Much Ado about Two

A Pearl on Parallel Prefix Computation

Janis Voigtländer

Technische Universität Dresden

POPL'08

Parallel Prefix Computation

Given: inputs $x_1, \ldots, x_n$ and an associative operation $\oplus$

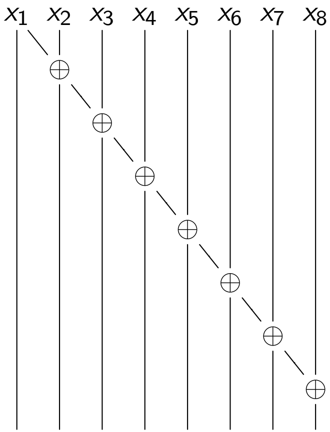
Task: compute the values $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n$

Parallel Prefix Computation

Given: inputs $x_1, \ldots, x_n$ and an associative operation $\oplus$

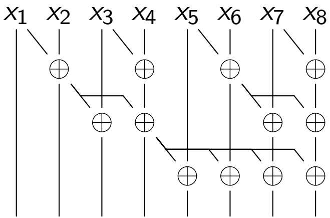
Task: compute the values $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n$

Solution:



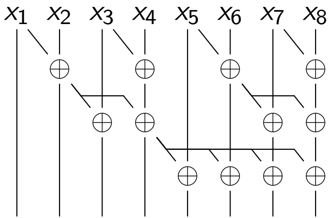
Parallel Prefix Computation

Alternative:

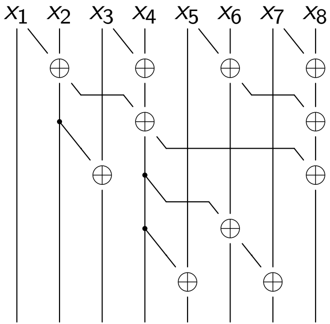


Parallel Prefix Computation

Alternative:

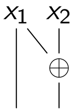


Or:

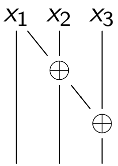


Or: ...

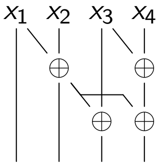
A la [Sklansky 1960]:



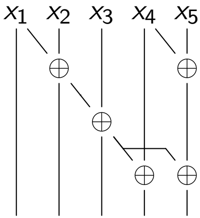
A la [Sklansky 1960]:



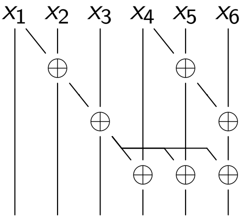
A la [Sklansky 1960]:



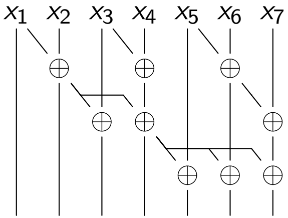
A la [Sklansky 1960]:



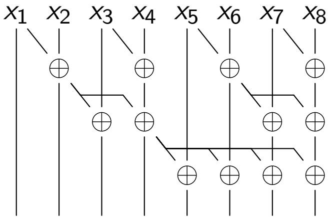
A la [Sklansky 1960]:



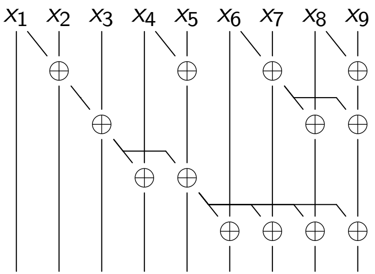
A la [Sklansky 1960]:



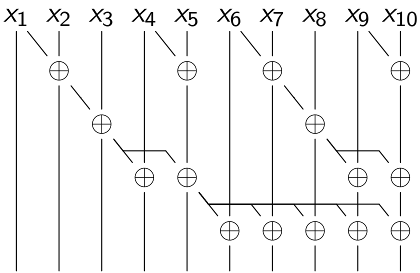
A la [Sklansky 1960]:



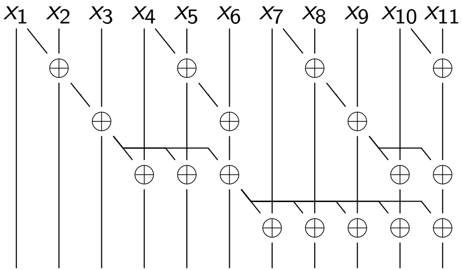
A la [Sklansky 1960]:



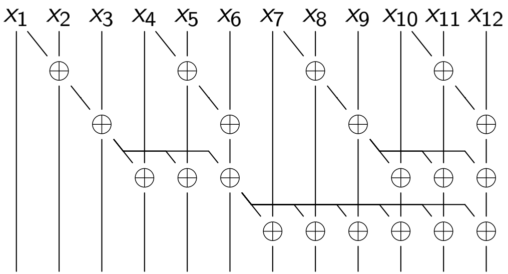
A la [Sklansky 1960]:



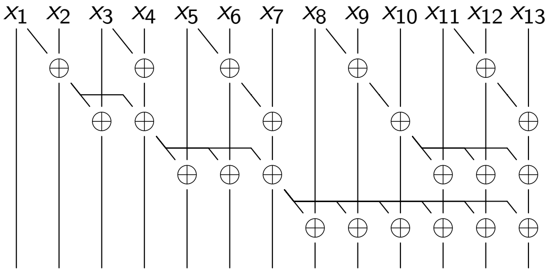
A la [Sklansky 1960]:



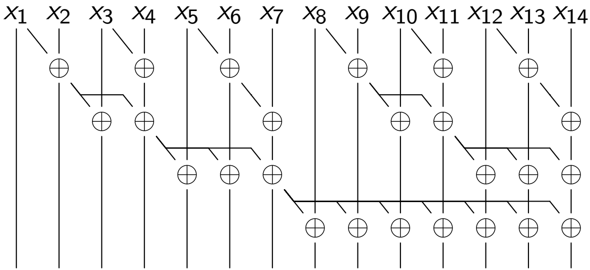
A la [Sklansky 1960]:



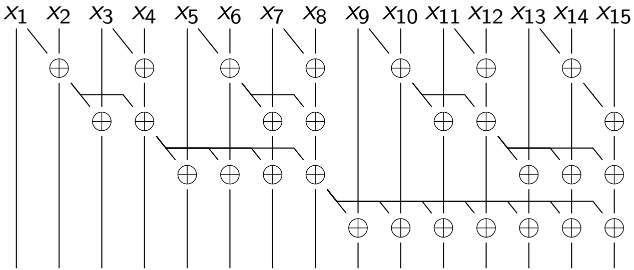
A la [Sklansky 1960]:



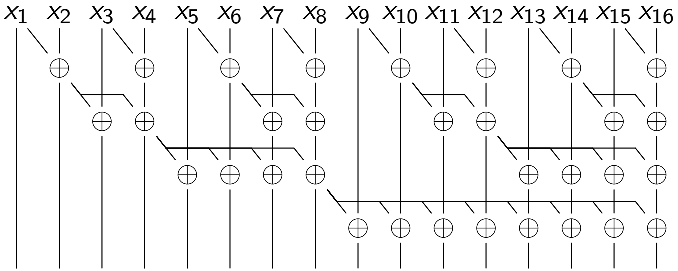
A la [Sklansky 1960]:



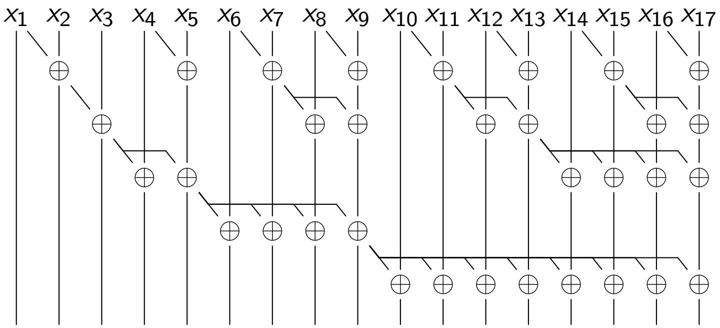
A la [Sklansky 1960]:



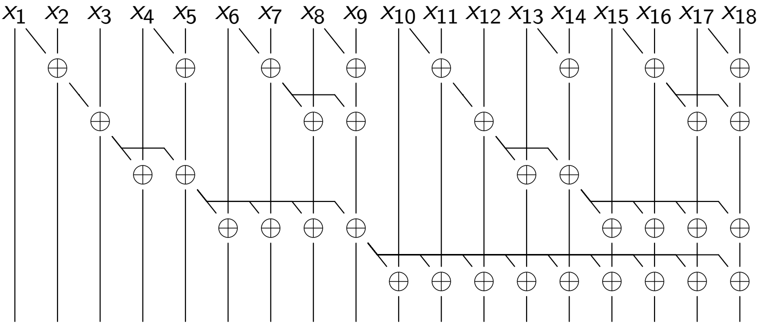
A la [Sklansky 1960]:



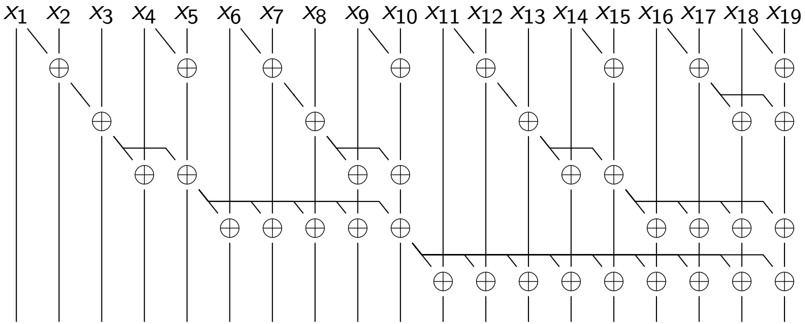
A la [Sklansky 1960]:



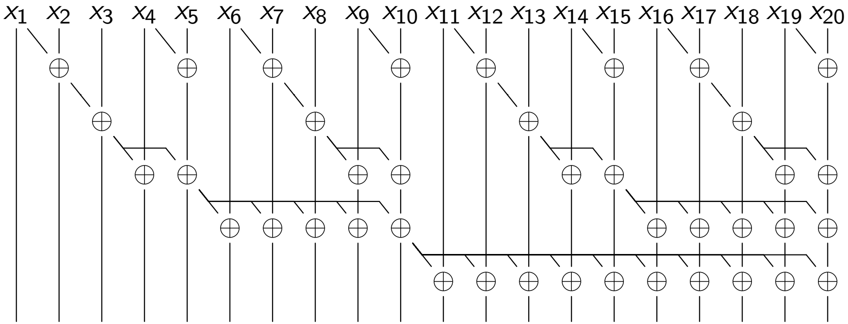
A la [Sklansky 1960]:



A la [Sklansky 1960]:



A la [Sklansky 1960]:



Sklansky's Method in Haskell

$$
\begin{aligned}
&\mathit{sklansky} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\mathit{sklansky}\ (\oplus)\ [x] = [x] \\
&\mathit{sklansky}\ (\oplus)\ \mathit{xs}\ = \mathit{us} +\!\!+ \mathit{vs} \\
&\qquad \textbf{where}\ t \qquad\quad\ = (\mathit{length}\ \mathit{xs} + 1)\ \text{`}\mathit{div}\text{`}\ 2 \\
&\qquad\qquad\quad (\mathit{ys}, \mathit{zs}) = \mathit{splitAt}\ t\ \mathit{xs} \\
&\qquad\qquad\quad \mathit{us} \qquad\ = \mathit{sklansky}\ (\oplus)\ \mathit{ys} \\
&\qquad\qquad\quad \mathit{vs} \qquad\ = [(\mathit{last}\ \mathit{us}) \oplus v \mid v \leftarrow \mathit{sklansky}\ (\oplus)\ \mathit{zs}]
\end{aligned}
$$

Sklansky's Method in Haskell

$$
\begin{array}{l}
\mathit{sklansky} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
\mathit{sklansky}\ (\oplus)\ [x] = [x] \\
\mathit{sklansky}\ (\oplus)\ \mathit{xs}\ = \mathit{us} + \!\!\! + \mathit{vs} \\
\qquad \textbf{where}\ \begin{array}[t]{lcl}
t & = & (\mathit{length}\ \mathit{xs} + 1)\ `\mathit{div}`\ 2 \\
(\mathit{ys}, \mathit{zs}) & = & \mathit{splitAt}\ t\ \mathit{xs} \\
\mathit{us} & = & \mathit{sklansky}\ (\oplus)\ \mathit{ys} \\
\mathit{vs} & = & [(\mathit{last}\ \mathit{us}) \oplus v \mid v \leftarrow \mathit{sklansky}\ (\oplus)\ \mathit{zs}]
\end{array}
\end{array}
$$

Wanted: reasoning principles, verification techniques, systematic testing approach, ...

A Knuth-like 0-1-2-Principle

Given: $$\begin{aligned} & \mathit{serial} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\ & \mathit{serial}\ (\oplus)\ (x : \mathit{xs}) = \mathit{go}\ x\ \mathit{xs} \\ & \qquad \textbf{where}\ \mathit{go}\ x\ [\,] \qquad\quad = [x] \\ & \qquad\qquad\quad\ \ \mathit{go}\ x\ (y : \mathit{ys}) = x : \mathit{go}\ (x \oplus y)\ \mathit{ys} \end{aligned}$$

A Knuth-like 0-1-2-Principle

Given: $\mathit{serial} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned} &\mathit{serial}\ (\oplus)\ (x : \mathit{xs}) = \mathit{go}\ x\ \mathit{xs} \\ &\qquad \textbf{where}\ \mathit{go}\ x\ [\,] \qquad\quad = [x] \\ &\qquad\qquad\quad \mathit{go}\ x\ (y : \mathit{ys}) = x : \mathit{go}\ (x \oplus y)\ \mathit{ys} \end{aligned}$$

$\mathit{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

A Knuth-like 0-1-2-Principle

Given: $$\begin{aligned}&\mathit{serial} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]\\&\mathit{serial}\ (\oplus)\ (x : xs) = go\ x\ xs\\&\qquad\textbf{where}\ go\ x\ []\qquad\quad = [x]\\&\qquad\qquad\quad go\ x\ (y : ys) = x : go\ (x \oplus y)\ ys\end{aligned}$$

$$\mathit{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

$$\textbf{data}\ \mathit{Three} = \mathit{Zero} \mid \mathit{One} \mid \mathit{Two}$$

A Knuth-like 0-1-2-Principle

Given: $\mathit{serial} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}
&\mathit{serial}\ (\oplus)\ (x : \mathit{xs}) = \mathit{go}\ x\ \mathit{xs} \\
&\quad \textbf{where}\ \mathit{go}\ x\ [\,] \qquad\quad = [x] \\
&\qquad\qquad\ \ \mathit{go}\ x\ (y : \mathit{ys}) = x : \mathit{go}\ (x \oplus y)\ \mathit{ys}
\end{aligned}$$

$\mathit{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

data $\mathit{Three} = \mathit{Zero} \mid \mathit{One} \mid \mathit{Two}$

Theorem: If for every $\mathit{xs} :: [\mathit{Three}]$ and associative $(\oplus) :: \mathit{Three} \to \mathit{Three} \to \mathit{Three}$,

$$\mathit{candidate}\ (\oplus)\ \mathit{xs} \ \equiv\ \mathit{serial}\ (\oplus)\ \mathit{xs}\,,$$

then the same holds for every type τ, $\mathit{xs} :: [\tau]$, and associative $(\oplus) :: \tau \to \tau \to \tau$.

Why 0-1-2? And How?

A question: What can *candidate* $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$ do, given an operation $\oplus$ and input list $[x_1, \ldots, x_n]$?

Why 0-1-2? And How?

A question: What can *candidate* $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$ do, given an operation $\oplus$ and input list $[x_1, \ldots, x_n]$?

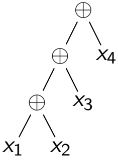
The answer: Create an output list consisting of expressions built from $\oplus$ and $x_1, \ldots, x_n$. Independently of the α-type !

Why 0-1-2? And How?

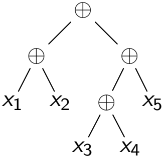
A question: What can *candidate* :: $(\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$ do, given an operation $\oplus$ and input list $[x_1, \ldots, x_n]$?

The answer: Create an output list consisting of expressions built from $\oplus$ and $x_1, \ldots, x_n$. Independently of the α-type !

Among these expressions, there are good ones:



,



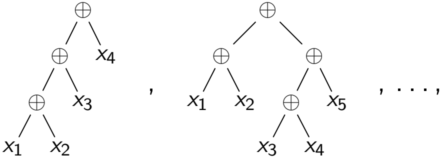
, ... ,

Why 0-1-2? And How?

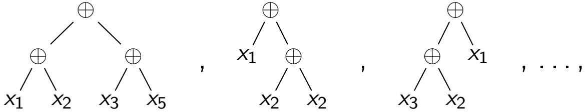
A question: What can *candidate* :: $(\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$ do, given an operation $\oplus$ and input list $[x_1, \ldots, x_n]$?

The answer: Create an output list consisting of expressions built from $\oplus$ and $x_1, \ldots, x_n$. Independently of the α-type !

Among these expressions, there are good ones:

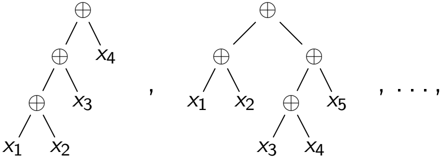


bad ones:

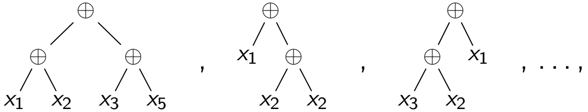


Why 0-1-2? And How?

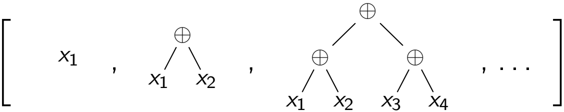
Among these expressions, there are good ones:



bad ones:



and ones in the wrong position:



That's How!

Let

$\oplus_1$	*Zero*	*One*	*Two*
Zero	*Zero*	*One*	*Two*
One	*One*	*Two*	*Two*
Two	*Two*	*Two*	*Two*

and

$\oplus_2$	*Zero*	*One*	*Two*
Zero	*Zero*	*One*	*Two*
One	*One*	*One*	*Two*
Two	*Two*	*One*	*Two*

That's How!

Let

$\oplus_1$	*Zero*	*One*	*Two*
Zero	*Zero*	*One*	*Two*
One	*One*	*Two*	*Two*
Two	*Two*	*Two*	*Two*

and

$\oplus_2$	*Zero*	*One*	*Two*
Zero	*Zero*	*One*	*Two*
One	*One*	*One*	*Two*
Two	*Two*	*One*	*Two*

If *candidate* ($\oplus_1$) is correct on each list of the form

$$[(Zero,)^* \; One \; (,Zero)^* \; (,Two)^*]$$

That's How!

Let

$\oplus_1$	*Zero*	*One*	*Two*
Zero	*Zero*	*One*	*Two*
One	*One*	*Two*	*Two*
Two	*Two*	*Two*	*Two*

and

$\oplus_2$	*Zero*	*One*	*Two*
Zero	*Zero*	*One*	*Two*
One	*One*	*One*	*Two*
Two	*Two*	*One*	*Two*

If *candidate* $(\oplus_1)$ is correct on each list of the form

$$[(Zero,)^* \; One \; (, Zero)^* \; (, Two)^*]$$

and *candidate* $(\oplus_2)$ is correct on each list of the form

$$[(Zero,)^* \; One, Two \; (, Zero)^*]$$

then *candidate* is correct for associative $\oplus$ at arbitrary type.

A Knuth-like 0-1-2-Principle

Given: $\mathit{serial} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}
&\mathit{serial}\ (\oplus)\ (x : \mathit{xs}) = \mathit{go}\ x\ \mathit{xs} \\
&\quad \textbf{where}\ \mathit{go}\ x\ [\,] \qquad\quad = [x] \\
&\qquad\qquad \mathit{go}\ x\ (y : \mathit{ys}) = x : \mathit{go}\ (x \oplus y)\ \mathit{ys}
\end{aligned}$$

$\mathit{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$\textbf{data}\ \mathit{Three} = \mathit{Zero} \mid \mathit{One} \mid \mathit{Two}$

Theorem: If for every $\mathit{xs} :: [\mathit{Three}]$ and associative $(\oplus) :: \mathit{Three} \to \mathit{Three} \to \mathit{Three}$,

$$\mathit{candidate}\ (\oplus)\ \mathit{xs} \quad \equiv \quad \mathit{serial}\ (\oplus)\ \mathit{xs}\,,$$

then the same holds for every type τ, $\mathit{xs} :: [\tau]$, and associative $(\oplus) :: \tau \to \tau \to \tau$.

The Overall Proof

- To get going, uses relational parametricity [Reynolds 1983] to derive a free theorem from *candidate*'s type [Wadler 1989].
- Remaining proof largely done by program calculation. (But also a bit "by picture".)
- Formalisation available in Isabelle/HOL:

  S. Böhme.
  Much Ado about Two. Formal proof development.
  The Archive of Formal Proofs.
  `http://afp.sf.net/entries/MuchAdoAboutTwo.shtml`

References I

R.P. Brent and H.T. Kung.
The chip complexity of binary arithmetic.
In *ACM Symposium on Theory of Computing, Proceedings*, pages 190–200. ACM Press, 1980.

G.E. Blelloch.
Prefix sums and their applications.
In J.H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 35–60. Morgan Kaufmann, 1993.

N.A. Day, J. Launchbury, and J. Lewis.
Logical abstractions in Haskell.
Haskell Workshop, 1999.

References II

D.E. Knuth.
The Art of Computer Programming, volume 3: Sorting and Searching.
Addison-Wesley, 1973.

J.C. Reynolds.
Types, abstraction and parametric polymorphism.
In *Information Processing, Proceedings*, pages 513–523.
Elsevier Science Publishers B.V., 1983.

M. Sheeran.
Searching for prefix networks to fit in a context using a lazy functional programming language.
Hardware Design and Functional Languages, 2007.

References III

- J. Sklansky.
  Conditional-sum addition logic.
  IRE Transactions on Electronic Computers, EC-9(6):226–231, 1960.

- P. Wadler.
  Theorems for free!
  In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.