

UNIVERSITÄT PASSAU  
Fakultät für Informatik und Mathematik

Seminararbeit

**PATUS - Parallel Auto-Tuned Stencils**  
Ein Framework für Stencil-Berechnungen

im Hauptseminar „Software Engineering für Exascale Computing“

von Andreas Baier  
Passau, Mai 2013

## **Zusammenfassung**

PATUS ist ein Code-Generierungs- und Auto-Tuning Tool für die Klasse der Stencil-Berechnungen. Es produziert aus der Beschreibung eines Stencil-Codes mittels Strategien für die Traversierung von Grids und hardwarespezifischer Backends plattformspezifisch optimierten C-Code, der in Anwendungen eingebettet werden kann. Dabei steht eine parallele Ausführung mit möglichst geringer Kommunikation im Vordergrund. Beim Entwurf von PATUS wurde großen Wert auf eine hohe Wiederverwendbarkeit der beschriebenen Stencil-Muster und eine hohe Produktivität bei der Programmierung durch eine ausdrucksstarke Syntax gelegt.

# Inhaltsverzeichnis

1	Vorwort	2
2	Grundlagen	4
2.1	Stencil-Codes . . . . .	4
2.2	Feedback-basierte Optimierung . . . . .	6
2.3	Kerngedanken von PATUS . . . . .	7
2.4	Schematischer Aufbau . . . . .	7
2.5	Projekthintergrund . . . . .	9
3	Implementierungsdetails	9
3.1	Die Stencil-Spezifikation . . . . .	9
3.2	Die Strategy-Definition . . . . .	11
3.3	Statische Optimierungen . . . . .	13
3.4	Auto-Tuning . . . . .	15
3.5	Einschränkungen . . . . .	17
4	Anwendungsbeispiele und Benchmarks	18
4.1	Large-Scale Erdbeben Simulation: AWP-ODC . . . . .	18
4.2	Testergebnisse . . . . .	20
5	Fazit und Aussicht	22
	Abbildungsverzeichnis	24
	Quelltextverzeichnis	24
	Literaturverzeichnis	25

# 1 Vorwort

Die Landschaft des High Performance Computings steht vor einem Umbruch. Die bis 2020 erwartete Klasse von Supercomputern im Exascale Bereich ( $10^{18}$  FLOPS) erfordert ein Umdenken, was Hardware-Design, Algorithmik aber auch das Software-Engineering betrifft, da sich architekturbedingt die Topologien für Prozessoren, Speicher und Verbindungsnetzwerke grundlegend ändern müssen. Bis-her wurde im HPC-Bereich Software in der Regel spezifisch auf eine Plattform hin entwickelt und optimiert mit stark plattformabhängigen Code als Ergebnis.[Nai11]

Plattformspezifische Optimierungen machen den Quellcode unübersichtlich und es ist schwer, die Software zu warten. Das Programmieren auf Low-Level-Ebene mit MPI und OpenMP erhöht die Fehleranfälligkeit, da man sich nicht nur auf die eigentliche Berechnung konzentrieren kann, sondern auch das korrekte Verteilen der Daten und die Kommunikation der Prozesse untereinander organisieren muss. In Anbetracht dessen, dass Supercomputer nur wenige Jahre *aktuell* sind, wird so die Wiederverwendbarkeit erschwert und die Produktivität maßgeblich gemindert.

Zwar existieren mittlerweile mehrere Standards und Frameworks wie OpenCL, die plattformübergreifende Parallelprogrammierung erlauben, die resultierenden Anwendungen besitzen bei generischer Programmierung aber eine geringe Performanzportabilität.

Eine Studie, durchgeführt an der Universität Basel, unterstreicht die oben genannten Umstände anschaulich [BCR<sup>+</sup>11]:

In der Veranstaltung *High-Performance-Computing* mussten Master-Studenten, die zuvor noch keine Erfahrungen im High-Performance-Computing besaßen, die klassische Wellengleichung mittels unterschiedlicher Parallelitätsmodelle umsetzen. Anschließend wurden die Studenten gebeten, Lernkurve und Zeitaufwand zu bewerten. Die Ergebnisse der Studie sind in Abbildung 1 und 2 zu sehen.

Prog. Model	Working Hours	Parallel Overhead	Performance	Lines of Code	Learning Curve
Java	3.75	36%	4.79	231	2.0
Chapel	2.67	0%	1.49	110	2.0
OpenMP	3.25	7%	5.99	196	1.7
MPI	17.50	186%	6.19	597	2.5
CUDA	6.00	—	23.10	196	2.0
PATUS	0.85	—	7.97	22	1.3

Abbildung 1: Ergebnisse des Fragebogens der Studie Run, stencil, run (Grafik aus [BCR<sup>+</sup>11]).

Hier ist die hoch bemessene Einarbeitungszeit von MPI und OpenMP hervorzuheben. In Abbildung 2 lässt sich erkennen, dass die Performanz von OpenMP zwar

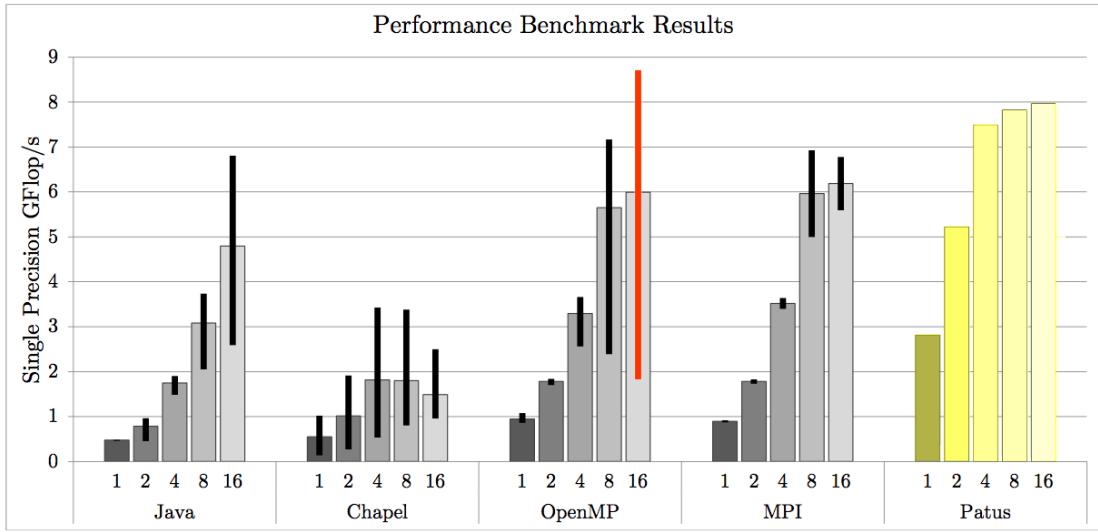


Abbildung 2: Benchmarkergebnisse der implementierten Lösungen der Studenten. Gelb hervorgehoben: Benchmarkergebnisse von PATUS. Schmale schwarze/rote Balken zeigt die Streuung der Performanz der Lösungen (Grafik aus [BCR<sup>+</sup>11], bearbeitet).

sehr hoch ausfallen kann, allerdings die Streuung der Ergebnisse ebenso hoch ausfällt. Eine Umsetzung mittels des Frameworks PATUS zeigte hingegen eine geringe Einarbeitungszeit bei hoher Skalierbarkeit und Performanz auf. Eine detaillierte Erläuterung der Ergebnisse findet sich in [BCR<sup>+</sup>11].

Im Projekt PATUS wurde versucht oben genannte Nachteile zu adressieren, indem man einerseits eine Abstraktionsebene durch domänenspezifische Sprachen für die Programmierung eingeführt hat und sich den plattformspezifischen Quellcode generieren lässt. Hierbei werden Eigenheiten entsprechender Plattformen berücksichtigt, um eine höchstmögliche Performanz zu erzielen. Dabei wurde eine hohe Wiederverwendbarkeit und Performanzportabilität des Quellcodes anvisiert mit dem Ziel die Produktivität maßgeblich zu steigern. Dabei konzentriert sich PATUS auf die Klasse der Stencil-Codes, einer Klasse von Berechnungen, die auf Punkte-Gittern mit festen Berechnungsmustern statt findet.

In dieser Arbeit wird auf Idee und Konzept sowie auf Aufbau und Workflow der Software PATUS eingegangen.

## 2 Grundlagen

PATUS ist ein Akronym für **P**arallel **A**uto-**T**uned **S**tencils. Es ist ein Tool für Quellcode-Generierung und Auto-Tuning von Stencil-Codes. Hierfür bietet es zwei domänenspezifische Sprachen zur Beschreibung der Stencilberechnung an. In die Codegenerierung gehen Beschreibungen von Hardwarecharakteristika ein und in erster Instanz wird Quellcode generiert, der mittels feedback-orientierter Optimierung, dem Auto-Tuning, optimiert wird. Am Ende wird Quellcode erzeugt, der in Anwendungen eingebettet werden kann und zahlreiche Optimierungen für die Zielplattform enthält.

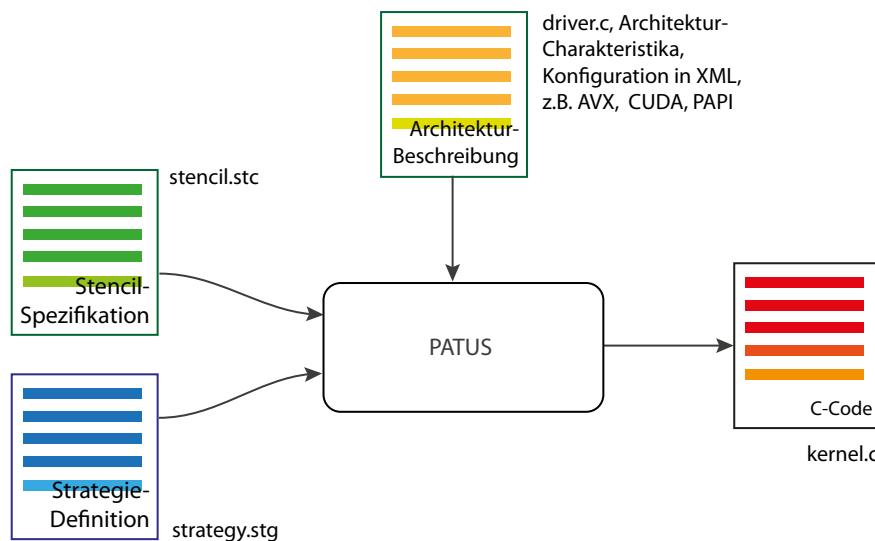


Abbildung 3: Quellcodeerstellung mit PATUS.

### 2.1 Stencil-Codes

PATUS ist ein Softwareframework für Stencil-Codes. Bei Stencil-Codes lässt sich die Problemdomäne durch ein (mehrdimensionales) Punktegitter darstellen und die Berechnung durch ein festes Zugriffsmuster beschreiben, mittels dessen man über das Gitter iteriert. In jedem Iterationszyklus  $t$  wird dabei aus jedem Punkt sowie benachbarten Punkten des Gitters ein neuer Wert des Gitters zum Zeitpunkt  $t+1$  berechnet (siehe Abb. 4).

Man unterscheidet zwei Klassen von Gittern: Gitter mit Datenabhängigkeiten und solche ohne Datenabhängigkeiten zwischen den Gitterpunkten. Erstere werden in der Regel über den *Gauss-Seidel/Rot-Schwarz-Verfahren* traversiert, für letztere ist die *Jacobi-Iteration* eine Strategie für die Traversierung des Gitters. PATUS

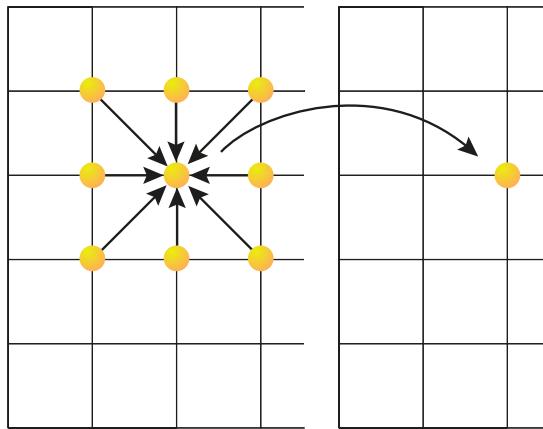


Abbildung 4: Muster für einen 9-Punkt Stencilcode.

unterstützt nur die Jacobi-Iteration als Traversierungsstrategie. Es werden innerhalb einer Gitter-Traversierung immer alle Punkte abgelaufen und die Reihenfolge der Berechnung darf keine Rolle spielen.[Chr11, S.83]

Werden für die punktweise Berechnung nur Punkte einer Iteration ohne Datenabhängigkeiten zugelassen, so ist der Stencilcode inhärent parallel. Dies macht Stencilcodes attraktiv für Prozessoren mit massiver Parallelität. Dabei liegen Stencilcodes im Allgemeinen in der Klasse  $\mathcal{O}(1)$  und besitzen somit eine geringe arithmetische Intensität (FLOPs/übertragene Datum<sup>1</sup>, siehe Abb. 5).

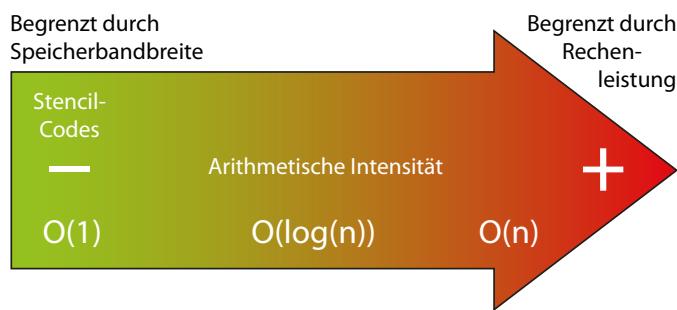


Abbildung 5: Arithmetische Intensität von Stencilcodes.

Stencilcodes sind daher speichergebunden und die Performanz wird durch die Speicherbandbreite begrenzt. Die Herausforderung liegt somit in einer effizienten Ausnutzung der verfügbaren Speicherbandbreite und folglich in einer effizienten Kommunikation.[Chr11, S.71ff][Nai11]

---

<sup>1</sup>in der Literatur finden sich unterschiedliche Angaben: Als Einheit wird entweder Byte oder ein Datum verwendet.

Stencilcodes finden ihre Anwendung in Lösungssystemen für partielle Differentialgleichungen, in der Entwicklung von Klima- und Wetter-Modellen, in komplexen physikalischen Simulationen z.B. für die Berechnung von Fluidynamik, in der Bildverarbeitung und medizinischen Problemstellungen.[CSC12][CSB11]

Abbildung 6 zeigt einige Stencil-Berechnungs-Muster für Beispiele, die dem PATUS-Quellcode beiliegen.

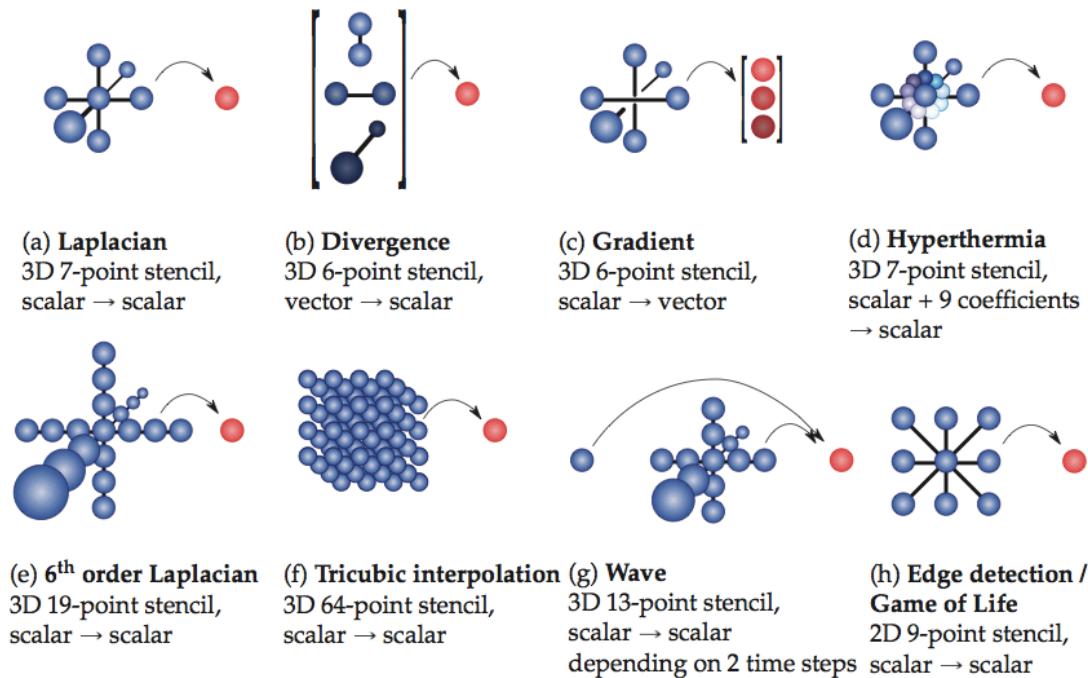


Abbildung 6: Verschiedene Stencilmuster. Die blauen Punkte bilden die Berechnungsgrundlage für den Gitterpunkt zum Zeitpunkt  $t+1$  (rot dargestellt).<sup>2</sup>

## 2.2 Feedback-basierte Optimierung

Bei PATUS wird feedbackbasierte Optimierung in Form von Auto-Tuning eingesetzt, um optimalen Quellcode für die jeweilige Zielplattform zu generieren. Hierfür werden Varianten des Quellcodes erzeugt, die mit einem Benchmark getestet werden. Die dem verwendeten Kostenmaß entsprechend optimale Variante, in diesem Fall, die mit der kürzesten Laufzeit, wird schließlich ausgewählt.

<sup>2</sup>Grafik aus [Chr11] übernommen.

## 2.3 Kerngedanken von PATUS

Folgende Kerngedanken standen bei der Entwicklung von PATUS Pate:

**Trennung von Berechnung und Traversierung** Bei PATUS wurde die Stencil-Berechnung in die Beschreibung der punktweisen Berechnung (*Stencil-specification*) und in die Traversierung des Gitters (*Strategy-Definition*) aufgeteilt. Dadurch ist es möglich, Stencil-Muster und Traversierungs-Strategien unabhängig voneinander wiederzuverwenden.

**Abstraktion von Plattformen** Architektspezifische Details werden in Backends gekapselt und erst in der Sourcecode-Generierung zum Quellcode ergänzt.

**Erweiterbarkeit durch Modularisierung** Die Architektur von PATUS ist modular aufgebaut und leicht um z.B. weitere Architekturen erweiterbar.

**Feedback-basierte Optimierung** Um optimierten Quellcode zu erhalten, wird in PATUS feedback-basierte Optimierung mittels *Auto-Tuning* eingesetzt. Hierfür werden Benchmark-Durchläufe von Varianten des Quellcodes durchgeführt, die in die Zielcode-Parametrisierung eingehen.

## 2.4 Schematischer Aufbau

PATUS ist in Java implementiert und benötigt mindestens Java 7 als Laufzeitumgebung. Außerdem benutzt es drei externe Bibliotheken:

Coco/R<sup>3</sup>, ein Compilergenerator unter GPL, wird benötigt um die Eingabedateien zu parsen. Das Cetus Framework<sup>4</sup> wird für die interne Darstellung des Quellcodes und für Source-to-Source-Transformationen benutzt, sowie um die Ausgabedateien zu generieren. Das Computeralgebrasystem Maxima<sup>5</sup> wird verwendet, um arithmetische Ausdrücke zu vereinfachen.

Der schematische Aufbau von PATUS ist in Grafik 7 abgebildet: Aus den Eingabedateien, einer Stencilspezifikation sowie Strategie-Definitionen generiert PATUS mittels plattformspezifischer Backends für eine Zielpfaltform optimierten C-Quellcode: Dazu gehören die *kernel.c*-Datei, die die Stencil-Berechnung enthält, ein *Makefile* und die *driver.c*-Datei, eine plattformspezifische *Benchmarking-Harness*. Bei letzterer handelt es sich um ein Codegerüst, über das ein Benchmark des Kernels erstellt werden kann. Dieser wird über den *Auto-Tuner* aufgerufen. Hierfür wurde der Kernel-Quellcode für die Berechnung mit zahlreichen Parametern ausgestattet bzw. es wurden Varianten der Berechnung erstellt. Die Werte für die

---

<sup>3</sup><http://www.ssw.uni-linz.ac.at/coco/>, Stand 02.06.2013

<sup>4</sup><http://cetus.ecn.purdue.edu/>, Stand 02.06.2013

<sup>5</sup><http://maxima.sourceforge.net/>, Stand 02.06.2013

Parameter und die Menge der Varianten bilden nun den Suchraum für den Auto-Tuner. Das Makefile enthält zu diesem Zweck mehrere Targets: Mit *make bench* wird aus der *kernel.c*- und *driver.c*-Datei ein Benchmark gebaut, der die möglichen Parameter auf der Konsole entgegen nimmt. *make tune* führt nun den Benchmark mittels des Auto-Tuners aus und versucht eine für die Zielplattform optimale Parametrisierung des Kernels zu ermitteln. Die Ergebnisse der Testläufe schreibt der Auto-Tuner in eine Header-Datei (*tuned\_params.h*), die zusammen mit dem Kernel in die Host-Anwendung eingefügt werden kann.

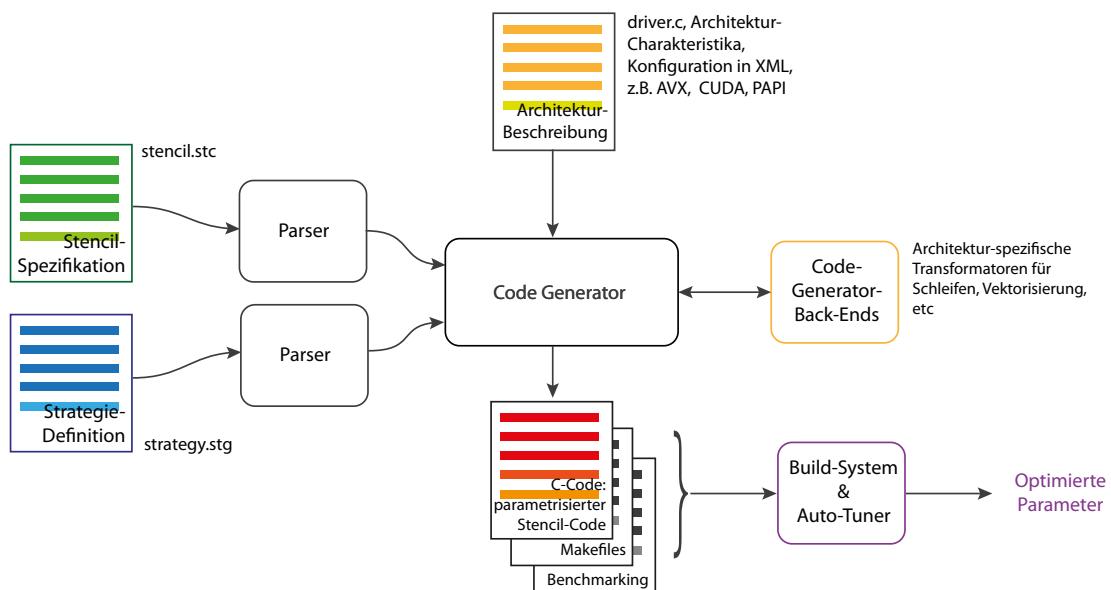


Abbildung 7: Der schematische Aufbau von PATUS.

Für das endgültige Programm kann der Quellcode mit handelsüblichen Compilern übersetzt werden, da es sich um standard-konformen C-Quellcode handelt.

Architektspezifische Details sind in den Backends des Codegenerators spezifiziert und werden über eine XML-Konfigurationsdatei bereitgestellt, über die ein Mapping von Funktionen oder Konzepten, z.B. Barriers, auf konkreten plattformspezifischen Implementierungen durchgeführt wird. Der Begriff Backend bezieht sich hier sowohl auf konkrete Hardware, wie die Nvidia CUDA-Architektur als auch auf Konzepte der Programmierung, wie OpenMP-PRAGMA-Direktiven. Das Framework ist stark modularisiert und auf Erweiterbarkeit ausgelegt, so dass weitere Architekturen leicht ergänzt werden können.[CSC12]

## 2.5 Projekthintergrund

PATUS ist im Rahmen der Doktorarbeit von Matthias-Michael Christen an der Universität Basel entstanden [Chr11] und wurde 2011 unter LGPL<sup>6</sup> veröffentlicht.<sup>7</sup> Das Framework befindet sich noch in der Entwicklung und liegt zum Schreiben der Arbeit in Version 0.1.3 *Alpha* vor. Es wird aktiv durch Matthias Christen weiterentwickelt und findet seinen Einsatz im Scientific Computing (siehe Kapitel 4.1, [SMS<sup>+</sup>09], [CSC12]) und in der Lehre [BCR<sup>+</sup>11].

## 3 Implementierungsdetails

Bei PATUS kommen zwei domänenspezifische Sprachen zum Einsatz. Die Stencil-Spezifikation beschreibt die punktweise Berechnung und die Strategy-Definition bestimmt, wie über das Gitter iteriert werden soll.<sup>8</sup>

### 3.1 Die Stencil-Spezifikation

Listing 1 zeigt den grundlegenden schematischen Aufbau einer Stencil-Spezifikation.

```
1 stencil name ( arguments )
2 {
3     iteration-space
4     [ number-of-timesteps ] operation
5     [ boundaries ]
6     [ initial ]
7 }
```

Listing 1: Schema einer Stencil-Spezifikation

Mit name wird der Funktionsname im späteren Kernel bestimmt und die Argumente entsprechen dem variablen Anteil im Stencil. Die zentralen Elemente sind der iteration-space und die Methode **operation**. Im iteration-space wird mittels des Keywords **domainsize** das Gitter selbst beschrieben und **operation** enthält in geschweiften Klammern die punktweise Berechnung.

Zusätzlich ist es möglich, die Anzahl der zeitlichen Iterationen über die Variable **t\_max** zu bestimmen. Standardmäßig wird das Gitter einmal traversiert. Momentan kann allerdings noch keine Abbruchbedingung formuliert werden (siehe Kap. 3.5). Mit **initial** kann angegeben werden, wie das Gitter initialisiert werden soll –

<sup>6</sup>GNU Lesser General Public License (<http://www.gnu.org/licenses/lgpl-3.0.en.html>), Stand 02.06.2013

<sup>7</sup><https://code.google.com/p/patus/>, Stand 02.06.2013

<sup>8</sup>Die folgenden Beispiele sind weitestgehend der Dokumentation und dem Quickstart-Guide aus dem PATUS Quellcode in SVN Revision 147 entnommen.

standardmäßig geschieht dies implizit – und boundaries legt eine spezielle Randbehandlung fest. Letzteres wird allerdings noch nicht optimiert und die Angabe einer Randbehandlung führt laut Autor zu Performanzseinbußen.

Stencil-Spezifikationen liegen entweder als *.stg*-Dateien vor oder können mittels PRAGMA-Anweisungen in C eingebettet werden (siehe Listing 2). In diesem Fall übernimmt PATUS die Rolle eines Präprozessors.

```

1 // C-Code
2 #pragma patus begin-stencil-specification
3 stencil [...]
4
5 }
6 #pragma patus end-stencil-specification
7 // C-Code

```

Listing 2: Eingebettete PATUS-DSL in C-Quellcode.

In Listing 3 sind unterschiedliche Möglichkeiten aufgezeigt, wie Gitter definiert werden können. Konstant und mit doppelter Genauigkeit (1), mit einfacher Genauigkeit unter Bereichsangabe (2), mehrdimensionale Arrays von Gitter (3) oder mehrdimensional mit Bereichsbeschränkung seitens der Arrayindizierung und der Dimensionen.

```

1 const double grid A
2 float grid B (-1 .. size_x)
3 const double grid C(0..sx, 1..sy, -2..sz+2)[3]
4 float grid D(min_x..max_x, min_y..max_y)[0..1, -1..3]

```

Listing 3: Unterschiedliche Gitter-Definitionen

Es existiert außerdem als syntaktischer Zucker eine Reduktion für Summe und Produkt. Listing 4 zeigt eine Zuweisung der Summe von Punkten des dreidimensionalen Gitters U zum Zeitpunkt  $t$  an das Gitter U zum Zeitpunkt  $t+1$ .

```

1 U[x,y,z; t+1] = { i=-1..1, j=-1..1, k=-1..1, r=1..2 : i^2+j^2+k^2==1 }
2   sum( U[x+r*i, y+r*j, z+r*k; t] );

```

Listing 4: Beispiel für eine Reduktionsoperation.

Die Positionen der für die Reduktion referenzierten Punkte werden mit  $i$ ,  $j$ ,  $k$ ,  $r$  bestimmt und mit der entsprechenden Ausdrücke in der Adressierung von  $U$  ermittelt. Zusätzlich ist durch  $i^2+j^2+k^2==1$  eine Bedingung an  $i$ ,  $j$ ,  $k$  geknüpft worden.

Listing 5 zeigt ein einfaches Beispiel für eine Stencilberechnung aus der Bildverarbeitung für eine Kantenerkennung. Es handelt sich dabei um einen 9-Punkt Stencil (siehe Abb. 8a) bestehend aus 10 FLOPS bei einer arithmetischen Intensität von 5.0.[Chr11, S.73]

---

<sup>8</sup>Grafiken aus [Chr11] S.67.

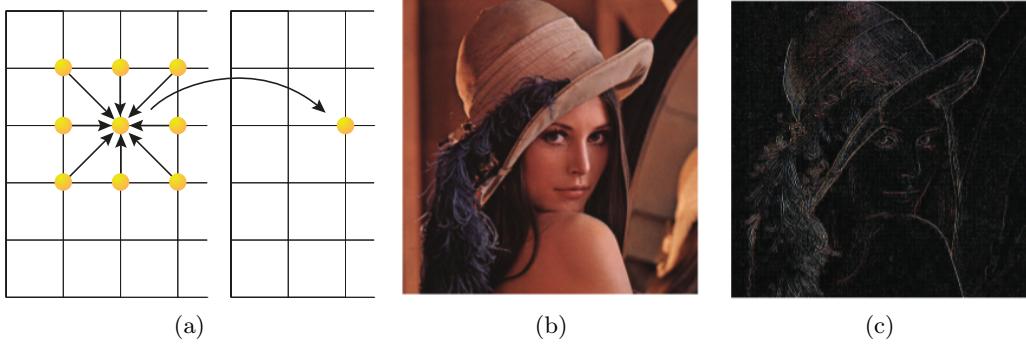


Abbildung 8: Kantenerkennung mit dem 9-Punkt Edge-Stencil.

Die Berechnungsgrundlage für den Stencil bildet die Matrix:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & -12 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

```

1 stencil edge (float grid U)
2 {
3   domainsize = (1 .. width, 1 .. height);
4
5   operation
6   {
7     U[x, y; t+1] = -12 * U[x, y; t]
8       + 2 * (U[x-1, y; t] + U[x+1, y; t]
9           + U[x, y-1; t] + U[x, y+1; t])
10      + U[x-1, y-1; t] + U[x+1, y-1; t]
11      + U[x-1, y+1; t] + U[x+1, y+1; t];
12   }
13 }
```

Listing 5: Stencil für eine Kantenerkennung

Der Stencil besitzt als Parameter nur das Grid der Bildpunkte in einfacher Genauigkeit. Die Größe des Iterationsraums (**domainsize**) wird über **width** und **height** automatisch inferiert.

### 3.2 Die Strategy-Definition

Strategien definieren, wie über Gitter traversiert wird. Insbesondere beschreiben sie die Art und Weise der Parallelisierung.

Listing 6 zeigt eine einfache Traversierungsstrategie, die dem Quellcode von

PATUS beiliegt: Es wird über das Grid iteriert und die äußerste Schleife parallelisiert.

```

1 strategy simple_parallel (domain u, auto int chunk = 1:u.size(1))
2 {
3   for t = 1 .. stencil.t_max
4   {
5     for point p in u(:, t) parallel schedule chunk
6       u[p; t+1] = stencil(u[p; t]);
7   }
8 }
```

Listing 6: Eine einfache Traversierungs-Strategie.

Dabei leitet **strategy** eine Traversierungsstrategie ein. Mit **domain** wird das zu traversierende Gitter beschrieben. Es folgt ein Parameter vom Typ *Integer*. Das Schlüsselwort **auto** veranlasst PATUS, den Parameter **chunk** für den Auto-Tuner verfügbar zu machen. Die **for**-Schleife dient zur Iteration über die zeitliche Dimension (Zeile 3): **stencil.t\_max** liefert hierfür die Anzahl der Durchläufe. Es folgt die äußere Schleife der Stencilberechnung. Es wird über jeden **point** p des Gitters U iteriert. Das Schlüsselwort **parallel** leitet die Parallelisierung ein und mit dem Schlüsselwort **schedule** kann die Größe der Datenblöcke bestimmt werden, die jedem Thread zugeordnet werden soll. Diese wird später über den Parameter **chunk** durch den Auto-Tuner variiert.

In Zeile 6 wird die eigentliche Berechnung des Gitterpunktes für den Zeitpunkt  $t+1$  angestoßen.

In Listing 7 ist ein weiteres Beispiel zu sehen, in dem eine Cache Blocking Strategie (siehe Kap. 3.3) umgesetzt ist: Zu diesem Zweck wird die Domäne der Gitterpunkte in Subdomänen unterteilt. **auto dim cb** gibt die Bestimmung der Größe der Dimensionen für den Auto-Tuner frei. Die entsprechenden Parameter werden automatisch abhängig von den Dimensionen des Gitters des Stencils erzeugt.

```

1 strategy cacheblocked_domain (domain u, auto dim cb, auto int chunk)
2 {
3   for t = 1 .. stencil.t_max
4   {
5     for subdomain v(cb) in u(:, t) parallel schedule chunk
6     {
7       for point p in v(:, t)
8         v[p; t+1] = stencil(v[p; t]);
9     }
10  }
```

Listing 7: Eine Cache-Blocking Strategie.

Mit **subdomain** v(cb) in Zeile 5 werden die entsprechenden Blöcke angelegt, über die dann pro Knoten iteriert wird. Der Parameter chunk legt hierbei die Anzahl der aufeinanderfolgenden Blöcke pro Thread fest.[Chr11, S.115ff]

### 3.3 Statische Optimierungen

PATUS erzeugt plattformspezifische Optimierungen auf Quellcodeebene. Es findet eine Parallelisierung des Quellcodes auf Task-Ebene durch OpenMP-PRAGMA-Anweisungen und auf Daten-Ebene entweder durch SIMD-Instruktionen für eine Vektorisierung oder durch die Erstellung eines Kernels für GPU-Computing statt. Der Fokus der Optimierungen liegt auf Vektorisierung, Loop-Unrolling, Cache-Blocking und NUMA-Awareness, die im Folgenden vorgestellt werden.

#### Vektorisierung

Der Quellcode kann durch *SSE*- (128 Bit Registerbreite) und *AVX-Instruktionen* (256 Bit) optimiert werden. PATUS unterstützt zudem die *Intel® MIC-Architektur*<sup>9</sup>, einer x86-kompatiblen Many-Core-Architektur, die Vektorbefehle mit 512 Bit Breite besitzt.<sup>10</sup> Der Quellcode kann sowohl mit Inline-Assembler-Anweisungen als auch mit den Vektoranweisungen der C-Laufzeitumgebung generiert werden. Der Vorteil der Inline-Assembler-Anweisungen liegt in einer Reduzierung der für die Berechnung notwendigen Instruktionen und effizienteren Indexkalkulationen auf der Zielplattform.[CSC12]

#### Loop-Unrolling

Ein weiterer Fokus liegt auf *Loop-Unrolling*. Beim *Loop-Unrolling* werden Schleifeniterationen „abgerollt“, indem die Anweisungen des Schleifenrumpfes wiederholt hintereinander aufgeführt werden. Die Schleife wird entweder vollständig entfernt oder die Anzahl der Iterationen wird reduziert. Somit wird der Overhead, der durch den Schleifenkopf erzeugt wird, vermieden oder zumindest gemindert bei allerdings gleichzeitig erhöhter Instruktionsanzahl. Die abgerollten Anweisungen können im weiteren Verlauf durch den Compiler optimiert werden. Die Schwierigkeit liegt in der Balance zwischen geringerem Schleifenoverhead und der Anzahl der hinzugekommenen Instruktionen. Wird die Anzahl der Instruktionen zu stark erhöht, dann kann dies Cache-Misses im (L1-)Instruktionscache verursachen. Das Verhältnis wird durch den *Abrollfaktor* festgelegt, der in Form eines Parameters

---

<sup>9</sup>Intel Many Integrated Core Architecture Multiprozessor Architektur

<sup>10</sup>[https://en.wikipedia.org/wiki/Intel\\_MIC](https://en.wikipedia.org/wiki/Intel_MIC), <http://www.intel.com/content/www/de/de/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>, Stand 16.06.2013.

dem *Auto-Tuner* zur Verfügung gestellt wird (siehe Kap. 3.4). PATUS ergänzt automatisch sog. *Clean-up Loops*, wenn die Anzahl der Iterationen nicht durch den Unrolling-Faktor teilbar ist.[Chr11, S.218ff][CSC12]

### Vertikale Speicherhierarchie

PATUS berücksichtigt im besonderen Maße die Cache-Hierarchie: Mit *Cache-Blocking*, einer Form des Tilings, werden die pro Prozess zu verarbeitenden Daten abhängig vom Prozessor-Cache der Zielplattform partitioniert. Das Ziel ist Ladeinstruktionen und Cache-Misses zu minimieren, indem die Cache-Nutzung maximiert wird, um so die Speicherbandbreite möglichst effizient auszunutzen. In Kapitel 3.2 wird ein Codebeispiel gezeigt, das ein Gitter entsprechend aufteilt.

Eine weitere Cache-Optimierung, das Cache-Bypassing, bezieht NUMA-Architekturen mit ein:

### NUMA-Awareness

Bei *NUMA (Non-Uniform Memory Access)* handelt es sich um ein verbreitetes Modell für Multiprozessorsysteme mit gemeinsamen Speicher. Der globale Speicher ist dabei in der Anzahl der Prozessoren partitioniert und jede Partition lokal an einen Prozessor angebunden. Die Prozessoren sind untereinander z.B. über einen Bus verbunden, bei AMD Opteron Systemen z.B. über eine Hypertransport-Verbindung<sup>11</sup>. Die Zugriffszeit auf ein Datum hängt somit in großem Maße davon ab, ob es im lokal angebundenen Speicher liegt, oder ob ein Umweg über einen anderen Prozessor genommen werden muss.[vgl. RR07, S.28, S.31f] Die folgenden Optimierungen versuchen diesen Umstand zu berücksichtigen. Sie werden bei PATUS unter dem Begriff *NUMA-Awareness* zusammengefasst.

1. *Thread-Affinity*. Durch z.B. Compiler-spezifische Direktiven, wie Intel- oder GNU-spezifische OpenMP-PRAGMA-Anweisungen, werden Prozesse an einen bestimmten Prozessor gebunden, um Daten-Migrationen von einem Speicherbereich in den Speicherbereich eines anderen Prozessors zu vermeiden.
2. *Node-Initialisierung*. Hierbei werden die von einem Prozessor genutzten Datensätze im Speicher von dem Knoten, auf dem der Prozess ausgeführt werden soll, initialisiert.
3. *Cache-Bypassing*. Werden Datensätze nicht mehr benötigt, dann wird bei diesem Verfahren der Cache des Prozessors umgangen. *Cache-Bypassing* hat

---

<sup>11</sup><http://www.amd.com/uk/products/technologies/hypertransport-technology/Pages/hypertransport-technology.aspx>, <http://de.wikipedia.org/wiki/HyperTransport>, Stand 16.06.2013.

einen großen Effekt auf *cc-NUMA-Systemen* (*cache coherent-NUMA*). Hierbei handelt es sich um NUMA-Systeme, die mit einem Cache-Kohärenzmechanismus ausgestattet sind, der die Caches der einzelnen Prozessoren abgleicht. Diese Verfahren können ein großes Datenaufkommen auf dem Bus verursachen.[RR07, S.31f]

[Chr11, S.107ff, S.190]

### Optimierung durch Compiler für Zielplattform

Da PATUS standardkonformen C-Code erzeugt, kann der Zielcode mit den zahlreichen Optimierungen handelsüblicher Compiler weiter optimiert werden.

## 3.4 Auto-Tuning

Beim Auto-Tuning handelt es sich um eine Form der feedback-basierten Optimierung. Hierfür werden Varianten des Quellcodes erzeugt, die mit einem Benchmark getestet werden. Die dem verwendeten Kostenmaß entsprechend optimale Variante, in diesem Fall, die Schnellste, wird schließlich ausgewählt.

Um die Varianten zu erhalten, wird der Quellcode parametrisiert, bzw. im Fall des *Loop-Unrolling* (siehe Kap. 3.3), werden mehrere Varianten der Funktion erzeugt. Der Kernel-Code wird darauffolgend in ein Benchmarking-Gerüst, der *Benchmarking-Harness*, eingebettet, über das der Kernel mittels des *Auto-Tuners* mit entsprechenden Parametern aufgerufen werden kann. Die *Benchmarking-Harness* besteht aus einer plattformspezifischen *driver.c*-Datei – ein C-Quellcode-Template – das Teil des jeweiligen Architektur-Backends ist.

Die Schwierigkeit des Auto-Tunings besteht in der Traversierung des Suchraums: Da es sich um das kartesische Produkt der Parameterräume handelt, wächst der Suchraum exponentiell in der Anzahl der Parameter. PATUS bringt zu diesem Zweck mehrere Strategien zur Traversierung des Suchraums mit:

- *Exhaustive-Search*: Diese Methode läuft den Suchraum vollständig ab.
- Zwei Greedy Algorithmen: Eine *Heuristik*, die vom Aufwand additiv in der Anzahl der Parameter ist, und die *General Combined Elimination*, bei der versucht wird, Parameter, die einen großen positiven Einfluss auf die Performance haben, früh zu fixieren.
- Der *Hooke-Jeeves Algorithmus*: Hier wird von dem aktuell besten Wert der Suchraum in diskreten Schritten entlang der Koordinatenachsen erforscht.
- Die *Powell's Methode*: Eine Erweiterung der Greedy-Algorithmen.

- Die *Nelder-Mead-Methode* oder *Downhill-Simplex-Search*: Dabei wird im Suchraum ein *Simplex* aufgespannt. Hierbei handelt es sich um ein Polytop mit  $n + 1$ -Ecken bei  $n$  Parametern. Die Ecken entsprechen den gegenwärtig optimalen Punkten, sortiert nach dem aktuellen Optimum. Das Simplex wird im Laufe der Suche transformiert – schlechte Werte werden durch Neue ersetzt – und um die besten Werte verkleinert, bis es lediglich den besten Wert enthält.
- Die *DIRECT-Methode*: Hierbei wird der Suchraum mit einem Gitter unterteilt. Zellen mit den besten Werten an den Ecken werden rekursiv zerteilt und spannen neue Untersuchräume auf.
- Ein *Genetischer Algorithmus*: Dieser basiert auf einer stochastischen Erkundung des Suchraums. Es werden Punkte, also Parameter-Kombinationen, im Suchraum ausgewählt, die die erste *Population* bilden. Danach wird nach den Prinzipien der Evolutionstheorie, *Mutation* – der Veränderung eines Wertes eines Parameters – und *Rekombination* – der Kreuzung zweier Punkte – die Parameter-Kombinationen verändert. Es folgt die *Selektion* der Punkte des Suchraums über eine *Fitness-Funktion*, die hier dem Benchmarking-Programm entspricht. Die ausgewählten Punkte bilden die Population des nächsten Durchlaufs.[Kra09, S.13ff]

Abbildung 9 zeigt eine Evaluation der Ergebnisse bzgl. der unterschiedlichen implementierten Algorithmen an.

Eine detaillierte Beschreibung der Algorithmen inkl. Pseudocode findet sich in [Chr11, S.131ff].

Zum jetzigen Zeitpunkt wird der genetische Algorithmus favorisiert, der mittels der *jgap*-Bibliothek<sup>12</sup> verwirklicht ist. Dieser erzielt trotz längster Optimierungsdauer die besten Ergebnisse bzgl. des Daten-Durchsatzes und setzt sich bei hoher Anzahl von Benchmarking-Läufen deutlich von den anderen Traversierungsstrategien ab. Zudem ist hier die Gefahr kleiner, in einem lokalen Optimum stecken zu bleiben, als bei den Greedy-Algorithmen. Dennoch haben die anderen Algorithmen durchaus ihre Daseinsberechtigung. Ist aufgrund eingeschränkter Parameter die Beschaffenheit des Suchraums im Vorhinein bekannt, so können bei Monotonie Greedy-Algorithmen zu einer schnelleren Konvergenz führen und bei einem sehr kleinen Suchraum gelangt die Exhaustive-Search garantiert zu einem globalem Optimum.

Der Auto-Tuner von PATUS ist als Stand-alone-Programm auch für externe Projekte nutzbar und ermöglicht es, die Ergebnisse der Durchläufe durch eine *Gnuplot*-kompatible Ausgabe zu visualisieren.

---

<sup>12</sup>Grafik aus [Chr11, S.144f] übernommen.

<sup>13</sup><http://jgap.sourceforge.net/>, Stand 16.06.2013.

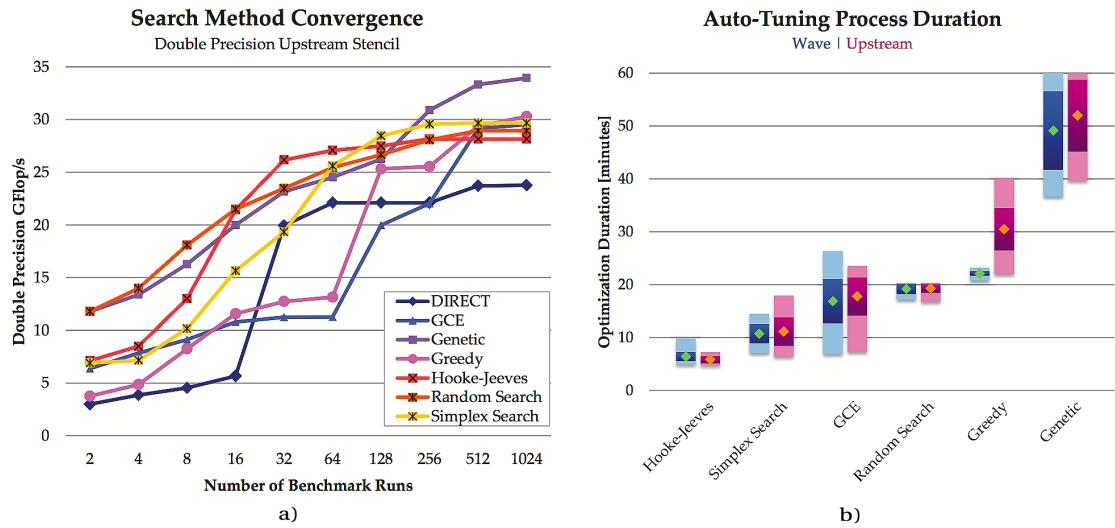


Abbildung 9: Evaluation verschiedener Suchraumtraversierungsstrategien.<sup>12</sup> Für die Evaluation wurde ein Stencilcode für die Wellengleichung und ein Laplace-Stencil 6. Ordnung (*Upstream*) eingesetzt (siehe Abb. 2.1 e), g). Abbildung a) zeigt das Konvergenzverhalten der implementierten Algorithmen, Abbildung b) die benötigte Zeit zur Traversierung des Suchraumes.

### 3.5 Einschränkungen

Da sich PATUS in einem frühen Entwicklungsstand befindet, unterliegt es noch einigen Einschränkungen, die im Folgenden aufgeführt sind<sup>14</sup>:

- Es wird nur die Jacobi-Iteration unterstützt: Alle für die Berechnung benötigten Punkte werden zum Zeitpunkt  $t$  gelesen und zum Zeitpunkt  $t+1$  geschrieben. Es ist nicht möglich, Datenabhängigkeiten zwischen Punkten bzw. Iterationen zu berücksichtigen, wie es Gauss-Seidel/Rot-Schwarz-Traversierungen erlauben. Insbesondere muss die Reihenfolge der Berechnungen irrelevant sein.
- Es werden alle Punkte des Gitters abgelaufen.
- Die Anzahl der Traversierungen des Gitters muss konstant sein. Für ein bedingtes Traversieren, z.B. wenn bei den Berechnungen ein Schwellwert erreicht wird, ist eine Reduktion über die parallel ausgeführten Berechnungen nötig. Ein entsprechendes Konstrukt befindet sich gerade in der Entwicklung.
- Es findet noch keine Optimierung bei Randbedingungen statt, so dass Berechnungen mit spezieller Randbehandlung mit Einbußen in der Performanz

---

<sup>14</sup>[vgl. Chr11, S.83f]

einhergehen.

- Es sind keinen bedingten Zuweisungen bei den Stencilberechnungen möglich.
- Es werden nur Systeme mit gemeinsamen Speicher unterstützt. Allerdings ist es möglich Quellcode zu erstellen, der pro Knoten ausgeführt wird (siehe Kap. 4.1).
- Die Unterstützung für CUDA befindet sich noch im Proof-of-Concept-Status.
- Für Fortran und CUDA werden momentan nur Kernel mit einem Zeitschritt unterstützt.

## 4 Anwendungsbeispiele und Benchmarks

### 4.1 Large-Scale Erdbeben Simulation: AWP-ODC

Das folgenden Beispiel wurde am *South California Earthquake Center (SCEC)* entwickelt. Die dort entwickelte Anwendung, *Anelastic Wave Propagation Code (AWP-ODC)*, ist eine Erdbeben-Simulation am San Andreas Graben in Kalifornien. Die Anwendung hatte das Ziel eine seismische Gefährdungskarte für den gesamten Staat Kalifornien zu erstellen. Sie ist mit MPI implementiert und extrem skalierbar: Im Produktionslauf bestand eine Dauerbelastung von 220 TFLOP/s für 24h auf 223.074 *Jaguar-Kernen*<sup>15</sup> des *Oak Ridge National Laboratory XT5*<sup>16</sup>, einem Supercomputer von Cray Inc.<sup>17</sup> mit 1,382 PFLOPS. Die erzeugte Simulation berechnete ein Erdbeben der Stärke 8 bei 2 Hz und einer Dauer von 360 Sekunden.

Die Anwendung basiert weitestgehend auf der Lösung der partiellen Differentialgleichungen

$$\frac{\delta v}{\delta t} = \frac{1}{\rho} \nabla \sigma, \frac{\delta \sigma}{\delta t} = \lambda(\nabla v)I + \mu(\nabla v \nabla v^T).$$

Die abhängigen Variablen sind der Geschwindigkeitsvektor  $v$  und Stress-Tensor  $\sigma$ .  $\lambda$  und  $\mu$  sind die Lamé-Koeffizienten,  $\rho$  die Dichte und  $I$  der Identitätstensor. Es ergeben sich zwei Stencils, „UXX“ und „ALLQ“, bestehend aus 5 und 33 Punkten, die 20 und 181 FLOPs erzeugen, bei einer arithmetischen Intensität von 1,0 bzw. 1,37 FLOPs/übertragenem Datum.

---

<sup>15</sup>Der Cray XT5 Jaguar-Supercomputer führte die Rangliste der TOP 500 im November 2009 an: <http://top500.org/lists/2009/11/>, Stand 16.06.2013.

<sup>16</sup><http://www.ornl.gov/>, Stand 16.06.2013.

<sup>17</sup><http://www.cray.com>, Stand 16.06.2013.

<sup>18</sup>Grafik aus [Chr11] S.181 übernommen.

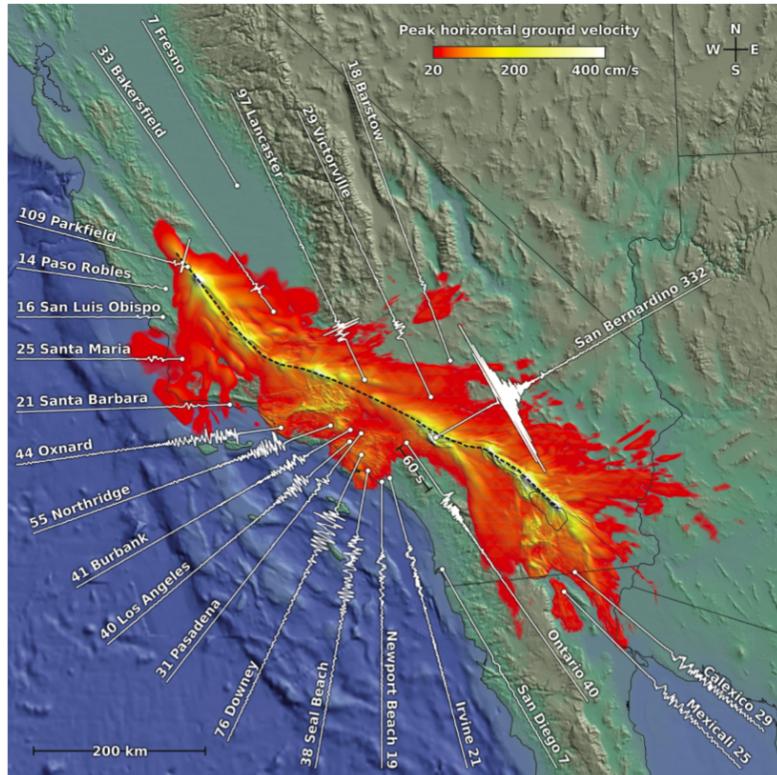


Abbildung 10: Erdbebensimulation am San Andreas Graben in Kalifornien.<sup>18</sup>

Für eine Evaluation wurde die Berechnung des AWP-ODC, die pro MPI-Knoten ausgeführt wird, in PATUS-DSL überführt. Als Compiler dienten der Intel Fortran Compiler 12.1.0, und GCC 4.6.1 für die PATUS-Kernel sowie OpenMPI in der Version 1.4.1. Der Benchmark wurde auf einem Intel Xeon Sandy Bridge System mit zwei E5-2670 CPUs mit je 8 Kernen ausgeführt. Dabei wurden zwei von der Anwendung unterstützte Modi ausgeführt, *viscous* und *elastic*.

Abbildung 11 zeigt die Ergebnisse eines Weak-Scaling Tests<sup>20</sup>: Es wurde der Original- und der mit PATUS erstellte Code für 1 bis 512 MPI-Prozessen bei einer lokalen Problemgröße von je  $256^3$ -Gitterpunkten getestet. Die vertikalen Balken zeigen die benötigte Zeit für die Berechnung.

Dabei konnte PATUS durchgehend einen Speedup gegenüber der Original-Implementierung erzielen. Je nach Konfiguration lag dieser zwischen 4% und 15%. Die Grafik zeigt einen Performanz-Verlust, der zwischen 1 bis 8 MPI-Prozessen immer stärker wird. Dieser trat aufgrund des gemeinsamen L3-Caches zwischen den Kernen der einzelnen Prozessoren auf. Für den Ausschlag bei 256 MPI-Prozessen

<sup>19</sup>Grafik aus [CSC12].

<sup>20</sup>Bei Weak-Scaling-Tests wächst die Problemgröße proportional zur Anzahl hinzugefügter Ressourcen.

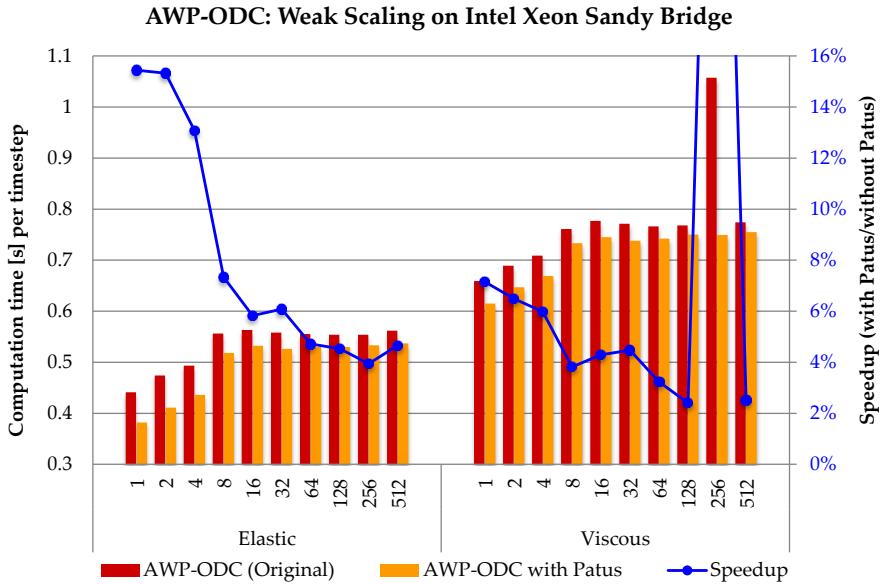


Abbildung 11: Testergebnisse des modifizierten AWP-ODC-Codes.<sup>19</sup>

ist keine Erklärung gefunden worden.

Eine umfassende Beschreibung findet sich in [CSC12][Chr11, vgl S.180ff].

## 4.2 Testergebnisse

Die folgenden Benchmark-Ergebnisse stammen aus [CSC12]. Es standen zwei Testumgebungen zur Verfügung: Das erste Testsystem ist Teil des Cray XE6 Jaguar Systems und basiert auf der AMD Opteron Interlagos Plattform. Es handelt sich um eine Dual-Sockel NUMA-Architektur mit je 2 Dies pro Sockel. Jedes Die enthält jeweils 4 Module, wobei pro Modul jeweils zwei Integer- und eine Floating Point Unit vorhanden sind. Die gemessene Speicherbandbreite beträgt 44 GB/s. Insgesamt standen 32 Hardware-Threads zur Verfügung. Das zweite System besteht aus zwei Xeon E5-2670 Prozessoren der Intel Sandy Bridge Plattform, mit je 8 Kernen pro Sockel. Die gemessene Speicher-Bandbreite ergab 62 GB/s pro Sockel. Beide Systeme unterstützen Vektorisierung mit SSE- und AVX-Befehlssätzen.

Es wurden sechs Kernel getestet, die als C-Referenzimplementierung und in Form von PATUS-DSL vorlagen (Anzahl der FLOPS und arithmetische Intensität sind in Klammern aufgeführt). Einen einfachen Laplace- (8 FLOPs, 1,0) und einen erweiterten Laplace Kernel, namens GSRB (29 FLOPs, 3,63), einen Wave- (16 FLOPs, 1,33) und Blur-Kernel (31 FLOPs, 3,88) sowie die Kernel des AWP-ODC aus Kapitel 4.1: UXX (20 FLOPs, 1,0) und ALLQ (181 FLOPs, 1,37).

<sup>21</sup>Grafik aus [CSC12].

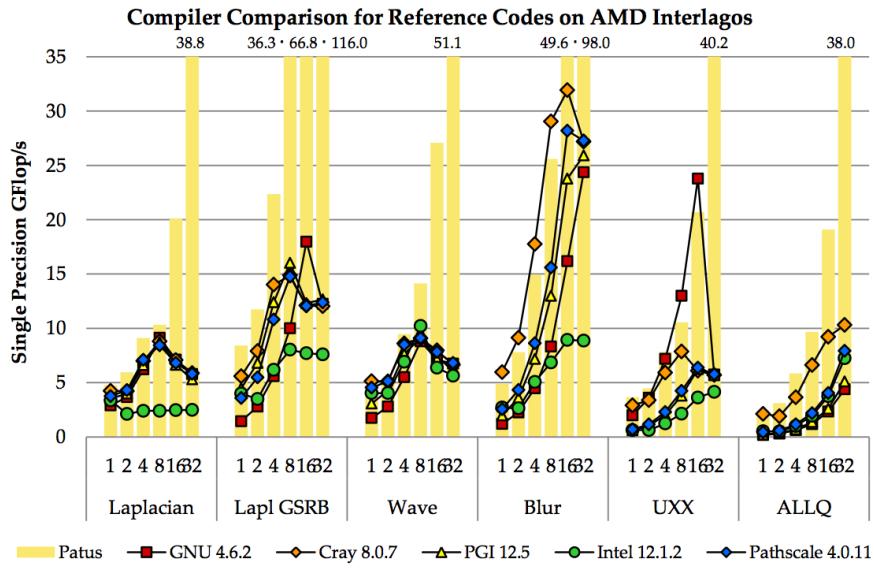


Abbildung 12: Testergebnisse auf der AMD Interlagos-Plattform.<sup>21</sup>

Diagramm 12 zeigt die Testergebnisse unter der AMD Interlagos Plattform. Die Referenzimplementierung wurde hierfür mit verschiedenen Compilern übersetzt. Die gelben Balken zeigen die Ergebnisse von dem mit PATUS optimierten Code. Der Quellcode wurde immer mindestens mit  $-O3$  kompiliert.

Es ist festzustellen, dass der mit PATUS optimierte Code deutlich besser skaliert als die Referenzimplementierung unabhängig vom verwendeten Compiler. Dabei produziert der native Compiler *Cray* im Schnitt bessere Ergebnisse als die anderen Compiler. Die Referenzimplementierung hatte unter allen Compilern beim Übergang von 16 auf 32 Hardware Threads einen enormen Performanceinbruch, als eine weitere NUMA-Domain hinzugefügt wurde. Lediglich der durch PATUS optimierte Code kann hier durch die in Kapitel 3.3 unter dem Stichwort *NUMA-Awareness* genannten Optimierungen weiter zulegen.

Abbildung 13 zeigt die Testergebnisse auf der Intel Sandy Bridge Plattform. Hier ist eine Aufschlüsselung des Performanzzuwachses anhand der verwendeten Optimierungen zu sehen. Es ist erneut festzustellen, dass der mit PATUS optimierte Code in der letzten Stufe beim Übergang von 8 auf 16 Kerne wesentlich besser skaliert als die Referenzimplementierung. Für die weitestgehend sehr guten Ergebnisse durch den Intel Compiler (unausgefüllte Raute) war eine manuelle Ergänzung von Intel-spezifischen PRAGMA-Anweisungen notwendig, wodurch explizit eine Optimierung durch SIMD-Befehle angestoßen wurde.

<sup>22</sup>Grafik aus [CSC12].

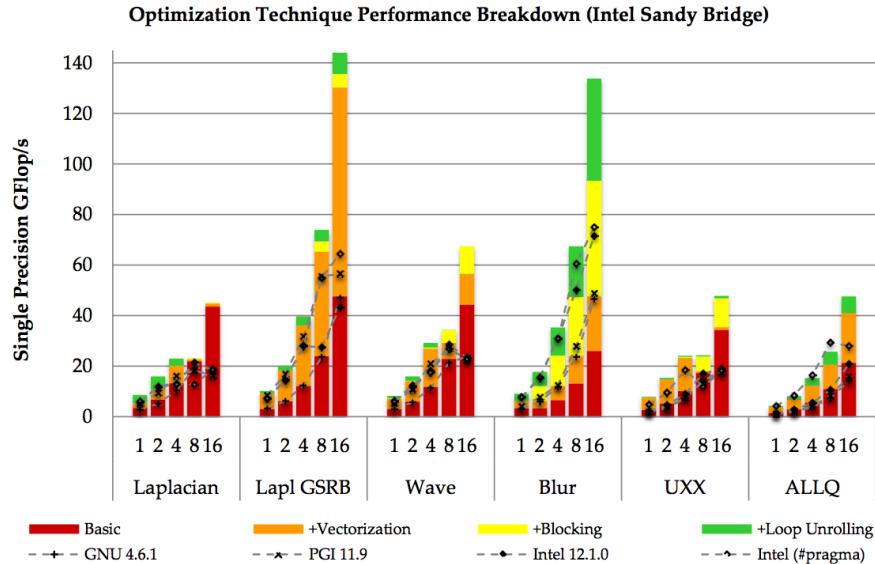


Abbildung 13: Testergebnisse auf der Intel Sandy Bridge-Plattform.<sup>22</sup>

## 5 Fazit und Aussicht

Trotz seines frühen Entwicklungsstands kann PATUS bereits jetzt viele der angestrebten Ziele erfüllen.

Durch die Trennung von punktweiser Spezifikation und Traversierung des Gitters wird die Stencil-Berechnung genauso wie die Strategie zum Iterieren des Gitters von unnötigem Ballast befreit. Sie sind kompakt formulierbar und leicht zu lesen.

Hierzu tragen maßgeblich die benutzten domänenspezifischen Sprachen bei: Die DSLs besitzen eine ausdrucksstarke Syntax, mit der leicht verständlicher Quellcode geschrieben werden kann. Sie erlauben eine kurze Einarbeitungszeit und – verglichen mit üblicher Programmierung im HPC-Bereich mittels MPI und OpenMP – ermöglichen es, die Produktivität deutlich zu steigern. Ebenfalls hervorzuheben ist die leichte Bedienbarkeit, die gute Dokumentation mit zahlreichen Beispielen und die Möglichkeit, die Stencil-DSL in C-Quellcode einzubetten.

Der durch die DSLs formulierte Quellcode ist außerdem vollkommen plattformunabhängig. Da in den Benchmarks gezeigt werden konnte, dass der generierte C-Quellcode zudem gut skaliert, kann ebenfalls eine gute Performanzportabilität erreicht werden. Die Testergebnisse zeigten zudem, dass aktuelle Compiler-Technik nicht in der Lage ist, ähnlich gute Ergebnisse automatisch zu erzeugen.

Aktuelle Forschung konzentriert sich momentan auf eine verbesserte Unterstützung des GPU-Computings. Hier zeigt sich ein großes Potential, da wie in

den Benchmarks zu sehen, Stencil-Berechnungen bereits jetzt stark von Vektorisierung auf CPUs, also Datenparallelität, profitieren. Aufgrund der viel höheren Speicherbandbreite von GPGPUs ist ein großer Performanzgewinn zu erwarten.

Bereits in der Doktorarbeit wird *temporales Blocking* intensiv diskutiert. Bei dieser Form der Optimierung wird die zeitliche Dimension beim Cache-Blocking berücksichtigt, sodass mehrere zeitliche Iterationen auf einem Knoten berechnet werden und Ergebniswerte, die als Eingabewerte der nächsten Iteration dienen, nicht erneut aus dem Speicher geladen werden müssen.

Weitere Forschung liegt im Bereich intelligenter auf Performanz-Modellen basierender Auto-Tuner, der Komposition von Stencils und einer verbesserten Integration von PATUS in bestehende Toolchains.

## Abbildungsverzeichnis

1	Ergebnisse des Fragebogens der Studie Run, stencil, run (Grafik aus [BCR <sup>+</sup> 11]). . . . .	2
2	Benchmarkergebnisse der implementierten Lösungen der Studenten. Gelb hervorgehoben: Benchmarkergebnisse von PATUS. Schmale schwarze/rote Balken zeigt die Streuung der Performanz der Lösungen (Grafik aus [BCR <sup>+</sup> 11], bearbeitet). . . . .	3
3	Quellcodeerstellung mit PATUS. . . . .	4
4	Muster für einen 9-Punkt Stencilcode. . . . .	5
5	Arithmetische Intensität von Stencilcodes. . . . .	5
7	Der schematische Aufbau von PATUS. . . . .	8

## Listings

1	Schema einer Stencil-Spezifikation . . . . .	9
2	Eingebettete PATUS-DSL in C-Quellcode. . . . .	10
3	Unterschiedliche Gitter-Definitionen . . . . .	10
4	Beispiel für eine Reduktionsoperation. . . . .	10
5	Stencil für eine Kantenerkennung . . . . .	11
6	Eine einfache Traversierungs-Strategie. . . . .	12
7	Eine Cache-Blocking Strategie. . . . .	12

## Literaturverzeichnis

- [ABC<sup>+</sup>06] ASANOVIC, Krste ; BODIK, Ras ; CATANZARO, Bryan C. ; GEBIS, Joseph J. ; HUSBANDS, Parry ; KEUTZER, Kurt ; PATTERSON, David A. ; PLISHKER, William L. ; SHALF, John ; WILLIAMS, Samuel W. ; YELICK, Katherine A.: The Landscape of Parallel Computing Research: A View from Berkeley / EECS Department, University of California, Berkeley. Version: Dec 2006. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>. 2006 (UCB/EECS-2006-183). – Forschungsbericht
- [BCR<sup>+</sup>11] BURKHART, H. ; CHRISTEN, M. ; RIETMANN, M. ; SATHE, M. ; SCHENK, Olaf: Run, Stencil, Run! A Comparison of Modern Parallel Programming Paradigms. In: *PARS-Mitteilungen 2011*, 2011
- [Chr11] CHRISTEN, Matthias-Michael: *Generating and Auto-Tuning Parallel Stencil Codes*. Basel, Philosophisch-Naturwissenschaftlichen Fakultät der Universität Basel, Diss., 2011
- [Chr12] CHRISTEN, Matthias-Michael: *PATUS Quickstart*. v.0.1, November 2012
- [Chr13] CHRISTEN, Matthias: *PATUS: CODE GENERATION AND AUTO-TUNING FOR PARALLEL STENCIL COMPUTATIONS*. 2013
- [CSB11] CHRISTEN, M. ; SCHENK, O. ; BURKHART, H.: PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In: *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011. – ISSN 1530–2075, S. 676–687
- [CSC12] CHRISTEN, Matthias ; SCHENK, Olaf ; CUI, Yifeng: Patus for convenient high-performance stencils: evaluation in earthquake simulations. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2012 (SC '12). – ISBN 978–1–4673–0804–5, 11:1–11:10
- [Doe12] DOERNER, Wendy: *Cache Blocking Techniques*. [software.intel.com/en-us/articles/cache-blocking-techniques](http://software.intel.com/en-us/articles/cache-blocking-techniques). Version: 08 2012, Abruf: 29.04.2013

- [Fow10] FOWLER, M.: *Domain-Specific Languages*. Pearson Education, 2010 (Addison-Wesley Signature Series (Fowler)). [http://books.google.de/books?id=ri1muolw\\_YwC](http://books.google.de/books?id=ri1muolw_YwC). – ISBN 9780131392809
- [Kra09] KRAMER, Oliver: *Computational Intelligence, Eine Einführung*. Berlin : Springer DE, 2009. – ISBN 978-3-540-79739-5
- [Nai11] NAIR, Ravi: Exascale Computing. Version: 2011. [http://dx.doi.org/10.1007/978-0-387-09766-4\\_284](http://dx.doi.org/10.1007/978-0-387-09766-4_284). In: PADUA, David (Hrsg.): *Encyclopedia of Parallel Computing*. Springer US, 2011. – ISBN 978-0-387-09765-7, 638-644
- [Pad11] PADUA, David A. (Hrsg.): *Encyclopedia of Parallel Computing*. Springer, 2011. – ISBN 978-0-387-09765-7
- [RR07] RAUBER, T. ; RÜNGER, G.: *Parallele Programmierung*. Springer, 2007. – ISBN 9783540465492
- [SMS<sup>+</sup>09] SCHENK, Olaf ; MANGUOGLU, Murat ; SAMEH, Ahmed ; CHRISTEN, Matthias ; SATHE, Madan: Parallel scalable PDE-constrained optimization: antenna identification in hyperthermia cancer treatment planning. In: *Computer Science - Research and Development* 23 (2009), Nr. 3-4, 177-183. <http://dx.doi.org/10.1007/s00450-009-0080-x>. – DOI 10.1007/s00450-009-0080-x. – ISSN 1865-2034