Remarks on cost analysis for patterns of parallelism

Vladimir Komendantsky

University of St Andrews, UK

15 May 2012
Université Paris-Est Créteil

Preliminary notes about Parallel Static Cost Analysis

1. Static performance analysis: prediction of run time of shapely skeletal parallel programs.
2. Model: Bounded Synchronous Parallelism (BSP), across various architectures.
3. Calculus of skeletons: Bird–Meertens Formalism (BMF).

Notes

formalism	foundational data structures
BSP	arrays
BMF	lists
- skeletons $\subset$ parallel patterns

Research programme

1. Develop an equational theory of patterns of parallelism supporting program refinement.
2. Develop a cost model that supports comparisons between refinements of a program.
3. Develop an algorithm for cost-driven optimal refinement search.

Theory of lists: Bird–Meertens Formalism

Example (Maximum segment sum)

The maximum of the sums of all the segments of a given list:

$$
\begin{array}{ll}
 & mss \\
= & \quad \{ \text{ definition of } mss \ \} \\
 & max/ \cdot +/* \cdot segs \\
= & \quad \{ \text{ definition of } segs \ \} \\
 & max/ \cdot +/* \cdot +\!\!+/ \cdot tails* \cdot inits \\
= & \quad \{ \text{ distributivity of map and fold } \} \\
 & max/ \cdot (max/ \cdot +/* \cdot tails)* \cdot inits \\
= & \quad \{ \text{ Horner's rule where } x \odot y = max\ (x+y)\ 0 \ \} \\
 & max/ \cdot \odot/_0^L * \cdot inits \\
= & \quad \{ \text{ defining property of left scan } \} \\
 & max/ \cdot \odot/\!/_0^L
\end{array}
$$

Where to add cost information to the theory of lists

Problem (Two different intermediate structures of *fold*)

$$/ : \ (a \to b \to a) \to a \to [b] \to a$$

1. $(zip\ +)/ : \ [a] \to [[b]] \to [a]$
   pointwise addition of lists; intermediate sums have the same size.
2. $+\!\!+/ : \ [a] \to [[b]] \to [a]$
   append of all the given lists; intermediate lists are the same or larger.

Example (Shapes of functions and data)

$$\#(zip\ +)\ \#x\ \#y \ \stackrel{def}{=}\ \text{if } \#x = \#y \text{ then } \#x \text{ else } error$$
$$\#+\!\!+\ \#x\ \#y \ \stackrel{def}{=}\ ?$$

Connection with bulk synchronous parallelism

Definition

Shapely program A program is **shapely** if the shape of its result is determined by the shapes of its inputs.

Shape analysis of a non-shapely program F:

1. Decompose F into a sequence of subprograms in which all intermediate shapes can be determined but not that of the result. *Try to match subprograms with barrier syncs and data redistributions.*
2. Determine shapes at the beginning of each subprogram (*reshaping point*).

Shaped language

Type system:

$$
\begin{array}{rrl}
\textbf{atomic types} & \delta ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{float} \mid \ldots \\
\textbf{array types} & \alpha ::= & X \mid \delta \mid [\alpha] \\
\textbf{shape types} & \sigma ::= & \tilde{\delta} \mid \#\alpha \\
\textbf{data types} & \tau ::= & \alpha \mid \sigma \\
\textbf{phrase types} & \theta ::= & U \mid \#U \mid \mathit{exp}\ \tau \mid \mathit{var}\ \alpha \mid \mathit{comm} \mid \theta \to \theta \\
\textbf{type schemes} & \phi ::= & \theta \mid \forall_\alpha X.\phi \mid \forall_\theta U.\phi
\end{array}
$$

Terms:

$$t ::= x \mid c \mid \lambda x.t \mid t\ t \mid t \text{ where } x = t$$

Examples of shapes

- $m \times n$ matrix of integers:

$$[m,n]^{\tilde{int}} : \#[int]$$

- vector of length m of vectors of length n:

$$[[m,n]^{\tilde{int}}] : \#[[int]]$$

- shape of a map:

$$\#map = \lambda\ f\ a.\ extendShape\ a\ (f\ (entryShape\ a))$$

where $extendShape\ a^{\tilde{\delta}}\ b^{\tilde{\delta}} = [a,b]^{\tilde{\delta}}$.

Parallelisation of a shaped program

- Specify a distribution of data structures across processors, that is, *match the shape of data with the shape of the processor network*.

Definition (Distribution / implicit parallelism)

A (simple) distribution for an array of type $[a]$ is an array of type $[[a]]$ where

- the outer shape: a virtual array of processors;
- the inner shape: the blocks assigned to each of the processors.

Cost analysis

The type $cost = \tilde{float}$ forms a commutative monoid under addition.

Definition (Cost of a superstep in the BSP model)

$$\text{cost of a superstep} = \max W_i + \max h_i * g + L$$

where

- W_i: local processing time on processor i,
- h_i: number of packets sent or received by the processor i,
- g: marginal cost of sending a packet (communication throughput / processor throughput),
- L: cost of barier synchronisation

BSP hardware parameters are (p, L, g) of type $H = size \times cost \times cost$.
The cost of a function $f : [a] \to [b]$ is given by a function

$$\#f : \#[a] \to \#[b] \times (H \to cost)$$

Cost monad

The functor $M\theta = \theta \times (H \to \mathit{cost})$ with pointwise operations is a commutative monad.

$$\begin{array}{rcl} M_0(\mathit{exp}\ \tau) & = & \mathit{exp}\ \#\tau \\ M_0(\theta_1 \times \theta_2) & = & M_0(\theta_1) \times M_0(\theta_2) \\ M_0(\theta_1 \to \theta_2) & = & M_0(\theta_1) \to M(M_0(\theta_2)) \end{array}$$

The cost of $t : \theta$ is $c(t) : MM_0(\theta)$, for example:

$$\begin{array}{ll} c(\mathit{map}) : & M(\alpha \to M\beta) \to M([\alpha] \to [\beta]) \\ c(\mathit{map}) = & \langle\ \lambda\ f.\ \langle\ \lambda\ a. \langle\ \#\mathit{map}\ (\mathit{fst} \cdot f)\ a, \\ & \qquad\qquad\qquad \lambda\ h.\ (\mathit{snd}\ (f\ (\mathit{entryShape}\ a))\ h) * b(a,h) \rangle, \\ & \qquad\quad \lambda\ h.\ 0 \rangle, \\ & \quad \lambda\ h.\ 0 \rangle \end{array}$$

where $b(a,h)$ is the size of the blocks of a with respect to the hardware h.

Patterns of data and computations: first approximation

$d ::=$		**data structure**
	$\hat{x}$	**matchable symbol**
	d t	**compound data**
$c ::=$		**computation**
	$\hat{x}$	**matchable symbol**
	$c \to t$	**compound computation**

Pattern calculus

$t ::=$	**untyped term**
x	**variable symbol**
$\hat{x}$	**matchable symbol**
$t\ t$	**application**
$[\theta]\ t \to t$	**case**

matching	$\{u/[\theta]\ p\}$
match reduction	$([\theta]\ p \to s)\ u \rightsquigarrow \{u/[\theta]\ p\}\ s$
Leibniz quality	$\mathrm{eq} = \lambda\ x.\ ([]\ x \to \mathrm{true} \mid \lambda\ y.\ \mathrm{false})$

Some notation:

$$\lambda\ x.\ t \underset{def}{=} [x]\ \hat{x} \to t$$

$$\mathrm{car} \underset{def}{=} [x, y]\ \hat{x}\ \hat{y} \to x$$

Divide and conquer in the pattern calculus

Views (additional matching clause):

$$\{u/[\theta]\ \text{view}\ f\ \ p\} = \{f\ \ u/[\theta]\ p\}$$

dc *divide combine conquer condition* $=$
  $|\ [x]\ \mathit{condition}\ \hat{x} \rightarrow \mathit{conquer}\ \ x$
  $|\ [x, y]\ \text{view}\ \mathit{divide}\ (\hat{x}, \hat{y}) \rightarrow \mathit{combine}\ (f\ \ x)\ (f\ \ y)$
  where $f =$ dc *divide combine conquer condition*

Remarks on implementations-in-progress

- The pattern calculus is capable of complementing BMF in certain important points.
- It is a computationally well-behaved formalism (progress, confluence) and therefore is tangible to interactive theorem proving.
- The approach of the shape calculus where shape types and shape functions form a distinct syntactic category can be taken further and applied to pattern calculus.
- We can use matching and substitution in many novel ways compared to the standard FP, in particular, **instead of dependent types**.

To be continued.