The Meta Transformation Tool for Skeleton-Based Languages

Marco Aldinucci

Computer Science Dept.
University of Pisa
Pisa – Italy
Skeletons

- Structured programming models
- Skeletons
  -> known, reusable, parallelism exploitation patterns
  -> Think to the analogy with seq. prog. (while do, for ...)
- Programmers concentrate on qualitative aspects of parallelism
- Tools deal with implementation details and quantitative aspects
  -> load-balancing, parallelism degree, messages size, etc.
- Specification of the software architecture
Why skeletons?

- Parallel programming is difficult and error prone
  → Skeletons have a clear, functional and parallel semantic

- Performance portability
  → Only performance can justify HPC high costs
  → Performance heavily depends on the “matching” of the program with the architecture

- Several MPI implementation of the same algorithm

<table>
<thead>
<tr>
<th></th>
<th>seq</th>
<th>farm</th>
<th>farm + pipe</th>
<th>pipe + 2*farm</th>
<th>pipe + farm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_s$  (sec)</td>
<td>6.03</td>
<td>0.39</td>
<td>1.30</td>
<td>0.72</td>
<td>4.99</td>
</tr>
<tr>
<td>$T_c$  (sec)</td>
<td>1207.76</td>
<td>84.50</td>
<td>286.62</td>
<td>151.67</td>
<td>1004.69</td>
</tr>
<tr>
<td>#PE</td>
<td>1</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>$\varepsilon$(%)</td>
<td>75.52</td>
<td>23.08</td>
<td>41.93</td>
<td>6.04</td>
<td></td>
</tr>
</tbody>
</table>

(Aldinucci & Danelutto. IASTED PDCS’99, MIT, Boston, USA)
Common skeletons

- Task parallel skeletons
  - parallelism in the computation of stream tasks
  - E.g. pipeline, task farm, etc.

- Data parallel skeletons
  - parallelism in the computation of a single task
  - E.g. map (independent forall), reduce, D&C, etc.

- Sequential skeletons
  - the degenerate case: no parallelism at all
  - sometime used to wrap functions written in a guest language
The methodology

- Ingredients
  - A skeleton-based language
  - A cost calculus to foresee program performance
  - A set of semantic-preserving rewriting rules

- Methodology
  1. Write an initial specification/program
  2. Evaluate its performance
  3. Transform it (until the performance is satisfactory)

- Wish list
  - A compiler for the language
  - A tool to transform (optimize) programs
Outline

- Framework & Motivation
  → Outline
  - The Meta transformation tool
    - A short introduction
    - Dealing with languages and rules
    - How it work
    - The architecture and the implementation
    - Running the tool
  - Conclusions
The Meta transformation tool: What does it do?

Given:

1. A Target Language (TL)

2. A set of (sound) rewriting rules (L → R) for TL

3. A program written in TL

the Meta tool:

- locates applicable transformations
- provides performance estimates
- (possibly) transforms the program accordingly with the (user) chosen rule

**Meta basically implements a (meta) term-rewriting system**
The Meta transformation tool: properties

- Interactive
- Graphical
- Fast
- Language independent
- Rule independent
- Manages mixed data and task parallel languages

Meta can be instantiated with a broad class of TL (three-tier languages) and sets of rules for TL.
Skeleton-Based (up to) three-tier languages

```
      Program
      /---\            /---\
     TP skel 1        TP skel 2         TP skel 3
     /         \      /         \      /         \
    TP skel 2       TP skel 3
    /     \        /     \        /     \
   DP skel A       DP skel B       DP skel C
   /   \           /   \           /   \
  seq fun 1 seq fun 2 seq fun 3
```

```
      Program
      /---\            /---\
     TP skel 1        TP skel 2         TP skel 3
     /         \      /         \      /         \
    TP skel 2       TP skel 3
    /     \        /     \        /     \
   DP skel A       DP skel C       DP skel B
   /   \           /   \           /   \
  seq fun 1 seq fun 2
```

**Lower levels skeletons cannot call upper level ones**

“The exploitation of task parallelism is (often) orthogonal with respect to the exploitation of data parallelism”
Three-tier languages: testbeds

**FAN: Functional Abstract Notation**

- Data parallel skeleton language
- FAN cost calculus

**Skel-BSP: Skeletons on top of the H-BSP**

- Task and data parallel skeleton language
- Implemented on top of (hierarchical) BSP
- BSP-like cost calculus

*Notice I don’t present here neither new languages nor new rules*

*Indeed, they have been presented and validated elsewhere*
Three-tier languages: example Skel-BSP

\[ TL_{\text{prog}} ::= TP \mid DP \]
\[ TP ::= \text{farm} (TP) \mid \text{pipe } \{TP\text{list}\} \mid DP \]
\[ DP ::= \text{map } Seq \mid \text{scanL } Seq \mid \text{reduce } Seq \mid Seq \mid \text{comp } \{\text{out Var, in Varlist}\} \{DP\text{list}\} \ldots \]
\[ Seq ::= \langle \text{a sequential C function} \rangle \]
\[ \ldots \]

\[ \text{comp.name} (\text{out outvar, in invars})\{
  \text{outvar}_1 = \text{dp.1 Op}_1 \text{ invars}_1 \\
  \vdots \\
  \text{outvar}_n = \text{dp.n Op}_n \text{ invars}_n\}\]

\text{comp} definition follows the single-assignment rule: there is at most one equation defining each variable. Skeletons into \text{comp} are executed in sequence on a single set of PEs.
Rewriting rules

A rule is a pair $L \rightarrow R$ where

- $L$ and $R$ are fragments of TL programs with variables $\nu_0, \nu_1, \ldots$
- $\nu_0, \nu_1, \ldots$ act as placeholders for any piece of program
- Every variable occurring in $R$ must occur also in $L$
- $L$ is not a variable
Rewriting rules: examples (1)

\[ T_{Sk} \quad \rightarrow \quad \text{farm (T}_{Sk} \text{)} \]

A \textbf{farm} replicates \textbf{T}_{Sk} without changing the function it computes.

\[
\text{pipe} \{ \\
D_{Sk_1} \ Op_1, \\
D_{Sk_2} \ Op_2, \\
< \cdots >_1 \\
D_{Sk_n} \ Op_n \}
\]

\[
\text{comp (out } z, \text{ in } a) \{ \\
b = D_{Sk_1} \ Op_1 \ a, \\
c = D_{Sk_2} \ Op_2 \ b, \\
< \cdots >_1 \\
z = D_{Sk_n} \ Op_n \ y\}
\]

Same functional (sequential) semantic. They differ in the execution model. \textbf{pipe} stages run on on different set of PEs, \textbf{comp} stages run in sequence on the same set of PEs.
Rewriting rules: examples (2)

\[
\text{comp (out outvar, in invars)} \{ \begin{array}{l}
< \cdots >_1 \\
q = \text{map} \; Op_1 \; p, \\
< \cdots >_2 \\
r = \text{map} \; Op_2 \; q, \\
< \cdots >_3 
\end{array} \}
\]

\[
\text{comp (out outvar, in invars)} \{ \begin{array}{l}
< \cdots >_1 \\
q = \text{map} \; Op_1 \; p, \\
< \cdots >_2 \\
r = \text{map} \; (Op_2 \circ Op_1) \; p, \\
< \cdots >_3 
\end{array} \}
\]

map (backwards) distribution through functional composition. We do not require the two maps to be adjacent in the program code. Meta provides the program with the additional assignment only if the intermediate result \(q\) is referenced into \(< \cdots >_2\) or \(< \cdots >_3\)
Tool behaviour

- TL program
  - front-end
    - TL → internal repr.
  - dependence tree
    - locate applicable
      - transformations
        - set of rules
          - a rule
            - transform the program
              - TL program
                - (optimized)
          - rebuild TL from internal represent.
        - user dialog
          - dependence forest
            - TL rule set
              - TL cost calculus
                - front-end
                  - TL → internal repr.
Representing program and rules

Both program and rules are represented by *dependence trees*

- A *dependence tree* (DT) is a labelled tree
- DT directly represents the data dependence among skeletons
- if Sk1 directly uses data produced by Sk2
  then they are adjacent in DT
- DT is built starting from parse tree (PT) and data flow graph (DFG) of the program

_both PT and DFG can be build using standard tools_

_Since program and patterns \((L)\) are trees, the search for applicable rules reduces to subtree matching_
Representing program and rules (cont’ed)

A rule is represented by a dependence tree pair

Keys: [u]p [d]own

Advantageous: Always

Where: fcomp(f,g) = f(g)

Cost \((n/p)_{\text{op}} + 2L\)

Cost \(2(n/p)_{\text{op}} + L\)

- Circled figures represents variables
- fcomp is special node representing functional composition
Building up the **Dependence Tree**

**Parse Tree (PT)**

```
DPblock
  \-- ts
  \-- \-- scanL
  \-- \-- \-- *sa
  \-- \-- \-- as
  \-- \-- \-- ts

\-- \-- +
\-- \-- ds
```

**Node attributes**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>as</td>
<td>ys</td>
</tr>
<tr>
<td>zs</td>
<td></td>
</tr>
</tbody>
</table>

**Data Flow Graph (DFG)**

```
scanL
  \-- *sa
  \-- as

map2
  \-- *sa

reduce
  \-- +
  \-- ds
```

**Dependence Tree**

```
DPblock
  \-- reduce

\-- +
\-- map2

\-- *sa
\-- Arg(as)
\-- scanL

\-- *sa
\-- Arg(ys)
```
Building up the *Dependence Tree* (cont’ed)

**Data Flow Graph (DFG)**

```
In parameters
\( \langle x, y \rangle \) — DPblock
```

**Dependence Tree**

```
DPblock
\( a \) — Arg \( x \) — \( b \) — Arg \( y \) — \( d \) — Arg \( y \)
```

shared subtrees: keep them shared or replicate them?
Locating applicable rules: subtree matching

Problem: Match a set of patterns against many subject trees

Solution: Hoffman-O’Donnell bottom-up algorithm:
Two phases: 1) preprocessing of the rules, 2) matching

Good news:
- The preprocessing phase have to be repeated only if either the rules or the language have been changed
- Matching really fast (even in practice): a single traversal of $T$

Bad news:
- The preprocessing phase may be expensive
- but, it is fast for a broad class of pattern sets
- Details in the paper
1. Front End
   - Build TL program from specific TL spec.
   - TL program

2. Normalization
   - TL program
   - TL rules
   - dependence tree generation
   - list of dependence pattern trees
   - PT & DFG
   - scanner/parser

3. Rule Manager
   - list of matched rules
   - PTs & DFGs
   - data dependence analyzer
   - TL rules

4. Transformation Engine
   - pattern matching
   - patterns and user dialog
   - PT & DFG
   - rule list
   - list of matched rules
   - pattern forest generation
   - matches presentation and user dialog
   - rule to apply
   - tree transformation
   - dependence tree generation

5. Back End
   - TL program
   - dependence tree generation
   - PT & DFG
   - TL rules
   - list of matched rules
   - scanner/parser
   - data dependence analyzer
   - PTs & DFGs
   - rule list

Customizable via instantiation of the ADT

Build by LEX & YACC
Would you like apply any rule [0=Exit] 0

Matches Found:

Would you like apply any rule [0=Exit] 1

Matches Found:

Would you like apply any rule [0=Exit] 1

Matches Found:

Would you like apply any rule [0=Exit] 1

Matches Found:

Would you like apply any rule [0=Exit] 1

Matches Found:

Would you like apply any rule [0=Exit] 1

Matches Found:

Would you like apply any rule [0=Exit] 1
MSS example in Skel-BSP (9 out of 20 formulations)
Would you like to apply any rule [0=Exit] 0

1) rule n. 18 (farm introduction)

2) rule n. 18 (farm introduction)

3) rule n. 18 (farm introduction)

4) rule n. 18 (farm introduction)

5) rule n. 18 (farm introduction)

Matches Found:

1) rule n. 18 (farm introduction)

2) rule n. 13 (SAR-ARA)

Would you like to apply any rule [0=Exit] 2

1) rule n. 18 (farm introduction)

2) rule n. 10 (map fusion)

3) rule n. 10 (map fusion)

Matches Found:

1) rule n. 18 (farm introduction)

2) rule n. 10 (map fusion)

3) rule n. 10 (map fusion)

Would you like to apply any rule [0=Exit] 2

1) rule n. 17 (farm elimination)

Matches Found:

1) rule n. 17 (farm elimination)

2) rule n. 18 (farm introduction)

Would you like to apply any rule [0=Exit] 1

Matches Found:

1) rule n. 17 (farm elimination)

2) rule n. 18 (farm introduction)

Would you like to apply any rule [0=Exit] 2

Matches Found:

1) rule n. 17 (farm elimination)

2) rule n. 18 (farm introduction)

Would you like to apply any rule [0=Exit] 2

Matches Found:

1) rule n. 17 (farm elimination)

2) rule n. 18 (farm introduction)

Would you like to apply any rule [0=Exit] 1

Matches Found:

1) rule n. 17 (farm elimination)

2) rule n. 18 (farm introduction)

Would you like to apply any rule [0=Exit] 2

Matches Found:

1) rule n. 17 (farm elimination)

2) rule n. 18 (farm introduction)

Would you like to apply any rule [0=Exit] 1

Matches Found:

1) rule n. 17 (farm elimination)

2) rule n. 18 (farm introduction)

Would you like to apply any rule [0=Exit] 2

Matches Found:

1) rule n. 17 (farm elimination)

2) rule n. 18 (farm introduction)

Would you like to apply any rule [0=Exit] 1

Matches Found:

1) rule n. 17 (farm elimination)

2) rule n. 18 (farm introduction)
Implementation

- Prototype in Ocaml 2.02 (about 2000 lines of code)
- Tested under Windows’98 and Linux RH6.2
- Tested over 2 target languages and about 20 rules
- The implementation (except graphical interface) is based on a single ADT which describes the dependence tree and the function working on it
- The implementation can handle many Target Languages via instantiation of the ADT
Discussion

1. Why the tool is interactive?
   → Because the rewriting calculus of TL, in the general case, is not confluent in performance and the solution space (may) grow exponentially with the number rules

2. Does the tool make any decision about the rule to apply?
   → No. But, it can be extended with your own heuristics, if you have them
   → Currently Meta optimises Skel-BSP data-parallel-free programs with a standard sequence of rules. Such “Normal Form” is proved to be the fastest among the semantic-equivalent formulations that can be reached with these rules (Aldinucci, Danelutto. IASTED PDCS’99, Boston, USA)

3. Does the tool really optimise the program?
   → It really depend on the TL, the set of rule and the cost calculus, not on the tool. The tool make you happy playing with your new skeleton language, that’s it.
Conclusions

Meta implements a (meta) rewriting system for skeleton-based languages

- It is independent from the target language and the rules
- It only requires a three-tier language schema
- Can be equipped with heuristics to make decisions on the rule to apply
- It is already implemented on a platform-independent language (Ocaml)
- It has a (simple) graphical interface and it is fast
- Is is easy to modify and to extend
- ...
- It is free!
FAN rules (1)

**Rule SR-ARA**
\[
\begin{align*}
  b & = \text{reduce } Op2 \ (\text{scanL } Op1 \ a) \\
  b & = \text{proj1} \ (\text{reduce } Op3 \ (\text{pair} \ (a,a))) \\
  & \text{If } Op1 \text{ distributes forward over } Op2 \\
  (a_1, b_1) Op3 (a_2, b_2) & = (a_1 \ Op2 (b_1 Op1 a_2), b_1 Op2 b_2)
\end{align*}
\]

**Rule AR-RA**
\[
\begin{align*}
  b & = \text{reduce } Op1 \ (\text{proj1} \ a) \\
  b & = \text{proj1} \ (\text{reduce } Op2 \ a) \\
  & \text{If } Op1 \text{ distributes forward over } Op4 \\
  (a_1, b_1) Op2 (a_2, b_2) & = (a_1 Op1 a_2, b_1 Op1 b_2)
\end{align*}
\]

**Rule SAR-ARA**
\[
\begin{align*}
  c & = \text{reduce } Op2 \ (\text{proj1} \ (\text{scanL } Op1 \ a)) \\
  c & = \text{proj1} \ (\text{proj1} \ (\text{reduce } Op3 \ (\text{pair}(a,a)))) \\
  & \text{If } Op1 \text{ distributes forward over } Op4 \\
  (a_1, b_1) Op3 (a_2, b_2) & = (a_1 Op4 (b_1 Op1 a_2), b_1 Op4 b_2) \\
  (a_1, b_1) Op4 (a_2, b_2) & = (a_1 Op2 a_2, b_1 Op2 b_2)
\end{align*}
\]

**Rule CS-CM**
\[
\begin{align*}
  b & = \text{scanL} \ Op \ (\text{copy} \ n \ a) \\
  b & = \text{map}_f \ (\text{copy} \ n \ a) \\
  f \ i \ x & = \text{fst} (\text{repeat} \ i \ (x,x)) \\
  \text{repeat} \ k \ x & = \text{if } k = 0 \text{ then } x \text{ else } \text{repeat} \ (k \ \text{div} \ 2) \ (\text{if } (k \ \text{mod} \ 2 = 0) \text{ then } e \ x \text{ else } a \ x) \\
  e(t, u) & = (t, u Op u), o(t, u) = (t Op u, u Op u)
\end{align*}
\]
FAN rules (2)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Time left hand side</th>
<th>Time right hand side</th>
<th>Improves if</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR-ARA</td>
<td>$3 \cdot m + \log p \cdot (2 \cdot (t_s + t_w) + 3)$</td>
<td>$2 \cdot m + \log p \cdot (t_s + 2 \cdot t_w + 2)$</td>
<td>always</td>
</tr>
<tr>
<td>AR-RA</td>
<td>$m + \log p \cdot (t_s + t_w + 1)$</td>
<td>$2 \cdot m + \log p \cdot (t_s + 2 \cdot (t_w + 1))$</td>
<td>never</td>
</tr>
<tr>
<td>SAR-ARA</td>
<td>$3 \cdot m + \log p \cdot (2 \cdot (t_s + t_w) + 3)$</td>
<td>$5 \cdot m + \log p \cdot (t_s + t_w + 1)$</td>
<td>$(t_s + t_w + 2) \cdot \log p &gt; 2m$</td>
</tr>
<tr>
<td>CS-CM</td>
<td>$p \cdot t_s + m \cdot (t_w + \frac{tx}{p} + 1)$</td>
<td>$p \cdot t_s + m \cdot (t_w + \frac{tx}{p} + 1)$</td>
<td>always</td>
</tr>
<tr>
<td>$M#M#$</td>
<td>$2 \cdot m$</td>
<td>$m$</td>
<td>always</td>
</tr>
<tr>
<td>$M#AM-M#$</td>
<td>$4 \cdot m$</td>
<td>$m$</td>
<td>always</td>
</tr>
</tbody>
</table>