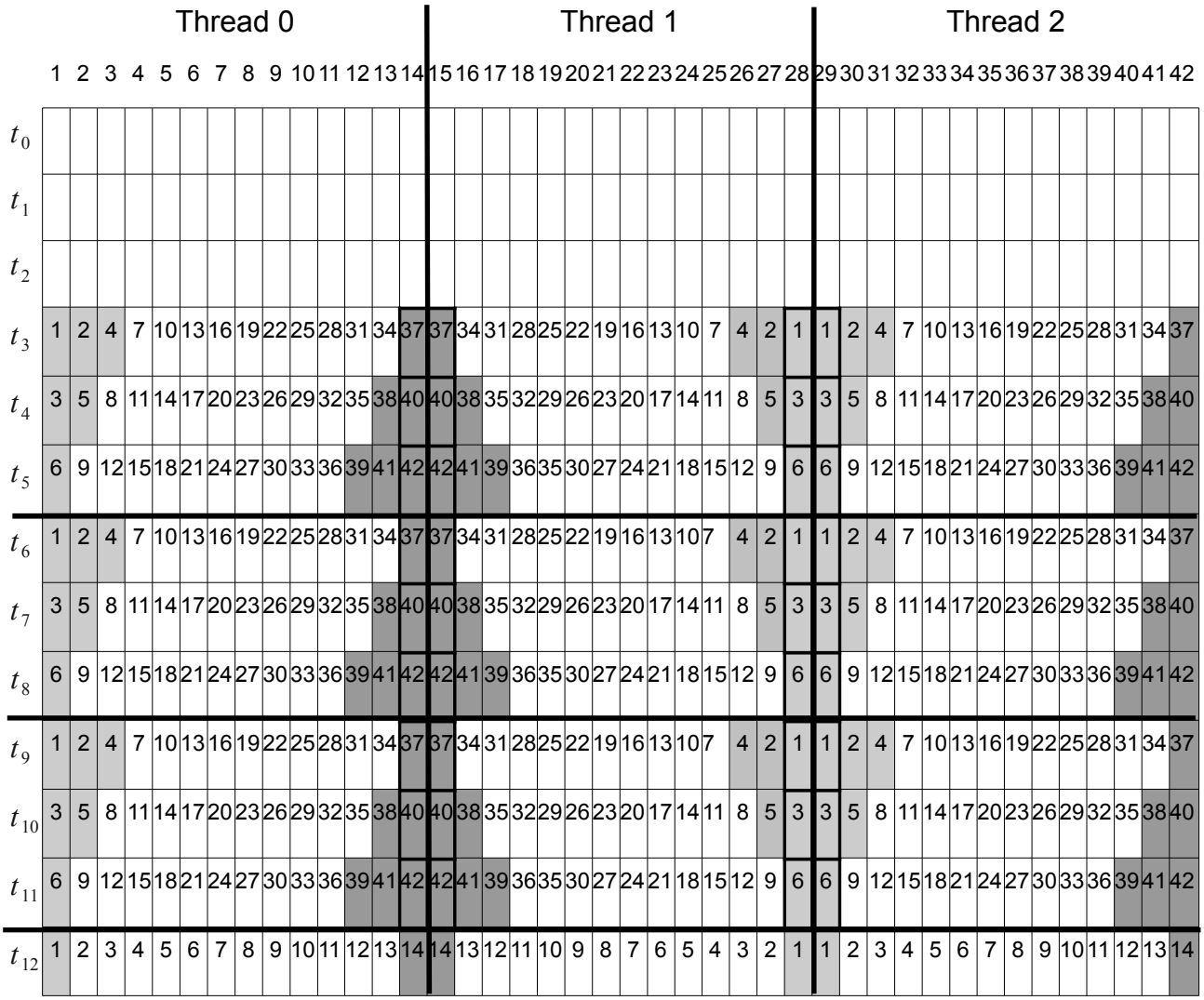
Wir schauen uns dazu ein Beispiel mit 3 Threads und einer Pipelinelänge von 3 an.



```
start := ⌊threadid * n / threads⌋;
nextstart := ⌊(threadid + 1) * n / threads⌋;
finalisationstart = ⌊nextstart − start / blocksize⌋ * blocksize + start;
if ( finalisationstart = = nextstart ) finalisationstart = nextstart − blocksize;
for(i := k − 1; i < H/h − 1; i := endOfPipe) {
  endOfPipe := min(i + pipesize, H/h − 1);
  pipesize := endOfPipe − i;

  // Initialisierungsphase
  for( j := start + blocksize; j ≤ start + pipesize; j := j + blocksize) {
    l0 := j, m := i;
    while(l0 > 0) {
      l0 := l0 − blocksize;
      if (l0 = = start) getRightLockFromLeftNeighbour(m);
      bkstage(l0, m);
      m := m + 1;
    }
    unlockLeftLock(m − 1);
  }
  // Sweepphase
  for(l0 := start + pipesize * blocksize; l0 < nextstart − blocksize;
  l0 := l0 + (pipesize + 1) * blocksize) {
    for(m := i; m < endOfPipe; m := m + 1) {
      bkstage(l0, m);
      l0 := l0 − blocksize;
    }
  }
  // Finalisierungsphase
  for( j := i; j < endOfPipe; j := j + 1){
    getLeftLockFromRightNeighbour(i);
    for(m := j, l0 := finalisationstart; m < endOfPipe; m := m + 1, l0 := l0 − blocksize) {
      bkstage(l0, m);
      if (l0 = = finalisationstart) unlockRightLock(m);
  }
}
```

Pseudocode 11: parallele Variante von "Pipe" mit Locks

Ungünstig an der Implementierung „Pipe“ ist, dass ein Thread evtl. lange darauf warten muss, dass der benachbarte Thread seine ganze Pipeline berechnet hat, bevor er beginnen kann. Zumindest muss in unserem Beispiel für „Pipe“ in der ersten Pipeline der Thread 1 auf Thread 0 und Thread 2 auf Thread 1 warten.
Es könnte also günstiger sein, wenn benachbarte Threads in unterschiedlichen Richtungen rechnen. Das bedeutet dann, einer in Richtung aufsteigender Komponenten und der andere in absteigender Richtung, weil dann die Zeit zwischen Freigabe und Anforderung der Locks kürzer ist.
Wir nennen diese Variante „Pipek“. Das Beispiel mit den 3 Threads und Pipelinelänge 3 sieht dann so aus:



Der Pseudo-Code für „Pipek“:

$$start := \left\lfloor \frac{threadid * n}{threads} \right\rfloor;$$
$$nextstart := \left\lfloor \frac{(threadid+1) * n}{threads} \right\rfloor;$$
$$if\,(threadid \bmod 2 == 0)\{$$
$$\quad for\,(i := k-1;\, i < \frac{H}{h} - 1;\, i := endOfPipe)\ \{$$
$$\quad\quad endOfPipe := min(i + pipesize,\, \frac{H}{h} - 1);$$
$$\quad\quad pipesize := endOfPipe - n;$$
$$\quad\quad //\ Initialisierungsphase$$
$$\quad\quad for(j := start + blocksize;\, j \le start + pipesize * blocksize;\, j := j + blocksize)\ \{$$
$$\quad\quad\quad l_0 := j,\, m := i;$$
$$\quad\quad\quad while(l_0 > 0)\ \{$$
$$\quad\quad\quad\quad l_0 := l_0 - blocksize;$$
$$\quad\quad\quad\quad if\,(l_0 == start)\ \ getRightLockFromLeftNeighbour(m);$$
$$\quad\quad\quad\quad bkstage(l_0, m);$$
$$\quad\quad\quad\quad m := m + 1;$$
$$\quad\quad\quad \}$$
$$\quad\quad \}$$
$$\quad\quad //\ Sweepphase$$
$$\quad\quad for(l_0 := start + pipesize * blocksize;\, l_0 < nextstart - blocksize;$$
$$\quad\quad l_0 := l_0 + (pipesize + 1) * blocksize)\ \{$$
$$\quad\quad\quad for(m := i;\, m < endOfPipe;\, m := m + 1)\ \{$$
$$\quad\quad\quad\quad bkstage(l_0, m);$$
$$\quad\quad\quad\quad l_0 := l_0 - blocksize;$$
$$\quad\quad\quad \}$$
$$\quad\quad \}$$
$$\quad\quad finalisationstart = \left\lfloor \frac{nextstart - start}{blocksize} \right\rfloor * blocksize + start;$$
$$\quad\quad if\,(finalisationstart == nextstart)\ finalisationstart = nextstart - blocksize;$$
$$\quad\quad //\ Finalisierungsphase$$
$$\quad\quad for(j := i;\, j < endOfPipe;\, j := j + 1)\ \{$$
$$\quad\quad\quad getLockFromRightNeighbour(j);$$
$$\quad\quad\quad for(m := j,\, l_0 := finalisationstart;\, m < endOfPipe;\, m := m + 1, l_0 := l_0 - blocksize)\ \{$$
$$\quad\quad\quad\quad bkstage(l_0, m);$$
$$\quad\quad\quad\quad if\,(l_0 == finalisationstart)\ \ unlockRightLock(m);$$
$$\quad\quad\quad \}$$
$$\quad \}$$

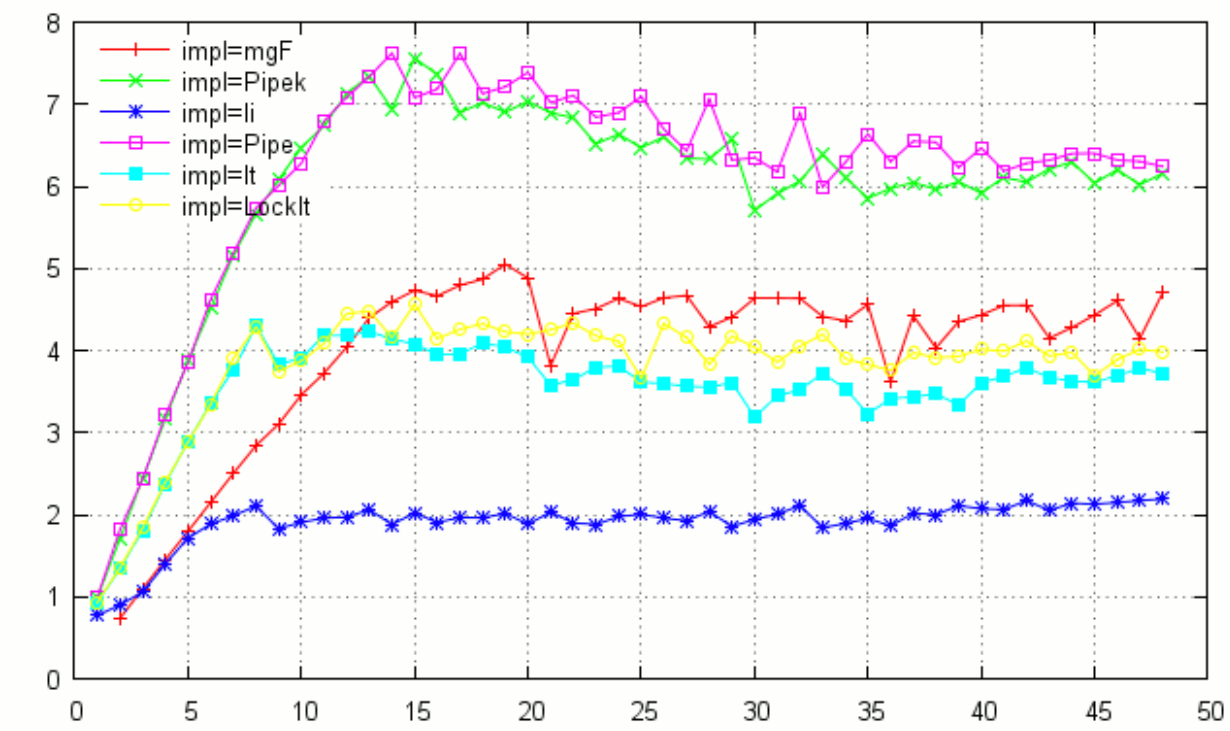
```
} else {
  for(i:=k−1; i<H/h−1; i:=endOfPipe) {
    endOfPipe:=min(i+pipesize, H/h−1);
    pipesize:=endOfPipe−n;
    initalisationstart=⌊(nextstart−start)/blocksize⌋∗blocksize+start;
    if(initalisationstart==nextstart)initalisationstart=nextstart−blocksize;

    // Initialisierungsphase
    for(j:=initalisationstart; j≥nextstart−pipesize; j:=j−blocksize) {
      l0:=j, m:=i;
      while(l0<nextstart) {
        if(l0==nextstart−blocksize) getLeftLockFromRightNeighbour(m);
        bkstage(l0, m);
        l0:=l0+blocksize;
        m:=m+1;
      }
      unlockRightLock(m−1);
    }
    // Sweepphase
    for(l0:=nextstart−(pipesize+1)∗blocksize; l0≥start;
    l0:=l0+(pipesize+1)∗blocksize) {
      for(m:=i; i<endOfPipe; m:=m+1) {
        bkstage(l0,m);
        l0:=l0+blocksize;
      }
    }
    // Finalisierungsphase
    for(j:=i; j<endOfPipe; j:=j+1){
      getRightLockFromLeftNeighbour(j);
      for(m:=j, l0:=start; m<endOfPipe; m:=m+1, l0:=l0+blocksize) {
        bkstage(l0,m);
        if(l0==start) unlockLeftLock(m);
    }
  }
}
```

Pseudocode 12: parallele Variante von "Pipek" mit Locks und konvergierenden Berechnunsreihenfolgen

3.3 Vergleich der parallelen Varianten

Im Folgenden wollen wir uns in Messreihe 12 anschauen, wie unsere parallelen Varianten auf Rechner 3 mit mehreren Threads skalieren. Der Speedup ist dabei relativ zum langsamsten sequentiellen Programmlauf einer Implementierung angegeben und bezieht sich nicht auf die eigene sequentielle Laufzeit der jeweiligen Implementierung. Das hat

den Vorteil, dass wir auch Vergleiche zwischen den Implementierungen machen können.



Messreihe 12: Speedup der parallelen Implementierungen "mgF", "li", "lt", "Locklt", "Pipe" und "Pipek" relativ zur Laufzeit der langsamsten sequentiellen Programmausführung einer Implementierung bei einem Differentialgleichungssystem mit n = 5120000 Komponenten, einem Integrationsintervall von H =0.01 und einer Schrittweite h=0.00001 auf Rechner 3. Des weiteren wurde für die Varianten „lt“ und „Locklt“ blocksize=1600 verwendet und für „Pipe“ und „Pipek“ blocksize=3200 und pipesize=3.

Wir sehen, dass die Implementierungen „Pipe“ und „Pipek“ mit „Pipelining“ am schnellsten sind und den größten Speedup aufweisen. Die loop-tiling Varianten „lt“ und „Locklt“ und die einfache Variante „mgF“ mit gespeicherten Funktionsauswertungen befinden sich im Mittelfeld. Die Variante „li“ ist am langsamsten.

Das entspricht genau der Reihenfolge der Speicherzugriffslokalitäten.

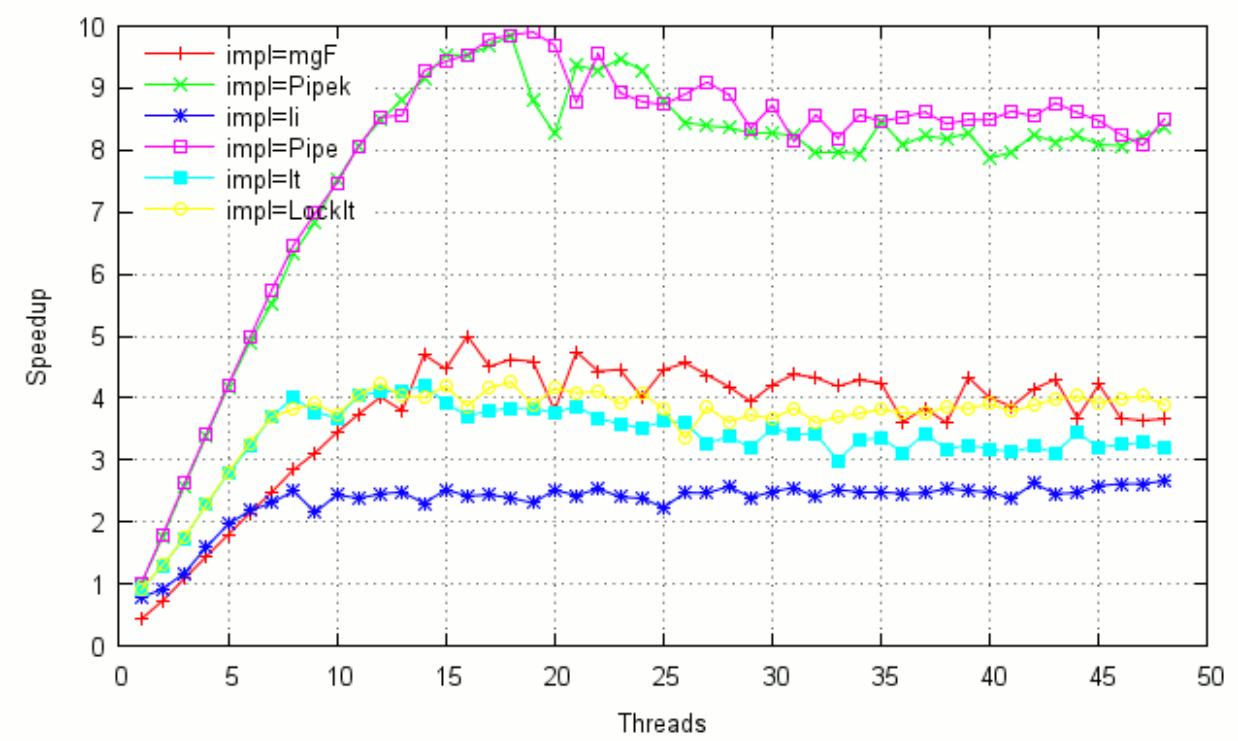
In den Schleifen von Variante „li“ wird mehrmals über einen großen Speicherbereich iteriert. „li“ hat die geringste Speicherzugriffslokalität. Die Speicherzugriffslokalität der loop-tiling Varianten „lt“ und „Locklt“ sowie der Variante „mgF“ ist besser, da hier nacheinander kleine Teilbereiche des Speichers abgearbeitet werden. Die Pipelining Varianten haben die höchste Speicherzugriffslokalität, da die Teilbereiche, die hier nacheinander berechnet werden, noch näher beieinander liegen.

Die Synchronisationmechanismen, die bei den verschiedenen Implementierungen eingesetzt wurden, scheinen keinen großen Einfluss auf die Ausführungszeiten zu haben.

Die Variante „Locklt“, die Locks zur Synchronisation verwendetet, ist nicht wesentlich schneller als Variante „lt“, die mit einer Barrier arbeitet. Die Variante „Pipek“, bei der die Threads in gegenläufiger Richtung die Komponenten der Vektoren berechnen, ist sogar für eine größere Anzahl an Threads etwas langsamer als die Variante „Pipe“, bei der die Threads in die gleiche Richtungen arbeiten. Insgesamt sind die Unterschiede aufgrund des Synchronisationsmechanismus hier aber gering im Vergleich zum Einfluss der Speicherzugriffslokalität.

Außerdem können wir für die einzelnen Implementierungen jeweils eine klare Grenze erkennen, ab der es sich nicht mehr lohnt, mehr Threads zu verwenden, weil die Wartezeiten, die durch die Synchronisation von mehr Threads entstehen, höher sind als die Zeit, die durch das parallele Rechnen von mehr Threads eingespart wird. Bei den Pipelining Varianten liegt diese bei etwa bei 14 Threads. Die „mgF“ Implementierung hat bei ca. 19 Threads ihre maximale Ausführunggeschwindigkeit erreicht und die loop-tiling Varianten werden ab 8 Threads nicht mehr schneller.

Ähnliches gilt für Messreihe 13, in der der Speedup bei einer Differentialgleichung mit weniger Komponenten dargestellt ist.



Messreihe 13: SpeedUp in Abhängigkeit von der Anzahl der verwendenten Threads für die parallelen Varianten von „mgF“, „li“, „lt“, „Locklt“, „Pipe“ und „Pipek“ für ein Differentialgleichungssystem mit n=1280000 Komponenten und einem Integrationsintervall von H=0.01 und einer Schrittweite h=0.0001 auf Rechner 3. Des weiteren wurde für die Varianten „lt“ und „Locklt“ blocksize=1600 verwendet und für „Pipe“ und „Pipek“ blocksize=3200 und pipesize=3.

4 Fazit

Wir haben gesehen, dass man durch Änderungen an der Berechnungreihenfolge des Adams-Bashforth-Verfahrens, die Speicherzugriffslokalität gegenüber einer direkten Umsetzung der Rechenvorschrift des Verfahrens, erhöhen kann.
Dies beschleunigt die sequentielle Ausführung des Adams-Bashforth-Verfahrens. Aber noch deutlicher ist der positive Effekt bei der Skalierbarkeit auf Multicore-Prozessor-Architekturen zu beobachten.
Trotzdem stößt man hier an eine Grenze, ab der sich das Verfahren durch Hinzunahme von mehr Prozessoren nicht mehr beschleunigen lässt. Es bleibt offen, ob man in Zukunft durch Verbesserungen in der Architektur von Multicore-Systemen noch bessere Ergebnisse erzielen kann oder ob es noch weitere Aspekte gibt, die die Ausführungszeiten beschleunigen können.

Verwendete Rechnersysteme

Bezeichnung im Text	**Rechner 1**	**Rechner 2**	**Rechner 3**
Hersteller	Fujitsu-Siemens	Intel	
Name	SCENIC W600	S7000 FC4URE	
Arbeitsspeicher			
Größe	1 GB	16 GB	128 GB
Prozessoren			
Anzahl	1	4	4
Name	Intel Pentium IV	Intel Xeon E7330	AMD Opteron 6172
Kerne pro Prozessor	1	4	12
Taktfrequenz	3 GHz	2,4 GHz	2,1 GHz
L1-Cache (Daten)			
Größe	8 KByte	32 KByte	64 KByte
Assoziativität	4-fach	8-fach	
Zeilengröße	64 Byte	64 Byte	
Latenz	2 Takte	3 Takte	
L1-Cache (Instruktionen)			
Größe	12 μ -Operationen	32 KByte	64 KByte
Assoziativität	8-fach	8-fach	
Zeilengröße	-	-	
Latenz	-	-	
L2-Cache (Unified)			
Größe	512 KByte	6 MByte 3 MByte (2 Kerne zusammen)	512 KByte
Assoziativität	8-fach	12-fach	
Zeilengröße	64 Byte	64 Byte	
Latenz	7 Takte	15 Takte	
L3-Cache(Unified)			
Größe	-	-	12 MByte 6 MByte (6 Kerne zusammen)
Assoziativität	-	-	16-fach
Zeilengröße	-	-	64 Byte
Latenz	-	-	-

Bezeichnung im Text	Rechner 1	Rechner 2	Rechner 3
Software			
Betriebssystem	GNU/Linux	GNU/Linux	GNU/Linux
Distribution	OpenSUSE 11.2	OpenSUSE 11.2	OpenSUSE 11.2
Kernel	2.6.32-0.2-default	2.6.31.8-0.1-desktop	2.6.31.14-0.1-desktop
Compiler	gcc 4.4.1	gcc 4.4.1	gcc 4.4.1

Verwendetes Testproblem: BRUSS2D-MIX

Bei unserem Testproblem Bruss2D-Mix handelt es sich um den sogenannten Brüsselator.

Dies ist ein System aus vier hypothetischen chemischen Reaktionsgleichungen:

$$\begin{array}{lll} I & A & \overset{k_1}{\rightarrow} X \\ II & B+X & \overset{k_2}{\rightarrow} Y+D \\ III & 2X+Y & \overset{k_3}{\rightarrow} 3\mathrm{X} \quad (autokatalytisch) \\ IV & X & \overset{k_4}{\rightarrow} E \\ \hline I-IV & A+B \rightarrow D+E & \end{array} \tag{16}$$

Wir interessieren uns nun für die Konzentrationsänderungen der Stoffe während des chemischen Prozesses. Die Reaktionen laufen jeweils in den Geschwindigkeiten $k_1,\ldots,k_4$ ab.

Wir erhalten also für die zeitlichen Konzentrationsänderungen:

$$\begin{aligned} \dot{A} &= -k_1 \\ \dot{B} &= -k_2 B X \\ \dot{D} &= -k_2 B X \\ \dot{E} &= k_4 X \\ \dot{X} &= k_1 A + k_2 B X + k_3 X^2 Y - k_4 X \\ \dot{Y} &= k_2 B X - k_3 X^2 Y \end{aligned} \tag{17}$$

Die Konzentrationen von A und B werden immer konstant gehalten. Die Konzentrationen der Endprodukte D und E interessieren uns nicht. Bedeutsam sind dagegen die Differentialgleichungen zur Berechnung der Konzentrationen von X und Y. Zur Vereinfachung nehmen wir weiterhin an, dass die Reaktionsraten $k_{1,},\ldots,k_4=1$ sind.

Wir setzen $u=X$, $v=Y$ und erhalten dann die folgenden zwei Differentialgleichungen:

$$\begin{aligned} \frac{\partial u}{\partial t} &= A + u^2 v - (B+1)u \\ \frac{\partial v}{\partial t} &= Bu - u^2 v \end{aligned} \tag{18}$$

Berücksichtigen wir noch die Konzentrationsänderungen aufgrund von Diffusionsvorgängen in zwei Koordinatenrichtungen, werden die Differentialgleichungen partiell:

$$\frac{\partial u}{\partial t} = A + u^2 v - (B+1)u + \alpha\left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}\right)$$
$$\frac{\partial v}{\partial t} = Bu - u^2 v + \alpha\left(\frac{\partial^2 v}{\partial x_1^2} + \frac{\partial^2 v}{\partial x_2^2}\right) \quad (19)$$

Um diese Gleichungen näherungsweise mit dem Computer zu lösen, benutzen wir die Linienmethode, d.h. wir rechnen approximativ auf einem NxN Gitter diskrete Werte

$$x_{ij} = \frac{1}{N+1}\begin{pmatrix} i \\ j \end{pmatrix}, \quad 1 \le i \le N, 1 \le j \le N \quad (20)$$

aus.

Wir ersetzen dabei die zweiten örtlichen Ableitungen durch Differenzen.

$$\Delta x = \Delta x_1 = \Delta x_2 = \frac{1}{N+1} \quad (21)$$

Die diskrete Approximation von 15 sieht mit 16 und 17 also so aus:

$$\frac{du_{ij}}{dt} = A + u_{ij}^2 v_{ij} - (B+1)u_{ij} + \frac{\alpha}{(\Delta x)^2}(u_{i-1,j} + u_{i,j-1} - 4u_{ij} + u_{i+1,j} + u_{i,j+1})$$
$$\frac{dv_{ij}}{dt} = B u_{ij} + u_{ij}^2 v_{ij} + \frac{\alpha}{(\Delta x)^2}(v_{i-1,j} + v_{i,j-1} - 4v_{ij} + v_{i+1,j} + v_{i,j+1}) \quad (22)$$

Unser Testproblem Bruss2D-Mix hat folglich eine Systemgröße von.

$$2\mathrm{N}^2 \quad (23)$$

Berechnet man die u_{ij} in der Reihenfolge

$$u_{11}, v_{11}, u_{12}, v_{12}, \ldots, u_{ij}, v_{ij}, \ldots u_{NN}, v_{NN} \quad (24)$$

, so beträgt die Zugriffsdistanz:

$$d(f) = 2\mathrm{N} \quad (25)$$

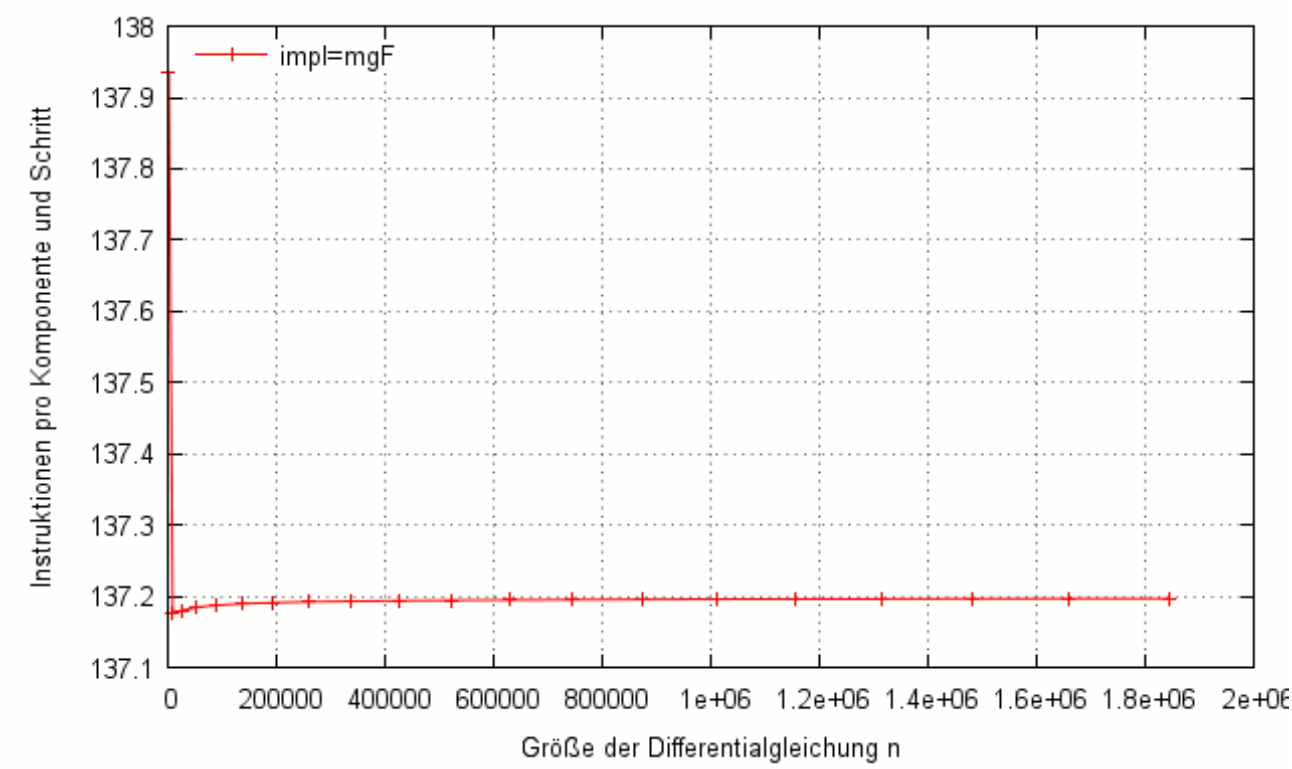
Des weiteren verwenden wir für die Parameter: $A = 1; B = 3{,}4$. Für den Diffusionskoeffizienten $\alpha = 2 * 10^{-3}$.
Die Anfangsbedingungen lauten:

$$u(x_{1,} x_2) = 0{,}5 + x_2$$
$$v(x_{1,} x_2) = 1 + 5\mathrm{x}_1 \quad (26)$$

Des weiteren gelten die Neumann-Randbedingungen:

$$\partial \frac{u}{\partial} n = 0$$
$$\partial \frac{v}{\partial} n = 0 \qquad (27)$$

Nach (18) müssen zur Auswertung der rechten Seite pro Komponente die gleiche Anzahl an Rechenoperationen gemacht werden. Wir überprüfen diese Annahme, indem wir für verschieden große Differentialgleichungssysteme die Anzahl der Instruktionen pro Komponente mit den Hardware Performance Countern messen.



Messreihe 14: Instruktionen pro Komponente und Zeitschritt für verschiedene Systemgrößen, gemessen auf Rechner 1 bei einem Integrationsintervall von H=0.01 und Schrittweite h = 0.0005

Aus Messreihe 15 geht hervor, dass unsere Annahme im Großen und Ganzen zutrifft. Bei kleinen Differentialgleichungen sehen wir einen Anstieg der Instruktionen pro Komponente. Das liegt daran, dass dort die einmal auszuführenden Instruktionen (z.B. zum Speicherallozieren) stärker ins Gewicht fallen als bei größeren Differentialgleichungen und wir außerdem ein kurzes Integrationsintervall gewählt haben, um den Rechenaufwand bei großen Differentialgleichung nicht ausufern zu lassen.

Literaturverzeichnis

1: Matthias Korch, Effiziente Implementierung eingebetteter Runge-Kutta-Verfahren durch Ausnutzung der Speicherzugriffslokalität, 2006
2: E. Hairer, S.P. Norsett, G.Wanner, Solving Ordinary Differential Equations I, 1993

Hiermit versichere ich, Konrad Ley, dass ich meine Bachelorarbeit selbständig verfasst, keine anderen Hilfsmittel als die von mir angegeben Quellen und Hilfsmittel benutzt und die Arbeit nicht bereits zur Erlangung eines akademischen Grades eingereicht habe.

Wonsees, 21.11.2010 _____

(Konrad Ley)