Karlsruhe Institute of Technology

Optimierung von Stencil-Algorithmen für invasive Architekturen

Bachelorarbeit von

Klaus Fischnaller

an der Fakultät für Informatik
Lehrstuhl Programmierparadigmen
IPD Snelting

Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Betreuender Mitarbeiter:	Dipl.-Inform. Matthias Braun

Bearbeitungsdauer: 01. Juni 2012– 28. September 2012

Kurzfassung

Diese Bachelorarbeit beschreibt eine Implementierung eines Frameworks für die Berechnung von Stencil Codes auf invasiven Architekturen in der Programmiersprache X10. Die Anpassung an die invasive Architektur erfordert es, die Arbeit möglichst parallelisierbar zu strukturieren und flexibel mit veränderbarer Ressourcenverfügbarkeit umgehen zu können.

Die Modellierung des Frameworks achtet darauf, dass beliebige Stencil Codes sehr einfach implementiert werden können. Dafür wird ein einfach zu verwendendes Interface angeboten.

Am Ende werden Messungen vorgenommen, soweit dies auf einer nicht invasiven Architektur überhaupt möglich ist. Dabei werden einzelne Größen, wie etwa die Feldgröße oder die Stencilgröße, variiert und ausgewertet, welche Einflüsse dies auf die Laufzeit hat.

Inhaltsverzeichnis

1. Einleitung

Dank der technischen Entwicklung und der anhaltenden Minituriarisierung, können immer kompaktere Schaltkreise in einen Chip gepackt werden. Schon 1965 hat Gordon Moore diesen Trend erkannt und Moore's Law formuliert. Dieses besagt, dass sich die Anzahl der Schaltkreise auf einem Computerchip in gleichbleibenden Intervallen[1] verdoppelt. Eine modernere Interpretation des Gesetzes geht sogar davon aus, dass sich nicht nur die Anzahl der Schaltkreise, sondern damit auch die Leistung der Chips stetig verdoppelt. Da die Schaltkreise jedoch nicht endlos verkleinert werden können, sondern irgendwann an die Grenze des physisch Machbaren stoßen werden, kann dieses exponentielle Leistungswachstum auf diese Art und Weise nicht ewig anhalten. Stattdessen versucht man den Leistungszugewinn in letzter Zeit immer stärker dadurch zu erzielen, mehrere Chips zusammenzuschließen und Aufgaben parallel ablaufen zu lassen. So bleiben die einzelnen Chips zwar etwa auf dem gleichen Niveau, gemeinsam steigert sich ihre Leistung jedoch enorm.

Supercomputer sind schon immer dadurch so schnell, indem sie durch sehr viele parallele Prozessoren ihre parallelisierten Algorithmen abarbeiten. Auch im Heimgebrauch werden Prozessoren mit immer mehr Rechenkernen angeboten. Vier bis sechs Kerne sind inzwischen nichts Besonderes mehr. Damit jedoch alle Kerne möglichst gut ausgelastet werden, muss auch die Programmierung an parallele Architekturen angepasst werden. Viele Aufgaben eignen sich sehr gut dafür, sie verteilt auf verschiedenen Prozessoren berechnen zu lassen. Ein gutes Beispiel dafür ist die Computergrafik, wo Bilder auf hochparallelisierten Grafikchips berechnet werden. Doch auch in anderen Wissenschaften gibt es viele Anwendungsgebiete, die für Parallelberechnung prädestiniert sind. Etwa die Berechnung von Hitzeausbreitung in Flüssigkeiten oder die Simulation von zellulären Automaten. Aufgaben wie diese lassen sich häufig mit so genannten Stencil-Algorithmen umsetzen. Wie solche Algorithmen sich auf parallelen Architekturen am besten umsetzen lassen, untersucht diese Bachelorarbeit. Dabei wird speziell die vom KIT mitentwickelte invasive Architektur betrachtet, die es ermöglicht, während der Laufzeit Rechenaufgaben bei dynamischer Ressourcenverfügbarkeit auf verschiedene Recheneinheiten zu verteilen.

[1] Ursprünglich ging er von einem Intervall von einem Jahr aus.

1.1 Ähnliche Arbeiten

Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures [DMV+08] untersucht Optimierungstechniken für Stencil Algorithmen auf einer Vielzahl aktueller (Jahr 2008) Multicore-Rechner. Dabei werden für diese Aufgabe sowohl gewöhnliche Rechnerarchitekturen untersucht, als auch die Eignung von Grafikkarten. Als Ergebnis kommt man zu der Erkenntnis, dass die parallele Berechnung zwar ein wichtiger, aber auch nur ein Teil der Optimierungen ausmacht. Als genau so wichtig werden etwa hierarchisch aufgebaute Speicher oder algorithmische Optimierungen aufgeführt.

Bei der Anzahl der Prozessoren ergab sich für gut parallelisierbare Algorithmen, dass für diese viele einfache Prozessoren besser geeignet sind, als wenige komplexe, die meistens eher auf serialisierte Berechnungen ausgelegt sind.

Grafikkarten scheinen sich für die Berechnung von Stencil Codes besonders gut zu eignen, da die Speicherbandbreite viel höher ist, als auf anderen Systemen. Jedoch ist häufig nur wenig Speicher verfügbar.

High Performance Stencil Code Algorithms for GPGPUs [SF11] untersucht die Implementierung von Stencil Codes auf modernen (Jahr 2011) Grafikkarten. Diese sind, wie invasive Systeme, sehr auf parallele Berechnungen ausgelegt. Besonders optimiert sind sie für Berechnungen mit Fließkommazahlen, da dies in der Computergrafik häufig vorkommt. Allerdings gibt es auch eine Reihe von Nachteilen, die mit dieser Technik kommen. Der Speicher hat eine hohe Latenz und die Recheneinheiten der Grafikkarten, die so genannten *Shader*, müssen auf einer besonderen Art programmiert werden.

Die Arbeit testet verschiedene Herangehensweisen, möglichst effizient Stencil Codes zu berechnen. Dabei kommt man zu dem Ergebnis, dass der vorgestellte *pipelined wavefront* Algorithmus in dem Benchmark der schnellste ist. Allerdings ergab die Untersuchung auch, dass der Speicher aktuell noch zu beschränkt ist, um größere Datenmengen zu bearbeiten.

2. Grundlagen

2.1 Invasive Architektur

Die invasive Architektur ist eine vom KIT mitentwickelte Rechnerarchitektur[THH+11]. Diese führt ein komplett neues Paradigma der parallelen Berechnung ein, das *invasive computing*. Rechner dieser Architektur verfügen über zahlreiche Recheneinheiten. Der Unterschied zu herkömmlichen Computerarchitekturen ist die Möglichkeit der Prozesse, diese Einheiten bis zu einen gewissen Grad selbst zu verwalten. So kann sich beispielsweise ein komplexer paralleler Algorithmus anfangs einen Pool von Recheneinheiten (Claim) exklusiv für seine Berechnungen reservieren und bei Bedarf während der Laufzeit zusätzliche Recheneinheiten anfordern (invade) oder ungenutzte Recheneinheiten wieder freigeben (retreat).

Recheneinheiten bestehen je nach Konfiguration aus einem Prozessor, einen Speicherbereich, der möglichst in der Nähe des Prozessors liegt, und Hilfsprozessoren wie etwa einer FPU. Um diese Architektur nutzen zu können, benötigen entsprechende Programmiersprachen mindestens die folgenden Befehle zur Steuerung der Ressourcen.

Invade wird benötigt, um benachbarte Recheneinheiten zu beanspruchen und sie dem aktuellen Claim hinzuzufügen. Ein Beispiel für eine solche Funktion:

P = invade(sender_id, direction, constraints)

Wobei *sender_id* den Prozess identifiziert, der das Invade angestoßen hat. *Direction* bestimmt, aus welcher Richtung neue Ressourcen angefordert werden sollen, etwa *West* oder *All*. In *constraints* sind Auflagen für die Auswahl kodiert, zum Beispiel das Vorhandensein einer FPU oder die Anzahl der gewünschten Recheneinheiten.

Infect überträgt den auszuführenden Programmcode auf alle im Claim beinhaltenden Recheneinheiten. Beispiel:

infect(claim, code)

Der in *code* beinhaltete Programmcode wird auf alle Recheneinheiten in *claim* übertragen und ausgeführt.

Retreat gibt die in einem Claim gebundenen Ressourcen wieder frei. Wenn nach der Ausführung des Codes, der mit *infect* übertragen wurde, die Recheneinheit nicht mehr benötigt wird, kann sie mit dieser Funktion wieder für andere Prozessoren

zur Verfügung gestellt werden. Wird sie doch noch benötigt, kann sie auch ohne erneutem *invade* mit Code "infiziert" werden. Beispiel für *retreat*:

retreat(claim)

Zu erwähnen ist, dass *claim* in diesem Fall nicht unbedingt der gesamte Claim sein muss, sondern auch eine Untermenge davon sein kann. Man muss also nicht alle beanspruchten Ressourcen freigeben, sondern kann auch nur einzelne Recheneinheiten zurückgeben.

2.1.1 Ressourcenverteilung nach Bedarf

Die Aufgabe der Ressourcenverteilung liegt üblicherweise im Aufgabenbereich des Betriebssystems. Anwendungen auf invasiven Systemen sind sich ihres Ressourcenverbrauchs bewusst und können jederzeit mehr Recheneinheiten vom Betriebssystem anfordern. Das führt unweigerlich zu der Frage, wie entschieden wird, welcher Prozess die gewünschten Ressourcen bekommen soll, sobald es zu einer Ressourcenknappheit kommen sollte. Ein Ansatz dafür ist es, eine Schnittstelle zwischen Programmen und Betriebssystem zur Verfügung zu stellen, über welche das Betriebssystem abfragen kann, welchen Speedup sich das Programm bei Erhalt der gewünschten Ressourcen verspricht. Je nach Speedup kann das Betriebssystem entscheiden, welcher Prozess die zusätzlichen Ressourcen am effektivsten verwenden würde. Hier bleibt dem Betriebssystem nichts anderes übrig, als den Prozessen zu vertrauen.

2.2 Stencil Codes

Stencil Codes sind eine Art Matrizenberechnung, bei der typischerweise durch benachbarte Zellenwerte ein neuer Wert berechnet wird[RRMc+97]. Das Wort **Stencil**[1] kommt daher, dass die benachbarten Werte wie durch eine Schablone, den so genannten Stencil, ausgewählt werden (siehe Abbildung 2.1). Anwendungsgebiete sind vor allem das Lösen von partiellen Differentialgleichungen, Bildbearbeitung und zelluläre Automaten.

Die Berechnungsregeln der Stencil Codes bestehen meistens aus einfachen Summen und Produkten. Gerade bei zellulären Automaten können die Berechnungen allerdings auch komplexer ausfallen.

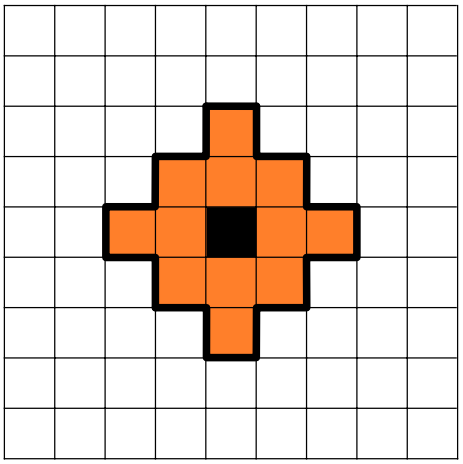

Abbildung 2.1: Der Zielwert wird durch die Werte in der Schablone berechnet.

Da die Schablone am Rand der Matrix über diese hinausragen könnte, müssen diese Fälle berücksichtigt werden. Häufig werden dort feste Werte zurückgegeben, oder die selbe

[1] Deutsch: Schablone

Matrix nochmal wiederholt oder gespiegelt hinter der Matrixgrenze angehängt. Welche Strategie dabei zum Einsatz kommt, hängt ganz von der Aufgabenstellung ab.

2.2.1 Beispiel: Conway's Game of Life

Conway's Game of Life ist ein zellulärer Automat[Gar], der 1970 vom britischen Mathematiker John Horton Conway entwickelt wurde. Der Automat besitzt ein zweidimensionales Feld mit Zellen, die jeweils zwei Zustände annehmen können: lebend und nicht lebend.

Die Zellenzustände des am Anfang befüllten Feldes werden mit jedem Schritt anhand der folgenden Regeln aktualisiert:

- Jede lebende Zelle mit weniger als 2 lebenden Nachbarzellen stirbt aus Einsamkeit.
- Jede lebende Zelle mit 2 oder 3 lebenden Nachbarzellen lebt weiter.
- Jede lebende Zelle mit mehr als 3 lebenden Nachbarzellen stirbt an Überbevölkerung.
- Jede nicht lebende Zelle mit genau 3 lebenden Nachbarzellen wird durch Reproduktion lebendig.

Diese vier einfachen Regeln erzeugen komplexe Muster, die dem Verhalten kleiner Mikroorganismen ähneln.

Der in diesem Automat verwendete Stencil ist ein Quadrat, das alle Nachbarzellen umschließt (Abb. 2.2).

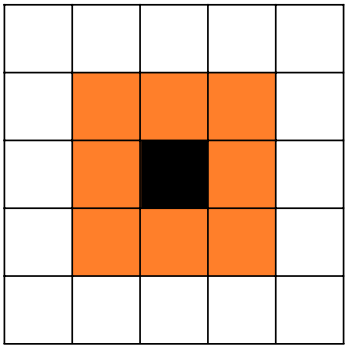

Abbildung 2.2: Der Stencil von Game of Life umschließt alle Nachbarzellen.

Am Rand des Feldes werden Nachbarzellen, die sich außerhalb befinden je nach Implementierung als nicht lebend betrachtet, oder das Feld wird wiederholt.

2.3 Programmiersprache X10

Konventionelle Programmiersprachen sind oft nicht oder nur umständlich in der Lage, Aufgaben parallel ablaufen zu lassen. Mit Threads unterstützen zwar fast alle gängigen Programmiersprachen Parallelität, diese lässt sich jedoch nur sehr beschränkt steuern, so dass es kaum möglich ist, bestimmte Aufgaben bestimmten Recheneinheiten zuzuweisen.

Die Programmiersprache X10, die ansonsten an Java angelehnt wurde, versucht diese Probleme anzugehen und führt dafür verschiedene neue Sprachkonzepte ein. Parallelität wird nicht mehr explizit in Threads verpackt, sondern direkt im Code gekennzeichnet. Den Rest übernimmt der Compiler.

X10 unterstützt dabei auch das Konzept von verschiedenen Recheneinheiten, die Places genannt werden. Somit lässt sich diese Eigenschaft sehr gut mit der Idee der Claims auf invasiven Architekturen verbinden.

2.3.1 Besonderheiten

In diesem Abschnitt wird auf einige semantische Besonderheiten der Sprache eingegangen. Da X10 sehr ähnlich der Sprache Java ist, wird nur die wichtigsten Änderungen betrachtet.

2.3.1.1 async und finish

Mit dem **async**-Keyword wird dem Compiler signalisiert, dass der folgende Block parallel zum Code nach dem Block ausgeführt werden soll. Um sicherzustellen, dass gestartete asynchrone Codestücke zu einem gewissen Zeitpunkt in der Ausführung bereits beendet wurden, wird das Keyword **finish** angeboten, welches am Ende seines umschlossenen Code-Blocks wartet, bis alle darin gestarteten asynchronen Programmstücke beendet wurden.

Das folgende Beispiel berechnet parallel die Werte der Variablen x und y, da x durch **async** parallel zum restlichen Programm ausgeführt wird. Erst nachdem y berechnet wurde, wird z asynchron berechnet[2]. Da die endgültige Summe, die in der Variable *summe* gespeichert werden soll, von den vorherigen parallel ausgeführten Berechnungen abhängig ist, muss ein **finish**-Block eingesetzt werden, um zu warten, bis die zwei asynchronen Berechnungen abgeschlossen wurden.

```
val a = 5, b = 2;
var x,y,z,summe : Int;
finish {
  async x = a+b;
  y = a-b;
  async {
    z = getRandomNumber();
    z = y+z;
  }
}
summe = x+y+z;
```

Wird **async** vor einen Schleifenkopf gesetzt, werden die einzelnen Schleifendurchgänge als Blöcke betrachtet, nicht etwa die ganze Schleife.

```
  val rand = new Random();
  val range : Range = (0..9);
  val x = new Array[Int](range, (p:Point(range.rank))=>rand.nextInt(10));
  val y = new Array[Int](range, (p:Point(range.rank))=>rand.nextInt(10));
  var z : Array[Int] = new Array[Int](range);
  finish async for (var i : Int = 0; i <= range.max(0); i++) {
    z(i) = x(i) + y(i);
  }
  Console.OUT.println(x + " + " + y + " = " + z);
```

Dieses Stück Code ist eine triviale Implementierung einer parallelen Vektoraddition. Es werden zwei *Int*-Arrays generiert und mit zufälligen Zahlen initialisiert. Anschließend werden die Zahlen mit dem selben Index parallel addiert und in ein drittes Array gespeichert.

2.3.1.2 at und GlobalRef

Anders als mit **async**, wo Code lokal parallel ausgeführt wird, ist es mit **at** möglich, Code auf einer anderen Recheneinheit auszuführen. Als Beispiel hierfür wären andere Computer[3] oder andere Recheneinheiten einer invasiven Architektur genannt.

Da der Code im **at**-Block auf einem anderen Ort (im Folgenden „Place“ genannt) ausgeführt wird, hat er keinen Zugriff auf Variablen außerhalb des Blocks. Umgekehrt hat

[2] Die Parallelität in diesem Fall ist sinnlos, soll jedoch nur die Funktionsweise verdeutlichen.
[3] Beispielsweise über MPI.

man außerhalb des Blocks auch keinen Zugriff auf Variablen, die im anderen Place erstellt wurden. Es ist jedoch möglich auf konstante Werte (mit *val* initialisiert) zuzugreifen. Der Compiler erkennt die im Code benötigten Werte und schickt sie mit zum anderen Place. So können Werte an den neuen Place übergeben werden. Jedoch ist das Zurückgeben von Ergebnissen so nicht möglich, da die in einem Block definierten Variablen nur innerhalb dieses Blockes sichtbar sind. Natürlich wäre es nicht sehr sinnvoll, Berechnungen auf andere Places auszulagern, wenn man dann nicht mehr auf das Ergebnis zugreifen könnte. Deshalb wurden die so genannten globalen Referenzen (**GlobalRef**) eingeführt. Diese beinhalten einen Verweis auf den Ursprungsplace und eine Referenz auf ein beliebiges Objekt im Speicher des ursprünglichen Places.

```
val PI = 3.1415;
val r = 4;

val umfang : Cell[Double] = new Cell[Double](0);
val ref = new GlobalRef[Cell[Double]](umfang);

finish at (here.next()) {
  val u = 2 * PI * r;
  at (ref.home) {
    ref().value = u;
  }
}

Console.OUT.println("Der Kreis hat einen Umfang von " + umfang.value);
```

Hiermit wird der Umfang eines Kreises auf einem anderen Place berechnet. Da ein Double zurückgegeben werden soll, GlobalRef jedoch nur Objekte referenzieren kann, muss die double-Variable in eine Container-Klasse gesteckt werden. Dafür bietet sich die Klasse Cell[4] an. Das GlobalRef-Objekt wird mit val als konstant erzeugt, so dass innerhalb des **at**-Blocks darauf zugegriffen werden kann.

Anschließend wird mit **at** der Place gewechselt. Als Parameter wird der gewünschte Place angegeben. **here** bezeichnet dabei immer den aktuellen Place. **here.next()** dementsprechend den nächsten.

Der Umfang wird berechnet und als konstanter Wert gespeichert. Mit Hilfe der globalen Referenz kann wieder zum Place, an dem das *umfang*-Objekt definiert wurde, zurückgewechselt werden und der Wert in der Variablen abgelegt werden.

Somit wurde das Ergebnis wieder zurückgeschickt und kann weiterverarbeitet werden.

2.3.2 Entwicklungsstand

X10 ist zum Zeitpunkt[5] noch in der Entwicklung. Die Sprache wurde bereits komplett spezifiziert, es könnten jedoch trotzdem noch Kleinigkeiten daran geändert werden. Entsprechend dem Entwicklungsstand ist auch die Qualität des SDK[6]. Die mitgelieferten Compiler sind leider sehr langsam und übersetzen den Code nicht direkt in Maschinensprache, sondern wahlweise in Java oder C++, bevor diese Übersetzung dann endgültig kompiliert wird. Dadurch verkompliziert sich auch die Fehlersuche, da Kompilierungsfehler häufig erst nach der Übersetzung auftreten und sich die Zeilennummern der Übersetzung nicht ohne weiteres in Zeilennummern des X10-Codes umwandeln lassen.

Bis auf die Sprachenspezifikation ist außerdem noch kaum Dokumentation vorhanden.

[4] Es gibt auch die Klasse GlobalCell, welche eine zusätzliche GlobalRef unnötig macht.
[5] August 2012
[6] Software Development Kit

3. Modellierung

3.1 Datenverteilung

Eine wichtige Aufgabe ist es, die Daten des Arrays auf die vorhandenen Recheneinheiten aufzuteilen. Trivialerweise könnte man das gesamte Array auf alle Recheneinheiten übertragen. Dies hätte allerdings großen Overhead zur Folge. Außerdem ist der Speicher der Recheneinheiten möglicherweise begrenzt, weshalb man sich auf das Nötigste beschränken sollte.

Das Nötigste sind alle Zellen, die vom Stencil umfasst werden, die also zur Berechnung auf der Recheneinheit benötigt werden. Eine Recheneinheit sollte jedoch nicht nur eine Zelle berechnen, sondern gleich mehre nacheinander, damit der Aufwand die zusätzlichen Daten zu übertragen nicht den der Berechnung an sich übersteigt. Wenn also mehr als eine Zelle pro Recheneinheit berechnet werden soll, werden dort alle Zellen benötigt, die von den Stencils aller zu berechnenden Zellen umfasst werden. Abbildung 3.1 verdeutlicht das Problem: Bei einzelnen Zellen ist die Auswahl der zu übertragenden Zellen sehr einfach. Auch bei mehreren Zellen scheint es auf dem ersten Blick kein Problem zu geben. Doch Probleme gibt es spätestens dann, wenn die zu berechnenden Zellen durch einen Zeilenwechsel getrennt werden.

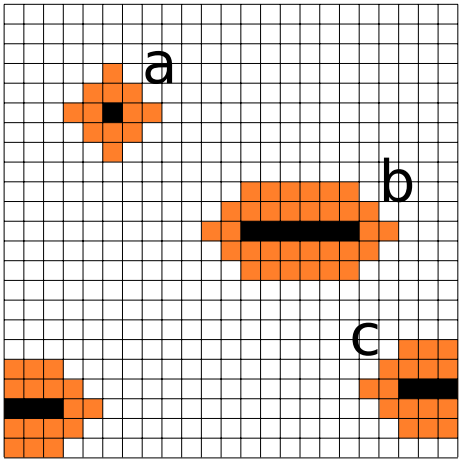

Abbildung 3.1: Das Herausschneiden von Arraystücken, die einen Zeilunumbruch beinhalten (c), ist kompliziert.

Die Abbildung zeigt drei Beispiele für Aufgabenteile, wie sie auf andere Recheneinheiten verteilt werden könnten. In Aufgabe (a) soll nur eine einzelne Zelle berechnet werden. Mit dem Herausschneiden eines Arraystücks gäbe es keine Probleme, da einfach ein Rechteck so ausgewählt werden kann, damit alle Zellen aus dem Stencil darin vorhanden sind. Auch wenn die Aufgabe wie in (b) gleich mehrere Zellen berechnen soll, kann sehr einfach ein umfassendes Rechteck ausgeschnitten werden. Dies macht erst dann Probleme, wenn zwischen den Zellen ein Zeilenumbruch stattfindet wie in Aufgabe (c).

Um dieses Problem zu umgehen, werden die Aufgaben nur zeilenweise aufgeteilt. Damit ist das kleinste Arraystück eine Zeile groß. Es kann jedoch auch mehrere Zeilen umfassen.

3.2 Allgemeine Struktur

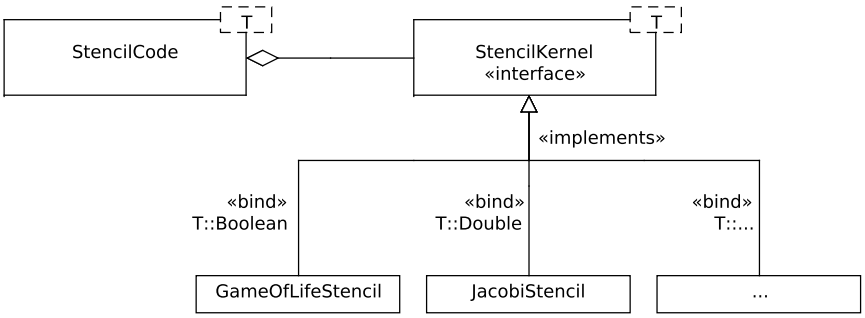

Abbildung 3.2: Vereinfachtes Klassendiagramm

In Abbildung 3.2 ist ein vereinfachtes Klassendiagramm zu sehen, in dem nur die eigentliche Struktur der Klassen zur Berechnung enthalten ist.

Die wichtigste Klasse ist **StencilCode**. Dort befindet sich die Wertematrix und die Aufteilung der Arbeit, so dass eine parallele Berechnung stattfinden kann.

Um die Berechnungsregeln austauschbar zu machen, wird das Strategie-Entwurfsmuster verwendet. Dafür wird das Interface **StencilKernel** bereitgestellt. Dieses Interface muss je nach gewünschtem Stencil Code neu implementiert werden. Es müssen vier Methoden geschrieben werden, welche in Abschnitt 3.4 genauer beschrieben werden.

Der Datentyp, den die Elemente in der Matrix haben sollen, wird über den generischen Typ **T** festgelegt. Damit kann also mit beliebigen Datentypen gearbeitet werden.

Nicht auf dem Klassendiagramm dargestellt ist die Hilfsklasse *StencilUtil*, welche statische Hilfsfunktionen, wie etwa zum Herausschneiden von Arraytücken, beinhaltet.

3.3 Klasse StencilCode

Die Klasse **StencilCode** (Abbildung 3.3) beinhaltet die eigentliche Funktionalität der Stencilberechnung. Da nicht in die selbe Matrix geschrieben werden soll, aus der auch gleichzeitig die Nachbarzellen gelesen werden, wird an zwei Arrays (*actual* und *next*) gearbeitet, die nach jedem kompletten Berechnungsdurchgang getauscht werden. Dabei wird in *next* nur geschrieben und aus *actual* nur gelesen.

Die Berechnungsregeln werden in der Implementierung von *kernel* festgelegt, die von *setKernel()* gesetzt werden kann. Auch das Feld muss anfangs erst von außen mit *setGrid()* gesetzt werden, da es evtl. mit bestimmten Werten initialisiert sein soll. Mit *getGrid()* lässt sich das aktuelle Feld jederzeit zurückgeben. So lässt sich beispielsweise nach einem Berechnungsdurchgang das Ergebnis auslesen.

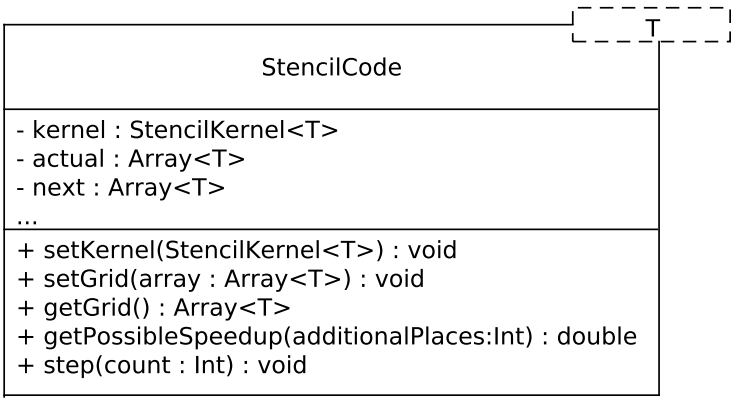

Abbildung 3.3: Die Klasse StencilCode

3.3.1 Arbeitsschätzung und Speedup-Funktion

Wie bereits in Abschnitt 2.1.1 beschrieben, wäre es möglich, die vom Betriebssystem bereitgestellten Ressourcen nicht nur davon abhängig zu machen, wie viele das Programm anfordert, sondern auch davon, welchen Geschwindigkeitszugewinn das Programm davon erwartet. Damit kann abgewägt werden, welche Programme ihre zusätzlichen Ressourcen nötiger haben. Diese Schnittstelle für das Betriebssystem ist jedoch noch nicht spezifiziert. Um dieses Verhalten jedoch trotzdem nachempfinden zu können, wird dieses Prinzip bei der Ressourcenanforderung in der Klasse **StencilCode** implementiert. Dort werden nur so lange neue Ressourcen angefordert, bis der erwartete Speedup unter einen Schwellwert fällt. Um diesen Speedup zu berechnen wird die Funktion *getPossibleSpeedup()* bereitgestellt, die als Parameter die Anzahl der zusätzlichen Recheneinheiten übergeben bekommt, zu denen sie den Speedup berechnen soll.

Für die Berechnung werden die Kosten zweier Aufgaben betrachtet:

1. Das Vorbereiten der Arbeit: Je mehr Recheneinheiten vorhanden sind, desto mehr Arbeit auf der Hauptrecheneinheit wird benötigt, um das Array in entsprechend kleine Stücke aufzuteilen und diese an die anderen Recheneinheiten zu senden.
2. Das eigentliche Berechnen: Diese Kosten beschreiben den Aufwand einer Recheneinheit ein Arraystück zu berechnen. Es sollte klar sein, dass diese Kosten insgesamt zwar konstant bleiben, aber es weniger Zeit benötigt, wenn mehr Recheneinheiten zur Verfügung stehen und daher mehr Arraystücke gleichzeitig berechnet werden können.

Die Funktion zur Berechnung des Speedups setzt sich wie folgt zusammen:

$$speedup(r, s, n, a) = \frac{r \cdot C_p + \frac{r \cdot s \cdot C_c}{n}}{r \cdot C_p + \frac{r \cdot s \cdot C_c}{n+a}}$$

r Anzahl der noch zu berechnenden Arraystücke.

s Anzahl der Zellen, die berechnen werden.

n Aktuelle Anzahl der Recheneinheiten.

a Anzahl der zusätzlichen Recheneinheiten, für die der Speedup berechnet werden soll.

C_p Kosten der Vorbereitung

C_c Kosten der Berechnung

Die Werte von C_p und C_c sind durch Ausprobieren ermittelt worden. Da sie jedoch je nach Stencil und Berechnungsregeln variieren, wäre eine mögliche Verbesserung der StencilCode-Klasse die Berechnung dieser Konstanten durch die Messung der benötigten Zeiten.

3.3.2 Invasive Aufgabenverteilung

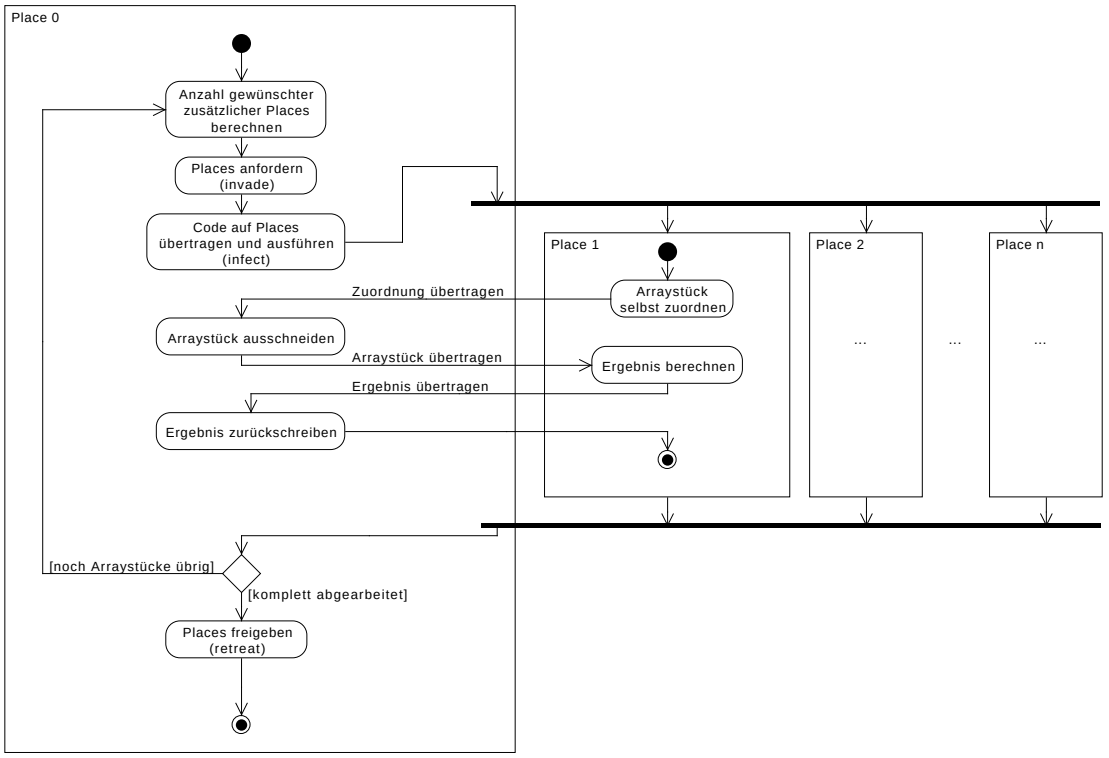

Abbildung 3.4: Aktivitätsdiagramm eines Berechnungsvorgangs

Auf Abbildung 3.4 ist der Ablauf eines kompletten Berechnungsschrittes der Funktion *step()* abgebildet.

Da durch die bereitgestellte *infect()*-Funktion der identische Code auf alle Places im Claim übertragen wird, ist es nicht ohne weiteres möglich, jedem Place ein anderes Arraystück zur Berechnung mitzuschicken. Aus diesem Grund ordnet sich jeder Place ein Arraystück selbst zu. Dafür wird ein Objekt der Klasse **AtomicInteger** eingesetzt. Dieses ist als val-Objekt in der **StencilCode**-Klasse von allen Places aus erreichbar.

Durch atomares Auslesen und Inkrementieren dieser Integer-Zahl kann jedem Place ein unterschiedlicher Teil des Gesamtarrays in Form eines Index zugewiesen werden. Um dieses Arraystück in den Place zu holen, wird die von X10 bereitgestellte Funktionalität benutzt, Codeteile auf anderen Orten auszuführen. Damit kann problemlos im Hauptplace auf den lokalen Speicher und damit auf das Gesamtarray zugegriffen werden, um den gewünschten Teil herauszukopieren und nur diesen anschließend zurück zum Berechnungsplace zu übertragen.

Ähnlich verhält es sich mit der Übertragung des Ergebnisses, nachdem das Arraystück verarbeitet wurde. Das Ergebnis wird auf den Hauptplace übertragen und dort in das Ergebnisarray geschrieben.

Dieser Vorgang wiederholt sich so oft, bis alle Arraystücke bearbeitet wurden. Dabei wird nach jedem Schritt versucht, zusätzliche Claims (Places in X10) zu erringen, falls dies einen

Speedup verspricht. Freigegeben werden die Claims jedoch erst nachdem alles berechnet wurde, da sie erst am Ende nicht mehr benötigt werden.

3.4 Interface StencilKernel

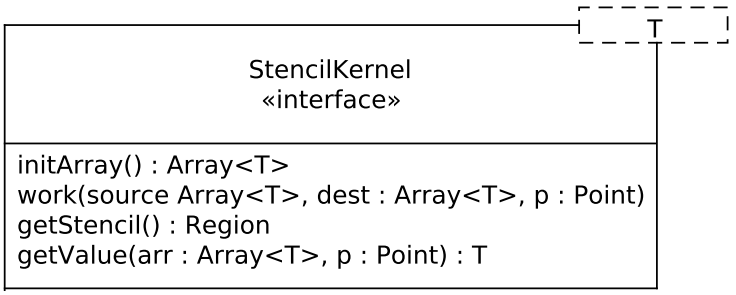

Abbildung 3.5: Das Interface StencilKernel

Über dieses Interface lassen sich beliebige Stencil Codes implementieren. Dafür werden lediglich vier Funktionen benötigt. Die Festlegung von Arraygröße und anderen Eigenschaften sollte über den Konstruktor erfolgen und in der Klasse abgespeichert werden. Diese Werte können zum Beispiel von der Funktion *initArray()* genutzt werden, um ein Anfangsarray in der richtigen Größe für die Berechnung zu erstellen. Dabei kann das Array gleich mit Anfangswerten gefüllt werden. Um die Größe der herausgeschnittenen Arraystücke berechnen zu können, wird die Funktion *getStencil()* implementiert. Diese gibt ein Rechteck zurück, in das alle von der Schablone umfassten Zellen enthalten sind. Dies gilt auch, wenn die Schablone nicht rechteckig ist. Dabei wird angenommen, dass sich die momentan berechnete Zelle im Zentrum (Nullpunkt) befindet. Die Rückgabe für einen Stencil, der alle acht Nachbarzellen beinhaltet wäre entsprechend ein Rechteck von $(-1, -1)$ nach $(1, 1)$.

Je nach Aufgabenstellung kann es nötig sein, Randfälle anders zu behandeln. Manchmal ist es sinnvoll bei Zugriffen auf Zellen außerhalb des Arrays einen konstanten Wert zurückzugeben, manchmal das Array zu wiederholen oder es zu spiegeln.

Um diese Entscheidung dem Entwickler eines Stencil-Kernels zu überlassen, wird die Funktion *getValue()* angeboten. Diese bekommt ein entsprechendes Array und eine Koordinate übergeben. Die Funktion soll den den Wert des Arrays an der entsprechenden Koordinate zurückgeben. An dieser Stelle ist es möglich, die Koordinate auf Randfälle zu betrachten und beim Eintreten eines solchen Falles ein gewünschtes Ergebnis zurückzugeben. Häufig wird hier gewünscht, dass sich das Array hinter dem Rand wiederholt. Aus diesem Grund wird in der Hilfsklasse *StencilUtil* eine Funktion angeboten, der eine Koordinate und Arraygröße übergeben werden und eine neue Koordinate zurückgibt, die auf die Zelle des wiederholten Arrays zeigt. Die beschriebene Funktion heißt *StencilUtil.repeat()*.

Die eigentliche Berechnungsvorschriften werden in *work()* implementiert. Diese Funktion bekommt ein Array mit den aktuellen Werten und eines für die Ergebnisse übertragen. Außerdem wird die entsprechende Koordinate übergeben, an der das Ergebnis berechnet werden soll.

3.4.1 Beispiel: GameOfLifeStencil

Als Beispiel für eine Implementierung der StencilKernel-Schnittstelle wird hier Game of Life vorgestellt. Das Feld besteht aus *boolean*-Werten, weshalb der generische Typ T zu *boolean* gebunden wird. Dem Konstruktor der Klasse werden die gewünschte Höhe und Breite des Arrays übergeben. Um das Feld automatisch befüllen zu lassen, werden bei der Konstruktion des Arrays in *initArray()* zufällig lebende und tote Zellen verteilt.

```
public def initArray() : Array[boolean] {
    val rand = new Random();
    val r = (0..(width-1))*(0..(height-1));
    val x = new Array[boolean](r, (p:Point(r.rank))=>rand.nextBoolean());
    return x;
}
```

Der Stencil in Game of Life ist ein Quadrat über die acht Nachbarzellen. Die Implementierung von *getStencil()* lautet also:

```
  public def getStencil() : Region {
    return (-1..1)*(-1..1);
  }
```

In diesem Beispiel werden die Ränder so betrachtet, als würde sich das selbe Array hinter ihnen wiederholen. Dafür wird eine Hilfsfunktion der *StencilUtil*-Klasse benutzt.

```
  public def getValue(val arr:Array[boolean], val p : Point) : boolean {
    return arr(StencilUtil.repeat(p,[width,height]));
  }
```

Die eigentliche Berechnung der Zellen findet in der folgenden Funktion statt:

```
  public def work(val source:Array[boolean], var destination:Array[boolean], val ↩
      p:Point) {
    val x : Array[Int] = [-1, 0, 1, 1, 1, 0,-1,-1];
    val y : Array[Int] = [-1,-1,-1, 0, 1, 1, 1, 0];
    var count : Int = 0;
    // Lebende Nachbarn zaehlen
    for (var i : Int = 0; i < 8; i++) {
      val np = StencilUtil.repeat(p+[x(i),y(i)],[source.region.max(0),source.↩
          region.max(1)]);
      if (source(np)) count++;
    }
    // Game of Life - Regeln
    if (source(p)) {
      if (count < 2) destination(p) = false;
      else if (count == 2 || count == 3) destination(p) = true;
      else if (count > 3) destination(p) = false;
    } else {
      if (count == 3) destination(p) = true;
      else destination(p) = false;
    }
  }
```

Hier werden zuerst die lebenden Nachbarzellen gezählt und anschließend mit Hilfe der *Game of Life*-Regeln der neue Zustand der Zelle berechnet.

4. Messungen

Die invasive Architektur befindet sich, wie die Programmiersprache X10, noch in der Entwicklung. Aus diesem Grund können Messungen nur auf einem Simulator durchgeführt werden. Auch dieser hat erst ein frühes Entwicklungsstadium erreicht und der mit dem X10 SDK mitgelieferte Compiler hält sich mit Optimierungen zurück. Die Messergebnisse sind deshalb mit Vorsicht zu genießen. Außerdem lassen sich die Ergebnisse nicht auf ein richtiges invasives System übertragen, da der Testrechner nicht in der Lage ist, so viele Berechnungen wirklich parallel auszuführen, wie es eigentlich der Fall sein sollte.

4.1 Aufbau und Messung

Als Rechner kommt folgender Aufbau zum Einsatz:

Prozessor AMD Phenom(tm) II X6 1090T Processor (6 cores)

Arbeitsspeicher 8 GB

Betriebssystem Linux (Kernel 3.5.3)

Es werden die folgenden Softwareversionen verwendet.

X10 SDK 2.2.2 (es wird der X10 zu C++ Compiler verwendet)

InvadeX10 Simulator Entwicklungsversion vom 19.06.2012

Der Simulator ist so konfiguriert, damit er 28 Places zur Berechnung bereitstellt.

Für die Messung wird das GNU/Linux time-Kommando verwendet, welches dafür benutzt werden kann, die Laufzeit einer Anwendung zu messen. Dadurch wird auch die Initialisierung des Simulators gemessen. Da dies jedoch bei allen Messungen der Fall ist, bleiben die Ergebnisse vergleichbar.

4.2 Skalierbarkeit

Bei diesen Messungen wird darauf geachtet, wie Veränderungen verschiedener Größen die Laufzeit beeinflussen. Sofern nicht anders angegeben wird das Programm mit einer Arraygröße von 50×50 initialisiert. Die verwendete *StencilKernel*-Implementation macht nichts außer die Ursprungszelle zu inkrementieren und zurückzuschreiben. Um trotzdem Nachbarzellen zu übertragen, wird ein quadratischer Stencil verwendet, der alle acht Nachbarzellen umfasst. Diese Nachbarzellen werden jedoch bei der Berechnung nicht verwendet.

Das Array wird bei der Erzeugung mit Nullwerten befüllt und es werden zehn Berechnungsdurchläufe durchgeführt.

4.2.1 Variation der Feldgröße

Bei dieser Messung wird die Größe des Arrays verändert. Das Feld bleibt quadratisch, die Breite und damit die Höhe werden variiert.

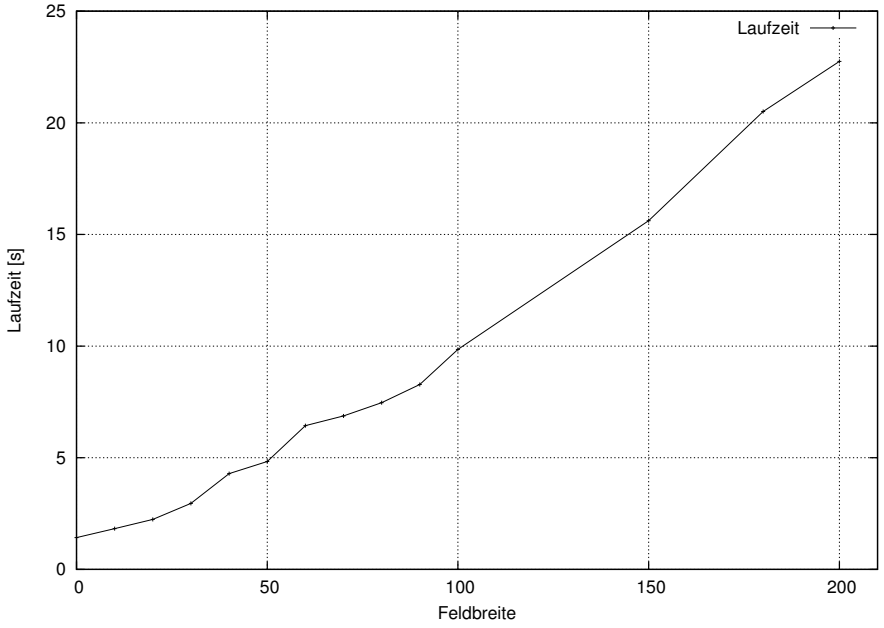

Abbildung 4.1: Laufzeit je nach Feldgröße

Trotz quadratischer steigender Anzahl der zu berechnenden Zellen, steigt die Laufzeit lediglich linear. Grund hierfür ist, dass das Array zeilenweise (je 2 Zeilen pro Place) aufgeteilt und parallel berechnet wird. Da die Zeilen im Gegensatz zu den vorhandenen Zellen linear steigen, steigt auch die Laufzeit annähernd linear. Die Steigungsveränderung kommt hauptsächlich durch die Aufteilung und Übertragung der Arraystücke zustande.

Da der Testrechner nur sechs Kerne besitzt, können zudem immer nur sechs Berechnungen gleichzeitig durchgeführt werden, selbst wenn es im Simulator bis zu 28 Places gibt. Da die Zeit außerhalb des Simulators gemessen wird, macht sich diese sequentielle Berechnung auch in der Messung kenntlich. Bei sehr wenigen Zeilen ist die Steigung daher kleiner, da dort auch alles außerhalb der Simulation parallel ablaufen kann.

4.2.2 Variation der Stencilgröße

Um die Auswirkungen der Stencilgröße zu untersuchen, wird bei diesen Messungen die Größe des Stencils verändert. Die Form bleibt dabei ein Quadrat. Auch hier steigt die

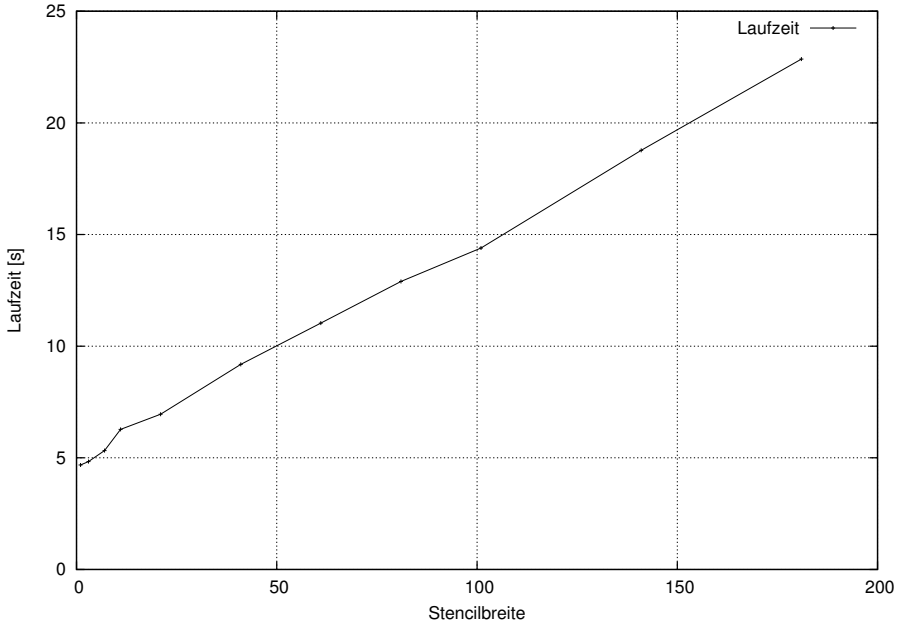

Abbildung 4.2: Laufzeit je nach Stencilgröße

Laufzeit linear. Die Anzahl der Arraystücke bleibt konstant. Nur die Größe des übertragenen Arraystücks steigt linear mit der Stencilgröße, da das eigentliche Array von einem Rand der halben Stencilbreite umkreist wird.

4.2.3 Variation der verfügbaren Recheneinheiten

Bei dieser Messung werden die verfügbaren Places im Simulator verändert. Hierfür wird ein anderer Testrechner eingesetzt, da dort die Parallelität besser betrachtet werden kann. Zum Einsatz kommt hier ein Rechner mit 16 echten Kernen und 32 GB RAM.

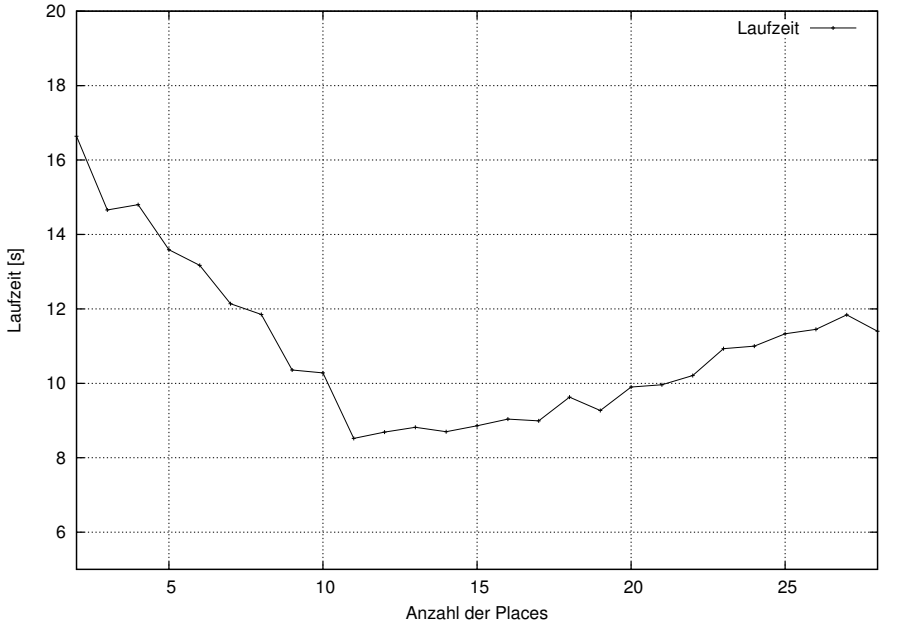

Abbildung 4.3: Laufzeit je nach Anzahl an Places

Auf Abbildung 4.3 ist gut zu erkennen, dass die Laufzeit mit steigender Anzahl an Kernen sinkt. Dies ließ sich erwarten, da die Arbeit auf die Places aufgeteilt wird. Die Verteilung

der Arbeit scheint dabei weniger Zeit zu verbrauchen, als die gewonnene Zeit durch die Parallelisierung. Erst bei etwa elf Places steigt die Laufzeit wieder. Dies könnte damit zusammenhängen, dass der Rechner nur 16 Kerne besitzt und die restlichen Kerne etwa vom Simulator oder für andere Aufgaben benötigt werden. Da die Berechnungen bei mehr Places als verfügbaren Kernen wieder teilweise sequentiell ablaufen, steigt die Laufzeit wieder.

4.3 Adaptivität

Auf invasiven Architekturen kann es vorkommen, dass verschiedene Prozesse um die spärlich gesäten Ressourcen ringen. In solchen Fällen müssen sich die Anwendungen den Umständen anpassen und trotzdem fehlerfrei funktionieren. Bei dieser Messung wird eine solche Situation simuliert und das Verhalten des Stencilprogramms beobachtet.

Im Simulator werden zwei Anwendungen gestartet. Zum Einen das Stencil-Testprogramm, welches fünf mal hintereinander einen Stencil Code auf einem gesamten Array berechnet und zum Anderen das RandomLoad-Programm, welches zufällig Ressourcen anfordert und wieder freigibt.

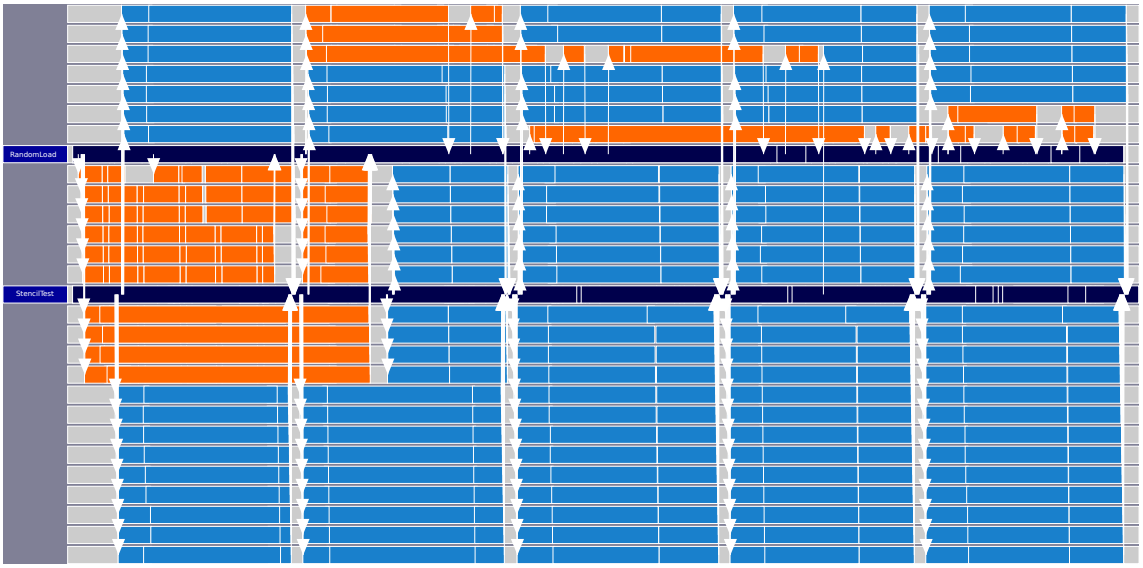

Abbildung 4.4: Ressourcenkampf zwischen Stencil-Testprogramm (blau) und RandomLoad (orange)

Auf Abbildung 4.4 sind die Claims (y-Achse) abhängig von der Zeit (x-Achse) dargestellt. Vom Stencil-Testprogramm belegte Claims sind blau, die von RandomLoad orange.

Wie zu sehen ist, beginnt RandomLoad schon vor dem Stencil-Test mit der Belegung von Claims. Grund dafür ist wohl die Vorbereitung der Berechnung des Stencil-Programms. Dazu gehören etwa die Erstellung und Befüllung des Arrays. Sobald das Stencil-Testprogramm mit der Invasion von Claims beginnt, beansprucht es sofort alle freien Ressourcen und auch nachdem RandomLoad Claims wieder freigibt, werden diese vom Stencil-Programm aquiriert. Nur zwischen zwei Berechnungsschritten[1] gibt das Stencil-Testprogramm seine Claims frei und RandomLoad damit die Gelegenheit Ressourcen zurückzuholen. Dieser Zeitslot ist jedoch so kurz, dass es RandomLoad nur selten schafft, einen neuen Claim zu erhalten.

Das Stencil-Testprogramm hat keine Probleme damit, die Berechnung durchzuführen, von RandomLoad lässt es sich dabei nicht stören.

[1] Zwischen zwei kompletten Matrixberechnungen.

Diese Messung zeigt, wie wichtig es ist, dass das Betriebssystem in die Ressourcenverteilung eingreifen kann, damit kein Prozess durch aggressive andere Prozesse verhungern muss.

5. Ausblick und zukünftige Arbeiten

Das invasive Computing ist zweifellos eine vielversprechende Technologie, die dem Trend der zunehmenden Parallelisierung von Rechenleistung folgt und mit der invasiven Architektur einen komplett neuen Rechneraufbau, angepasst an diese Umstände, vorschlägt.

Ob sich die invasive Architektur durchsetzen wird, lässt sich bisher leider nicht voraussagen. Dass sie jedoch mindestens in einigen Aufgabenbereichen eingesetzt wird, ist durchaus wahrscheinlich. Einer dieser Bereiche sind Stencil Codes, die aufgrund ihrer guten Parallelisierbarkeit als perfektes Beispiel gelten.

Diese Bachelorarbeit hat sich mit der Umsetzung dieser Aufgabe auseinandergesetzt und eine mögliche Implementierung von Stencil Codes auf einer invasiven Architektur untersucht. Damit hat sie eine Tür für weitere Verbesserungen und Optimierungen dieses Verfahrens geöffnet, die Thema für andere wissenschaftliche Arbeiten sein könnten.

Beispiele für solche Optimierungen sind:

Die Aufgabenverteilung auf die Claims könnte überarbeitet werden. Statt das Array zeilenweise zu verteilen, könnte es abhängig von der Größe des Feldes aufgeteilt werden. Wie groß müssen diese Teile sein, um die Laufzeit zu minimieren? Für die Umsetzung dieser Aufgabe muss auch die Übertragung der Arraystücke und ihrer Nachbarn überarbeitet werden. Ist die Übertragung über ein einfaches Array in diesem Fall noch ausreichend, oder wird eine neue Datenstruktur benötigt?

Die Kostenfunktion, die den Speedup berechnet, könnte verbessert werden. Aktuell verwendet sie für die Berechnung zwei durch Ausprobieren gewählte Konstanten: die Kosten für die Vorbereitung und die Kosten für die Berechnung eines Arraystücks. Könnten diese Werte durch Messungen zur Laufzeit ermittelt werden? Oder eignet sich vielleicht eine andere Art der Speedup-Berechnung viel besser als die hier vorgeschlagene?

Literaturverzeichnis

[DMV+08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf und Katherine Yelick: *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures*. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Seiten 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press, ISBN 978-1-4244-2835-9. `http://dl.acm.org/citation.cfm?id=1413370.1413375`.

[Gar] Martin Gardner: *MATHEMATICAL GAMES, The fantastic combinations of John Conway's new solitaire game 'life'.* `http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm`.

[RRMc+97] Gerald Roth, Gerald Roth, John Mellor-crummey, John Mellor-crummey, Ken Kennedy, Ken Kennedy, R. Gregg Brickner und R. Gregg Brickner: *Compiling Stencils in High Performance Fortran*. In: *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM*, Seiten 1–20. ACM Press, 1997.

[SF11] Andreas Schäfer und Dietmar Fey: *High Performance Stencil Code Algorithms for GPGPUs*. Procedia Computer Science, 4(0):2027 – 2036, 2011, ISSN 1877-0509. `http://www.sciencedirect.com/science/article/pii/S1877050911002791`, Proceedings of the International Conference on Computational Science, ICCS 2011.

[THH+11] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat und Gregor Snelting: *Invasive Computing: An Overview*. In: M. Hübner und J. Becker (Herausgeber): *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, Seiten 241–268. Springer, Berlin, Heidelberg, 2011. `http://pp.info.uni-karlsruhe.de/uploads/publikationen/teich11msoc.pdf`.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Karlsruhe, den 28. September 2012

(Klaus Fischnaller)