Das Polyedermodell vor dem Durchbruch?

Armin Größlinger
Erlangen, 30. März 2011

Lehrstuhl für Programmierung Prof. Christian Lengauer, Ph.D.
Fakultät für Informatik und Mathematik
Universität Passau

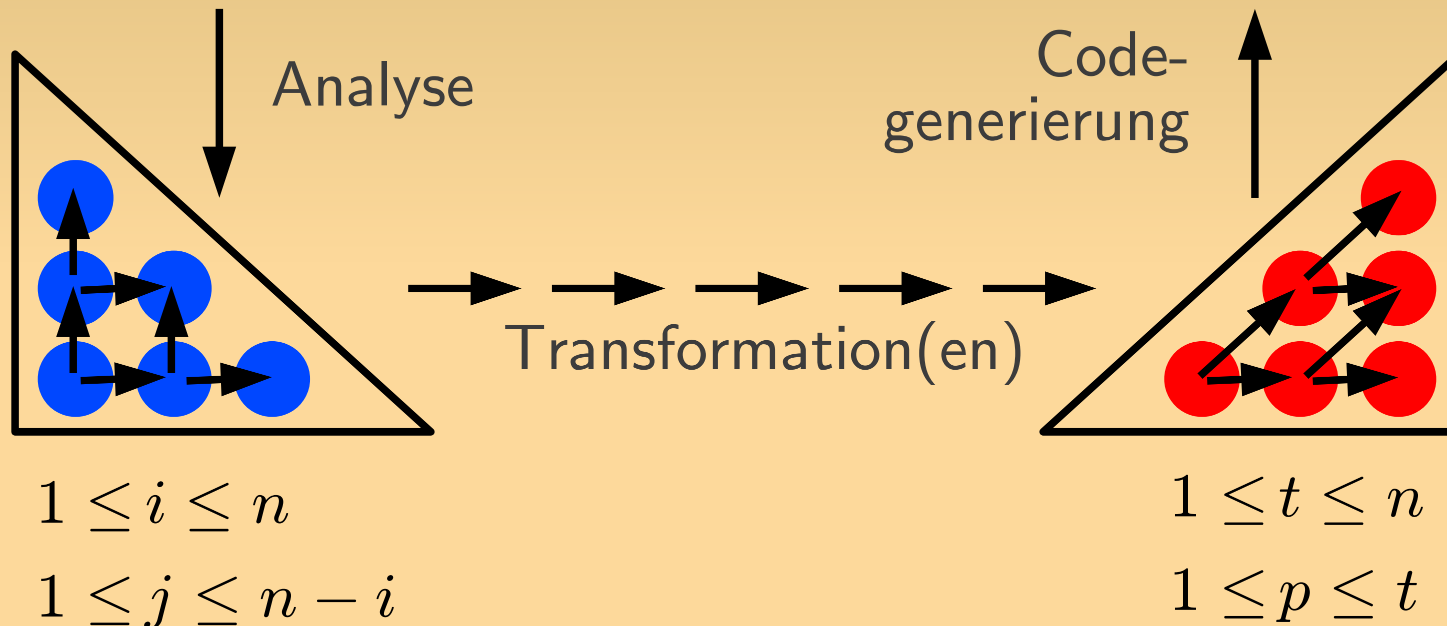
Übersicht

- **Polyedermodell**
  - Einführung
  - Abhängigkeitsanalyse
  - Parallelisierung, Kachelung
- Aktuelle Themen
  - GPUs
  - Erweiterung der Anwendbarkeit des Modells
- Verbindung des Polyedermodells mit
  - GCC, LLVM
  - Just-in-time-Kompilation

Polyedermodell

```
for (i=1; i<=n; i++)
   for (j=1; j<=n-i; j++)
      A[i][j]=A[i-1][j]+A[i][j-1];
```

```
for (t=1; t<=n; t++)
   parfor (p=1; p<=t; p++)
      A[t-p+1][p] = ...;
```



Abhängigkeiten:

$(i,j) \rightarrow (i{+}1,j)$

$(i,j) \rightarrow (i,j{+}1)$

$(t,p) \rightarrow (t{+}1,p)$

$(t,p) \rightarrow (t{+}1,p{+}1)$

Polyedermodell

- Schleifensätze mit Arrayzugriffen in den Rümpfen
- **Linearität**:
  - Schleifengrenzen und Arraysubskripte linear in umgebenden Schleifenvariablen und Parametern
    → Iterationsräume sind (Z-)Polyeder
    → Abhängigkeiten durch affine Relationen beschreibbar
  - solche Schleifensätze heißen SCoPs (static control parts)
- Einschränkungen ermöglichen ”Vollständigkeit”, d.h. *vollautomatische* Anwendung auf beliebige Eingaben
- Codes mit diesen Eigenschaften: wissenschaftliches Rechnen, Bildverarbeitung, etc.
- Parallelisierung: konservativ

Projekt in Passau

- Schleifenparallelisierer LooPo (seit 1994)
  - basiert auf Polyedermodell (mit Erweiterungen)
  - Parallelisierung zur Compile-Zeit
  - Source-to-Source Transformation
  - LooPo Teammitglieder involviert in
    - Graphite: Polyedermodell in GCC
    - Polly: Polyedermodell in LLVM
  - aktuelle Richtung (im Anfangsstadium): Parallelisierung zur Laufzeit

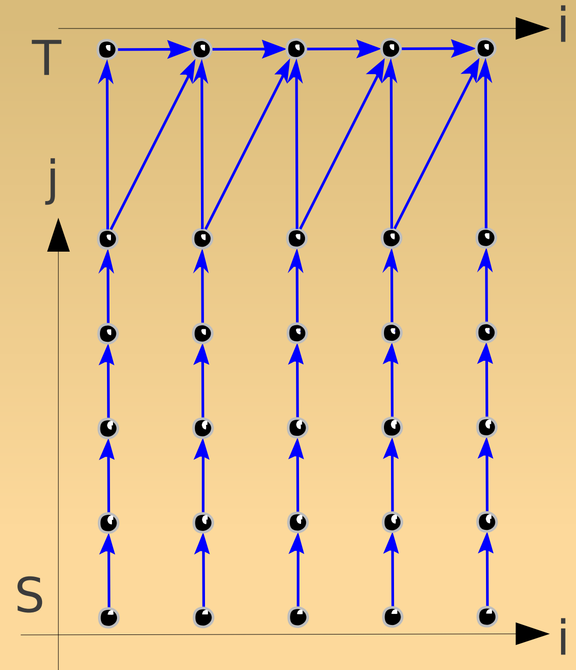
Modellierung

```
   for (i=0; i<n; i++) {
      for (j=0; j<n; j++)
S        C[i] += A[i] * B[j];
T     C[i] += C[i-1];
   }
```

- Iterationsräume (Domains):
  $\{ S[i,j] : 0 \leq i,j < n \}, \{ T[i] : 0 \leq i < n \}$
- Zugriffsfunktionen:
  $\{ S[i,j] \rightarrow A[i] \}, \{ S[i,j] \rightarrow B[j] \},$
  $\{ T[i] \rightarrow C[i] \}, \{ T[i] \rightarrow C[i-1] \}$
- sequenzielle Ausführungsordnung:
  $S[i,j] < S[i,j+1], S[i,j] < S[i+1,j']$
  $T[i] < T[i+1], S[i,j] < T[i]$

Abhängigkeitsanalyse

```
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++)
S           C[i] += A[i] * B[j];
T       C[i] += C[i-1];
    }
```



- Domains, Zugriffsfunktionen und Ausführungsordnung gehen in die Analyse der Abhängigkeiten ein.
- Beispiel: Abh. von C[i] in S nach C[i'-1] in T

$i = i' - 1$	Zugriffskonflikt
$0 \leq i,j < n$	Domain von S
$0 \leq i' < n$	Domain von T
$i \leq i'$	Ausführungsordnung

- Lineare Optimierung liefert unmittelbaren Vorgänger.
- Ergebnis: $\{ S[i,n-1] \rightarrow T[i+1] : 0 \leq i < n-1 \}$

Abhängigkeitsanalyse

- Abhängigkeitsanalyse muss im Ganzzahligen stattfinden, da Schleifen-Iterationen diskrete Einheiten sind.
- Daher arbeiten wir eigentlich nicht mit Polyedern, sondern mit Z-Polyedern.
- Z-Polyeder = Schnitt eines Polyeders mit einem ganzzahligen Gitter
- äquivalent: existenziell quantifizierte Dimensionen
- Beispiel:

```
    for (i=0; i<n; i+=2)
S       ...
```

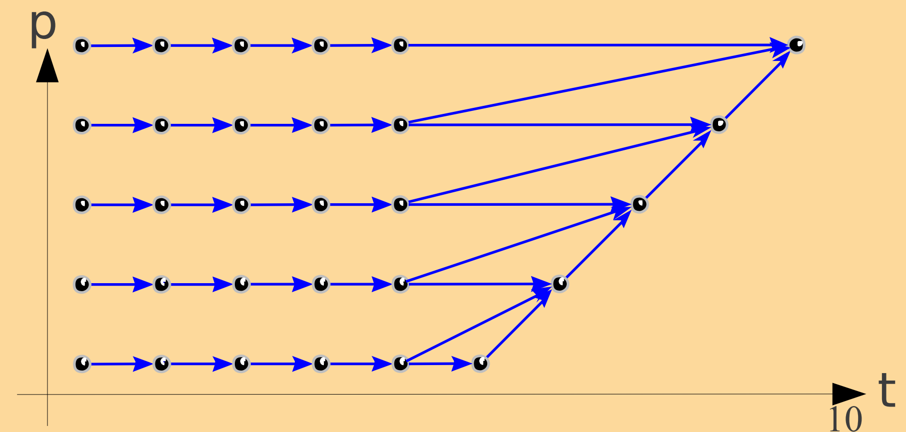
```
    { S[i] : exists z : 0 ≤ i < n and i = 2*z }
```

Modellierung

- In den meisten prototypischen Implementierungen:
  - Domains: Polyeder
  - Arrayzugriffe: Funktionen
  - Bibliothek: Polylib
- Polylib unterstützt theoretisch auch Z-Polyeder; leider schon in der Theorie inkorrekt.
- Heute: **isl** (integer set library von Sven Verdoolaege)
  - Mengen und Relationen mit affin-linearen Grenzen und existenziell quantifizierten Dimensionen.
  - Erste Bibliothek, die korrekt und effizient arbeitet.

Schedule und Placement

- Schedule:
  - ordnet jeder Operation einen Ausführungszeitpunkt zu,
  - muss die Abhängigkeiten respektieren.
- Placement:
  - ordnet jeder Operation einen Ausführungsort zu,
  - ist (zunächst) beliebig.
- Schedule und Placement werden zusammen oft Scattering genannt.
- Im Beispiel:
  { S[i,j] → scattering[j, i] }
  { T[i]   → scattering[i+n, i] }



Repräsentation von SCoPs

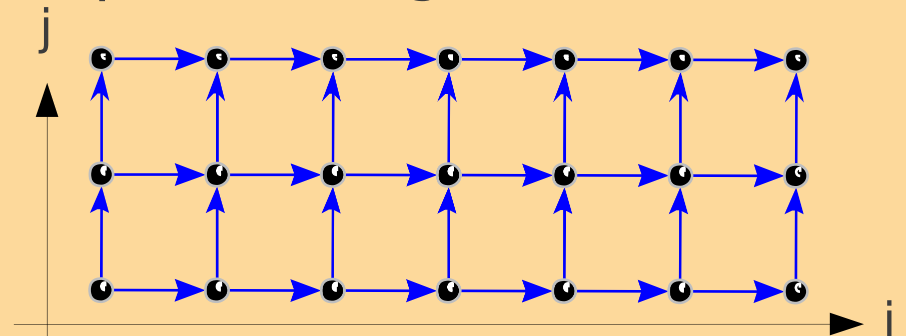
- Scattering kann rein sequenziell sein.
  → sowohl sequenzielles (Eingabe-)Programm als auch paralleles (Ausgabe-)Programm können gleichermaßen dargestellt werden durch Domains, Schedules und Zugriffsrelationen.
-
```
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++)
S           C[i] += A[i] * B[j];
T       C[i] += C[i-1];
    }
```
- Schedule für sequenzielle Ausführungsordnung:
  { S[i,j] → scattering[0, i, 0, j, 0] }
  { T[i]   → scattering[0, i, 1, 0, 0] }

Finden von Parallelität

- Besteht eine Abhängigkeit S[x] → T[y], so muss für die Schedules t_S und t_T von S und T gelten:

  $t_S(x)+1 \leq t_T(y)$

- Die Koeffizienten von t_S etc. (als affin-lineare Funktion in x etc.) können durch lineare Optimierung bestimmt werden.



- Für

```
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
S           A[i][j] = A[i-1][j] + A[i][j-1];
```

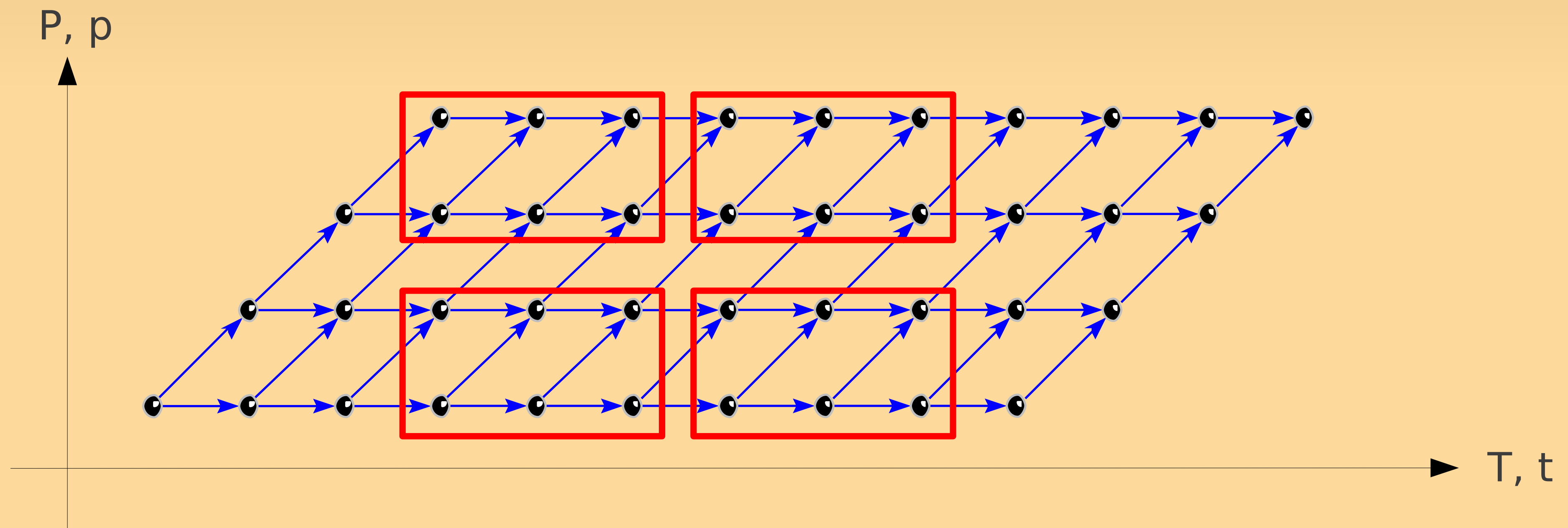
  berechnet man daher „traditionell“:

  { S[i,j] → scattering[**i+j**, j] }

- Für hohe Performanz ist ein weiterer Schnitt nötig.

Tiling (Kachelung)

- Vermeidung häufiger Synchronisation durch Vergröberung der gefundenen Parallelität.
- Beispiel: Kacheln mit Breite b und Höhe h; Koordinaten (Nummern) der Kacheln: (T,P)

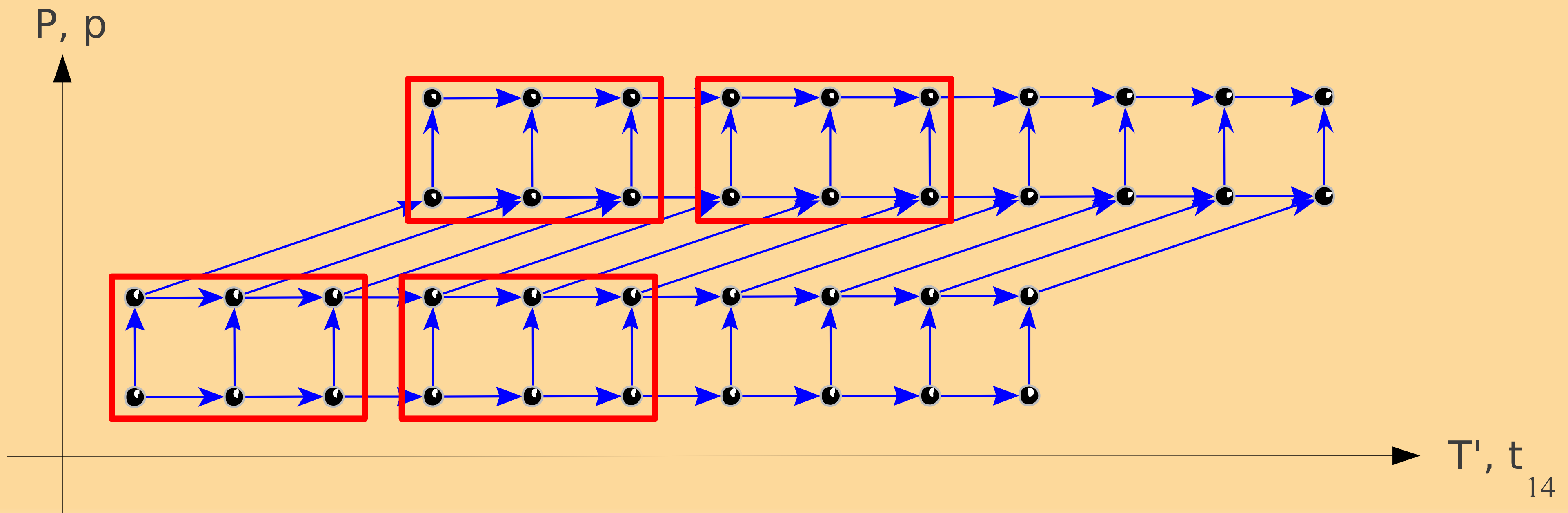


$$0 \leq t - b \cdot T \leq b - 1$$

$$0 \leq p - h \cdot P \leq h - 1$$

Skewing

- Die Parallelität auf Kachel-Ebene erfordert eine Verzögerung von Kacheln mit höherer P Koordinate:
  $T' = T + P$
- Wird jede Kachel auf einen Prozessorkern abgebildet, ist Parallelität innerhalb der Kachel nicht nötig.
- Statt $t_S(x) + 1 \leq t_T(y)$ jetzt $t_S(x) \leq t_T(y)$.



Forward Communication Only

- Suchen eines kommunikationsminimalen t mit $t_S(x) \leq t_T(y)$ per linearer Optimierung möglich.
- Lösungen können als Schedule- und Placementdimensionen verwendet werden. $p_S(x) \leq p_T(y)$ im Placement garantiert auch, dass durch Tiling keine Kommunikationszyklen entstehen.
- Problem: p=0 ist eine triviale Lösung ohne Parallelität.
- Lösungsmöglichkeiten:
  - Statt linearer Optimierung Betrachtung der Generatordarstellung des Systems und Wahl eines Rays (Griebl 2003: LooPo).
  - Einschränkung auf „nicht alle Koeffizienten von p sind 0“ durch Heuristik, z.B. Summe der Koeffizienten > 0 (Bondhugula 2007: PLuTo).

Übersicht

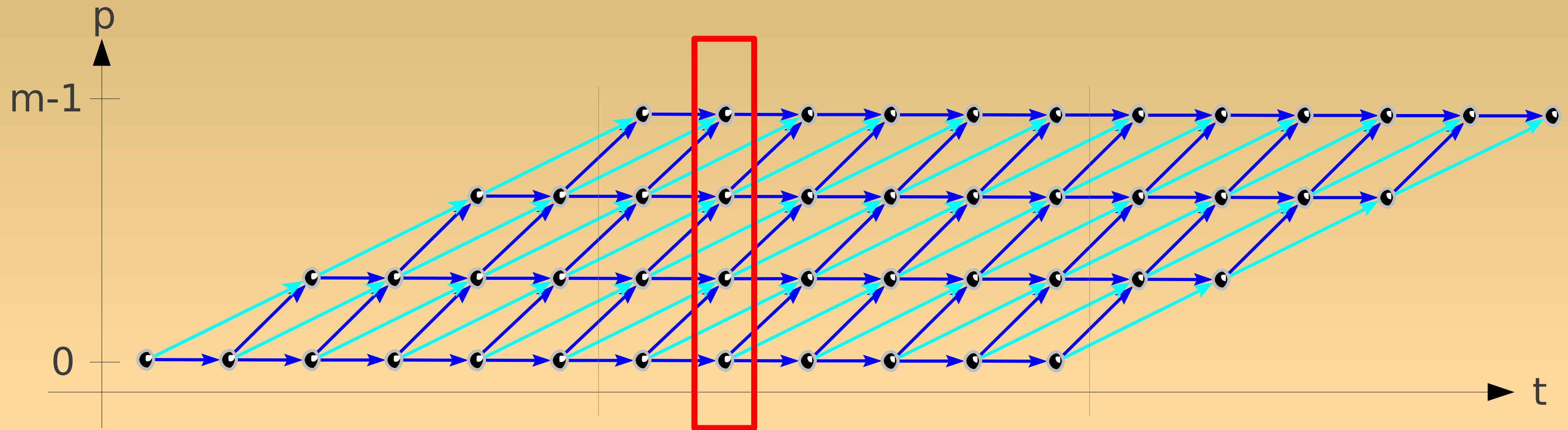
- Polyedermodell
  - Einführung
  - Abhängigkeitsanalyse
  - Parallelisierung, Kachelung
- **Aktuelle Themen**
  - GPUs
  - Erweiterung der Anwendbarkeit des Modells
- Verbindung des Polyedermodells mit
  - GCC, LLVM
  - Just-in-time-Kompilation

Parallelisierung für GPU

- Massive Parallelität erforderlich
  → Parallelität auch innerhalb der Tiles
- Benutzung von Scratchpad und/oder Textureinheiten für hohe Performanz erforderlich
- Polyedermodell ermöglicht Nummerierung der benutzten Elemente (→ Ehrhart-Theorie)
- Beispiel:
  1-dimensional successive over-relaxation (1-d SOR)

```
for (k=0; k<m; k++)
    for (i=1; i<n-1; i++)
        A[i] = (A[i-1] + A[i+1])/2;
```

Beispiel: 1-d SOR auf GPU




```
for (int t=0; t<=2*m-3; t++) { ... }

for (int t=2*m-2; t<=n-3; t++) {
    parfor (int p=0; p<m; p++) {
        int i = t - 2*p + 1;
        A[i] = (A[i-1] + A[i+1]) * 0.5;
    }
}

for (int t=n-2; t<=n+2*m-4; t++) { ... }
```

Iteration t benutzt die Elemente

`A[t-2m+2], ..., A[t+2]`

A[i] wird im Scratchpad an Position $\rho(i,t)$ gehalten mit

$$\rho(i,t) = i - (t - 2m + 2)$$

Lokalisierter (Pseudo-)Code

```
__scratchpad__ float L[];

...   // lade L mit Werten aus A für erste Iteration

for (int t=2*m-2; t<=n-3; t++) {
    L[2*m] = A[t+2];                  // 2m = ρ(t+2,t)

    parfor (p=0; p<m; p++)
        L[2*m-2*p-1] = (L[2*m-2*p-2] + L[2*m-2*p]) * 0.5;

    A[t-2*m+2] = L[0];              //  0 = ρ(t-2m+2,t)

    syncparfor (x=1; x<=2*m; x++)
        L[x-1] = L[x];
}

...   // Schreibe Werte aus L zurück in A
```

- Ausdruck ρ wird für kompliziertere Codes recht groß. Offenes Problem: Welches „einfache“ ρ ist fast optimal bzgl. Platzverbrauch im Scratchpad?

Erweiterung des Modells

- Modell nur anwendbar für Codes, die die Einschränkungen erfüllen.
- Erweiterungsmöglichkeiten:
  - Überapproximation von Domains und Zugriffsrelationen, z.B.

    ```
    for (i=0; i*i<n; i++)
        ... A[i*i] ...
    ```

    Domain: $\{ S[i] : 0 \leq i < n \}$
    Zugriff: $\{ S[i] \rightarrow A[x] : 0 \leq x \}$

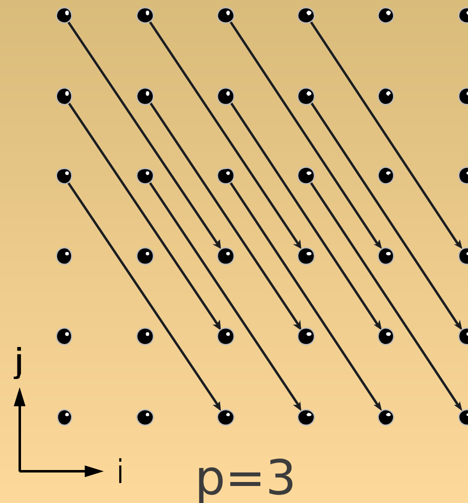
    → im Prinzip jede nicht-rekursive Funktion modellierbar (Benabderrahmane et al. 2010)
  - Mathematische Methoden jenseits der linearen Optimierung → schwierig

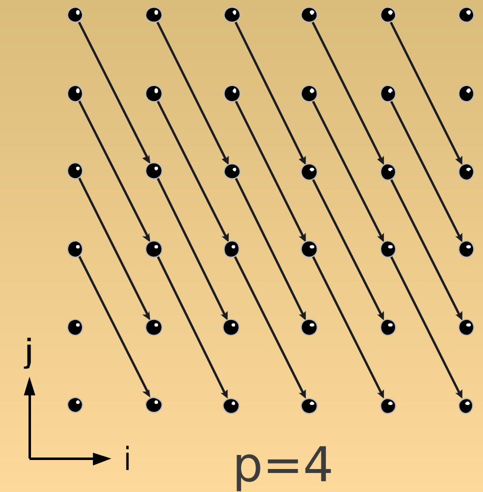
Nichtlinearitäten

- Abhängigkeitsanalyse
  - Schwierig, da ganzzahlige Lösungen erforderlich
  - Algorithmus für genau einen Parameter im Produkt mit Variablen (Größlinger, Schuster 2008)
- Codegenerierung
  - Für beliebige Polynome als Grenzen möglich (Größlinger 2009)
  - Für einige Spezialfälle bei parametrischer Kachelung existieren spezielle Lösungen (Hartono et al. 2009).

Abhängigkeitsanalyse mit einem nichtlinearen Parameter

```
for (i=0; i<=m; i++)
    for (j=0; j<=m; j++)
        ... A[p*i+2*j] ...
```





Welche Iterationen (i,j) greifen auf dasselbe Arrayelement zu?

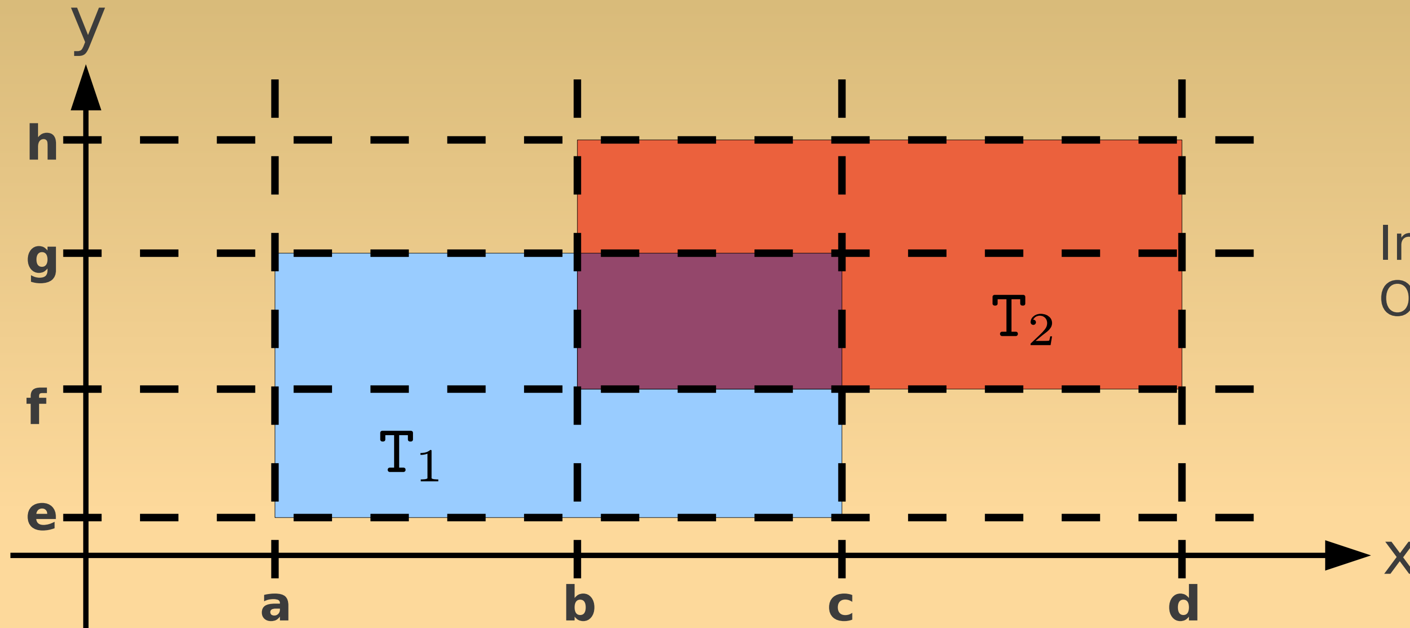
Ergebnis:

$$(i,j) \to (i+1, j-\tfrac{p}{2}) \quad \text{if} \begin{cases} p \equiv_2 0, m \geq 1, -2m \leq p \leq 2m, 0 \leq i \leq m-1, \\ \max(0, \tfrac{p}{2}) \leq j \leq \min(m, m+\tfrac{p}{2}) \end{cases}$$

$$(i,j) \to (i+2, j-p) \quad \text{if} \begin{cases} p \equiv_2 1, m \geq 2, -m \leq p \leq m, 0 \leq i \leq m-2, \\ \max(0, p) \leq j \leq \min(m, m+p) \end{cases}$$

Probleme:
- Große Konstanten führen leicht zu großen Fallunterscheidungen.
- Keine Hoffnung (aus mathematischen Gründen) für mehr als 1 Parameter.

Codegenerierung



In *lexikografischer* Ordnung aufzählen.

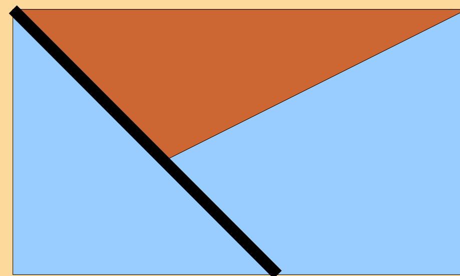
```
for (x=a; x≤d; x++) {
  for (y=e; y≤h; y++) {
    if (a≤x≤c ∧ e≤y≤g) T1;
    if (b≤x≤d ∧ f≤y≤h) T2;
  }
}
```

Aus Effizienzgründen: Keine Fallunterscheidungen in Schleifenrümpfen!

```
for (x=a; x≤b-1; x++)
  for (y=e; y≤g; y++) T1;
for (x=b; x≤c; x++) {
  for (y=e; y≤f-1; y++) T1;
  for (y=f; y≤g; y++) { T1; T2; }
  for (y=g+1; y≤h; y++) T2;
}
for (x=c+1; x≤d; x++)
  for (y=f; y≤h; y++) T2;
```

Polyedrische Codegenerierung

- Partitionierungen des Iterationsraumes und seiner Projektionen werden berechnet durch
  - Schnitte und Differenzen von Polyedern,
  - Projektionen von Polyedern.
- Invariante: Schnitte, Differenzen und Projektionen von Polyedern liefert wieder Vereinigungen von Polyedern → stets konvexe Mengen



- Partitionen (Polyeder) können stets dimensionsweise geordnet werden; die Wahl der Partitionierung beeinflusst nur die Effizienz des generierten Codes.

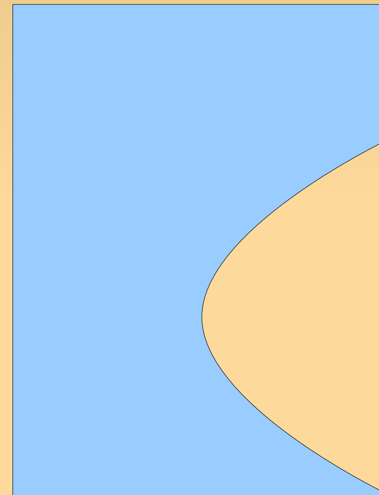
Schleifen für semialgebraische Iterationsräume

- Semialgebraische Menge =
  definiert durch polynomiale (Un-)gleichungen
- Können nicht-konvex sein

$$1 \leq x \leq 7$$

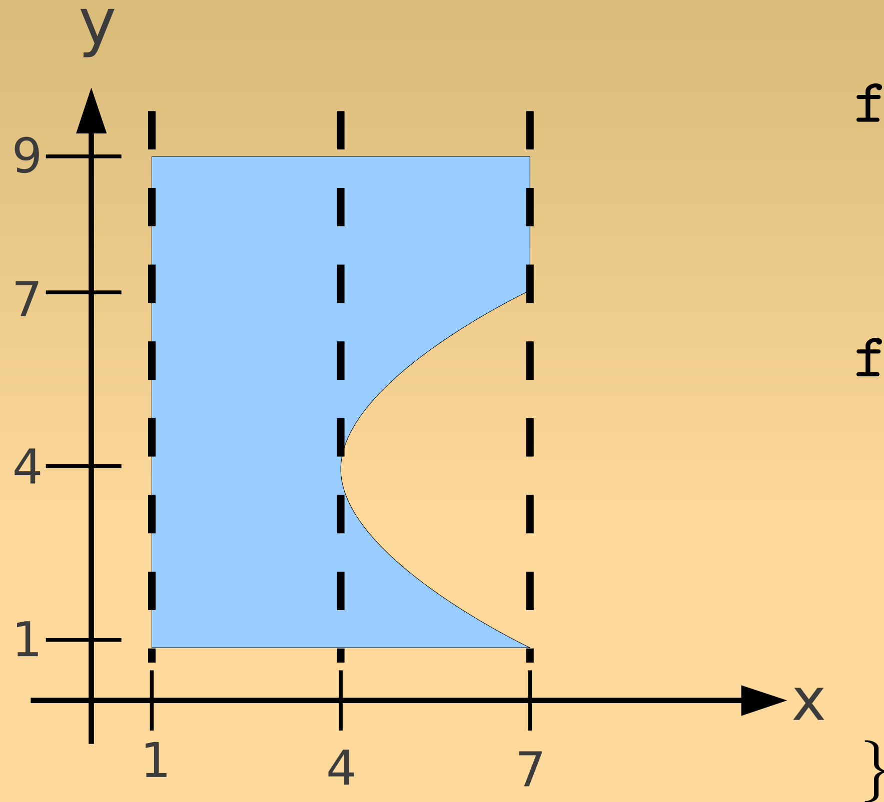
$$1 \leq y \leq 9$$

$$0 \leq (y-4)^2 + 12 - 3x$$



- Konvexität ist nicht erforderlich für Codegenerierung.
- Die Analogie zu dimensionsweise geordneten konvexen Mengen ist ***zylindrische Dekomposition***.

Ein semialgebraisches Beispiel




```
for (x=1; x≤4; x++)
   for (y=1; y≤9; y++)
      T(x,y);
for (x=4+1; x≤7; x++) {
   for (y=1; y≤⌊4−√(3x−12)⌋; y++)
      T(x,y);
   for (y=⌈4+√(3x−12)⌉; y≤9; y++)
      T(x,y);
}
```

$$1 \leq x \leq 7$$

$$1 \leq y \leq 9$$

$$0 \leq (y-4)^2 + 12 - 3x$$

Übersicht

- Polyedermodell
  - Einführung
  - Abhängigkeitsanalyse
  - Parallelisierung, Kachelung
- Aktuelle Themen
  - GPUs
  - Erweiterung der Anwendbarkeit des Modells
- **Verbindung des Polyedermodells mit**
  - GCC, LLVM
  - Just-in-time-Kompilation

- Bisher:
  - Source-to-Source Transformation von Fortran/C nach Fortran/C/CUDA
  - Optimierung zur Übersetzungszeit
  - Eingabe in eingeschränkter Sprache, z.B. kein Pointer Aliasing etc.
- Neue Ziele:
  - Analyse muss nicht auf Quelltext-Ebene erfolgen → Analyse auf Zwischencode zur Integration des Modells in einen Produktionscompiler
  - Optimierung zur Laufzeit
  - Zielcode muss nicht C sein → Generierung von Code für domänenspezifische Compiler

GCC und LLVM

- Graphite (GCC) und Polly (LLVM):
  - Erkennung von SCoPs im Zwischencode
  - Export von SCoPs in OpenScop und JSCoP Formaten
  - Import der (durch externe Tools) transformierten SCoPs
  - Generierung von Schleifen im Zwischencode aus den transformierten SCoPs (durch CLooG)
  - Erzeugung von OpenMP Code (in Arbeit)
  - Erzeugung von Vektorinstruktionen (in Arbeit)
  - LLVM: GPU-Backends von AMD und NVIDIA (in Arbeit)
- LooPo Teammitglieder in beiden Projekten aktiv

LLVM/Polly

- Mit Hilfe der Analysen in LLVM Erkennung von
  - Schleifen und
  - affin-lineare Ausdrücke für Schleifengrenzen und Arraysubskripte.
- Probleme:
  - Erkennung mehrdimensionaler Arrays: ist A[n*i+j] im Zwischencode eigentlich A[i][j]? (unproblematisch, wenn n eine bekannte Konstante ist)
  - SCoP-Erkennung funktioniert nur, nachdem bestimmte Umformungen am Zwischencode vorgenommen wurden. Unklar: Welche Optimierungsflags führen reproduzierbar zur Erkennung von SCoPs?

Polly Beispiel

- C Code

```
for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
        C[i] += A[i] * B[j];
    C[i] += C[i-1];
}
```

- Zwischencode:

```
...
for.body6:                                ; preds = %for.body6, %for.cond2.preheader
  %indvar = phi i64 [ 0, %for.cond2.preheader ], [ %indvar.next, %for.body6 ]
  %arrayidx.moved.to.for.body6 = getelementptr float* %A, i64 %indvar3
  %arrayidx29.moved.to.for.body6 = getelementptr float* %C, i64 %indvar3
  %arrayidx13 = getelementptr float* %B, i64 %indvar
  %tmp9 = load float* %arrayidx.moved.to.for.body6, align 4
  %tmp14 = load float* %arrayidx13, align 4
  %mul = fmul float %tmp9, %tmp14
  %tmp19 = load float* %arrayidx29.moved.to.for.body6, align 4
  %add = fadd float %tmp19, %mul
  store float %add, float* %arrayidx29.moved.to.for.body6, align 4
  %indvar.next = add i64 %indvar, 1
  %exitcond = icmp eq i64 %indvar.next, 100
  br i1 %exitcond, label %for.end, label %for.body6
...
```

JSCoP Ausgabe

- Erkannter SCoP (in JSON-Repräsentation)

```
{
    "name": "for.cond2.preheader => for.end35",
    "context": "{ [] }",
    "statements": [{
        "name": "Stmt_for_body6",
        "domain": "{ Stmt_for_body6[i0, i1] : i0 >= 0 and i0 <= 99 and i1 >= 0 and i1 <= 99 }",
        "schedule": "{ Stmt_for_body6[i0, i1] -> scattering[0, i0, 0, i1, 0] }",
        "accesses": [{
            "kind": "read",
            "relation": "{ Stmt_for_body6[i0, i1] -> MemRef_A[i0] }"
        }, ...]
    },
    {
        "name": "Stmt_for_end",
        "domain": "{ Stmt_for_end[i0] : i0 >= 0 and i0 <= 99 }",
        "schedule": "{ Stmt_for_end[i0] -> scattering[0, i0, 1, 0, 0] }",
        "accesses": [{
            "kind": "read",
            "relation": "{ Stmt_for_end[i0] -> MemRef_C[-1 + i0] }"
        }, ...]
    }]
}
```

Parallelisierung zur Laufzeit

- Vorteile:
  - Vereinfachung der Analyse
    - nichtparametrische Abhängigkeitsanalyse
  - Größere Anwendbarkeit der Analyse
    - weniger Nichtlinearitäten
    - Aliasing der Pointer bekannt
    - aufgerufene Methode bei dynamischer Bindung bekannt
  - Vereinfachung der Transformationen/Codegenerierung
    - Kachelgrößen bekannt
    - Maschinenparameter bekannt
    - geringere Zielcodegröße
- Nachteil: Optimierung muss schnell genug sein.

Aktuelle Arbeiten

- Evaluierung des Potenzials, d.h. wie viel Code ist dynamischer polyedrischer Optimierung zugänglich? (Andreas Simbürger, Passau)
- Integration eines Callbacks zur dynamischen (Re-)Kompilation mit Polly in den Just-in-Time-Compiler von LLVM (Bachelorarbeit in Passau)
- Anbindung von LooPo an die Polly/JSCoP Infrastruktur
- Entwicklung von „subpolyedrischen“ Algorithmen zur schnelle Approximation der exakten Lösung (z.B. Upadrasta, Cohen 2011)

Zusammenfassung: Durchbruch?

- Polyedermodell auf dem Weg in „echte“ Compiler
  - korrekte Modellierung (Sprachsemantik, math. Bibliotheken)
  - Tests mit LLVM Test Suite statt weniger Beispiele
- Behandlung aktueller Architekturen: GPUs, Multicores
  - Generierung domänenspezifischen Codes für spezielle Compiler statt C oder Zwischencode?
- Verbindung mit JIT
  - Komplexitätsreduktion
  - Erweiterung der Anwendbarkeit
- Das Polyedermodell ist reif für die Praxis.