Estimation of Parallel Complexity with Rewriting Techniques

Christophe Alias*, Carsten Fuhs†, Laure Gonnord‡

* INRIA & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France, christophe.alias@ens-lyon.fr
† Birkbeck, University of London, United Kingdom, carsten@dcs.bbk.ac.uk
‡ University of Lyon & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France, laure.gonnord@ens-lyon.fr

Languages, Compilation and Semantics, LIP Seminar
November, 3rd 2016
1. Introduction

2. Parallel complexity of regular programs
   Parallelization $\leftrightarrow$ Termination

3. Parallel complexity of recursive programs
   Termination $\leftrightarrow$ Parallelization

4. Conclusions
Parallel Systems

⇝ Many models: KPN (and variants), SDF (and variants), etc

Correctness
  - Determinism
  - Termination

Efficiency
  - Latency
  - Bandwidth
... generated by parallelizing compilers!

From the source code:

- Target latency? $\geq$ termination
- Target bandwidth?
**Trace** ($\Omega_{\mathcal{I}}, \prec_{\text{seq}}$): sequence of operations executed by the program on the input $\mathcal{I}$.

*Undecidable* in general!

- Restrict program model $\rightsquigarrow$ Polyhedral model
- Over-approximate $\Omega_{\mathcal{I}} \rightsquigarrow$ Abstract interpretation

**Regular Program** $\Omega_{\mathcal{I}}$ does not depend on $\mathcal{I}$.
$\rightsquigarrow$ Polyhedral model
for $i := 0$ to $2*N$
  $\ell_1$: $c[i] := 0$;
for $i := 0$ to $N$
  for $j := 0$ to $N$
    $\ell_2$: $c[i+j] := c[i+j] + a[i]*b[j]$;

- for loop with arrays + affine constraints
for i := 0 to 2*N
    ℓ₁: c[i] := 0;
for i := 0 to N
    for j := 0 to N
        ℓ₂: c[i+j] := c[i+j] + a[i]*b[j];

- **for loop with arrays** + affine constraints
- \( \Omega_I \) can be encoded with integer polyhedra:
  \[ \langle \ell_1, i \rangle : i \in [0, 2N] \]
  \[ \langle \ell_2, i, j \rangle : (i, j) \in [0, N]^2 \]
- Static analysis with ILP: dependences, scheduling, allocation
Each node (subtree) of $t$ is an operation of $\Omega_t$
Not exactly regular, but $\Omega_t$ is decidable!
**Data Dependence** relate communicating/conflicting operations, 
\[ \rightarrow \subseteq \Omega \times \Omega \]
\[ \leadsto \text{usually split into flow-, anti-, output-dependences} \]

**Schedule** Assign dates to operations, \( \theta : \Omega \rightarrow (D, \prec) \)

**Induced order** \( \prec_\theta = \{(s, t), \, \theta(s) \prec \theta(t)\} \)

**Correctness** \( s \rightarrow t \implies \theta(s) \prec \theta(t) \)

Hence: \( \rightarrow \subseteq \prec_\theta \)

---

Ordres séquentiels

...
// Compute $y = Ax$
for $i := 0$ to $N-1$
\[ \ell_1: y[i] := 0; \]
for $j := 0$ to $N-1$
\[ \ell_2: y[i] := y[i] + a[i][j] \times x[j]; \]
\[ \lambda = \mathcal{O}(N) \]

Latency $\lambda$ longest chain of $\to$
Degree of sequentiality smallest $s \in \mathbb{N}$ s.t. $\lambda = \mathcal{O}(N^s)$
Counterpart in the Tree Model?
Transition system

- $\Sigma$ set of states
- $\Sigma_0 \subseteq \Sigma$ initial states
- $\rightarrow \subseteq \Sigma \times \Sigma$ transition relation

Termination  Exhibit a ranking function $\rho : \Sigma \rightarrow (\mathcal{D}, \prec)$ s.t.

$$s \rightarrow t \Rightarrow \rho(t) \prec \rho(s)$$

WCET

$$\lambda = \max\{n, \exists \sigma_0 \in \Sigma_0 : \sigma_0 \rightarrow^n \sigma\}$$

Upper bound:

$$\lambda \leq \#\{\rho(\sigma), \exists \sigma_0 \in \Sigma_0 : \sigma_0 \rightarrow^* \sigma\}$$
\textbf{while}(p \neq q)\
\hspace{1em}\textbf{if}(p < q)\
\hspace{2em}p := p - q;\
\hspace{1em}\textbf{else}\
\hspace{2em}q := q - p;\

Each state of $\Sigma$ is a pair $\langle \ell, \vec{i} \rangle$

$\rho$ is affine per control point:

$\rho : \langle \ell, \vec{i} \rangle \mapsto Ai + b, \quad \Sigma \to (\mathbb{Z}^p, \ll)$
<table>
<thead>
<tr>
<th>Parallelization</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependence</td>
<td>Transition system</td>
</tr>
<tr>
<td>Schedule Latency</td>
<td>Ranking function</td>
</tr>
<tr>
<td></td>
<td>WCET</td>
</tr>
</tbody>
</table>
Outline

1. Introduction

2. Parallel complexity of regular programs
   Parallelization $\leftrightarrow$ Termination

3. Parallel complexity of recursive programs
   Termination $\leftrightarrow$ Parallelization

4. Conclusions
for $i := 0$ to $2*N$
  $\ell_1$: $c[i] := 0$;
for $i := 0$ to $N$
  for $j := 0$ to $N$
    $\ell_2$: $c[i+j] := c[i+j] + a[i]*b[j]$;

Idea: Given a consumer, find the last producer $\leadsto$ ILP.
Dependence Analysis

for i := 0 to 2*N
  ℓ1: c[i] := 0;
for i := 0 to N
  for j := 0 to N
    ℓ2: c[i+j] := c[i+j] + a[i]*b[j];

Idea: Given a consumer, find the last producer $\rightsquigarrow$ ILP.

$\rightarrow_N$ is an affine relation:
\[
\langle \ell_2, i-1, j+1 \rangle \rightarrow_N \langle \ell_2, i, j \rangle : i > 0 \land j < N \\
\langle \ell_1, i \rangle \rightarrow_N \langle \ell_2, 0, i \rangle : 0 \leq i \leq N \\
\langle \ell_1, i \rangle \rightarrow_N \langle \ell_2, i-N, N \rangle : N < i \leq 2N
\]
$N < i \leq 2N : (i) \rightarrow (i - N, N)$

$0 \leq i \leq N : (i) \rightarrow (0, i)$

$0 \leq i, j \leq N, i > 0, j < N : (i - 1, j + 1) \rightarrow (i, j)$

$\rightarrow_N$ is an affine relation:

$\langle \ell_2, i - 1, j + 1 \rangle \rightarrow_N \langle \ell_2, i, j \rangle : i > 0 \land j < N$

$\langle \ell_1, i \rangle \rightarrow_N \langle \ell_2, 0, i \rangle : 0 \leq i \leq N$

$\langle \ell_1, i \rangle \rightarrow_N \langle \ell_2, i - N, N \rangle : N < i \leq 2N$

... finitely represented as a graph (PRDG)
Scheduling

Timestamps are vectors of \((\mathbb{N}^{d\ell}, \ll)\).

**Affine schedule:** \(\theta_{\ell} : \vec{x} \mapsto A_{\ell} \vec{x} + B_{\ell} \vec{N} + \vec{c}_{\ell} \quad D_{\ell} \rightarrow (\mathbb{N}^{d\ell}, \ll)\)

Can be computed with **ILP**
Scheduling

 Timestamps are vectors of \((\mathbb{N}^{d_\ell}, \ll)\).

 **Affine schedule:** \(\theta_{\ell} : \vec{x} \mapsto A_{\ell} \vec{x} + B_{\ell} \vec{N} + \vec{c}_\ell \quad D_{\ell} \to (\mathbb{N}^{d_\ell}, \ll)\)

 Can be computed with ILP

 **Bonus:** reverse the order: termination algorithm! \([\text{Rank}, 2010]\)
Outline

1 Introduction

2 Parallel complexity of regular programs
   Parallelization $\leftrightarrow$ Termination

3 Parallel complexity of recursive programs
   Termination $\leftrightarrow$ Parallelization

4 Conclusions
public int treeMax() {
    int leftMax = Integer.MIN_VALUE;
    int rightMax = Integer.MIN_VALUE;
    if (this.left != null)
        leftMax = this.left.treeMax();
    if (this.right != null)
        rightMax = this.right.treeMax();
    return Math.max(this.val,
                     Math.max(leftMax, rightMax));
}

Each node (subtree) of $t$ is an operation of $\Omega_t$.

$\rightarrow$ can be encoded as a term rewrite system (TRS):

$$
\text{dep(Tree(val, left, right))} \rightarrow \text{dep(left)}
$$

$$
\text{dep(Tree(val, left, right))} \rightarrow \text{dep(right)}
$$

How to schedule (check the termination of) a TRS?

$\Rightarrow$ With monotone interpretations! [AProVE, KoAT]
Given a TRS $\rightarrow$ over a term algebra $\mathcal{T}(\Sigma)$:

- Assign each symbol $a/n \in \Sigma$ with $[a] : \mathbb{R}^n \rightarrow \mathbb{R}$.
- The monotone interpretation of a term is:

$$[f(t_1, \ldots, t_n)] = [f](t_1, \ldots, t_n)$$

- Correctness: $t \rightarrow u \Rightarrow [t] > [u]$.
- Polynomial interpretations can be computed by KoAT.
Putting it all together

```java
public int treeMax() {
    int leftMax = Integer.MIN_VALUE;
    int rightMax = Integer.MIN_VALUE;
    if (this.left != null)
        leftMax = this.left.treeMax();
    if (this.right != null)
        rightMax = this.right.treeMax();
    return Math.max(this.val,
                     Math.max(leftMax, rightMax));
}
```

<table>
<thead>
<tr>
<th>Monotone interpretation</th>
<th>Parallel complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{dep}] (x_1)) = x_1</td>
<td>(\lambda = \mathcal{O}(\text{height}(t)))</td>
</tr>
<tr>
<td>([\text{Tree}] (x_1, x_2, x_3)) = (x_2 + x_3 + 1)</td>
<td>(\lambda = \mathcal{O}(\text{height}(t)))</td>
</tr>
</tbody>
</table>

Monotone interpretation

1. \([\text{dep}] (x_1)\) = \(x_1\)
2. \([\text{Tree}] (x_1, x_2, x_3)\) = \(x_2 + x_3 + 1\)
What happens on polyhedral programs?

```plaintext
for (i=0; i<=N; i++)
  for (j=0; j<=N; j++)
    //Block S
    {
      m1[i][j] = Integer.MIN_VALUE;
      for (k=1; k<=i; k++)
        m1[i][j] = max(m1[i][j], H[i-k][j] + W[k]);
      m2[i][j] = Integer.MIN_VALUE;
      for (k=1; k<=j; k++)
        m2[i][j] = max(m2[i][j], H[i][j-k] + W[k]);
      H[i][j] = max(0, H(i-1, j-1) + s(a[i], b[i]),
                    m1[i][j], m2[i][j]);
    }
```

dep(i, j) → dep(i – 1, j – 1): 0 ≤ i ≤ n, 0 ≤ j ≤ n

dep(i, j) → dep(i – k, j): 0 ≤ i ≤ n, 0 ≤ j ≤ n, 1 ≤ k ≤ i

dep(i, j) → dep(i, j – ℓ): 0 ≤ i ≤ n, 0 ≤ j ≤ n, 1 ≤ ℓ ≤ j

Result: [dep](x₁, x₂) = x₁ + x₂  λ ≤ 2n

Same as in the polyhedral model!
Conclusions

Position:
- Two successful experiences of cross fertilization
  \emph{Parallelization} $\leftrightarrow$ \emph{Termination}.
- Monotonic interpretations can benefit to automatic parallelization.
- Natural extension of affine scheduling to recursive programs

Locks:
- How to define/find the best schedule?
- How to count the steps?
- Steps towards a parallelizing compiler:
  - Computation partitioning?
  - Generation of the parallel code given a schedule?
Thanks!