Optimization of Skeletal Parallel Programs

Zhenjiang Hu
University of Tokyo & JST

July 10th, 2002

Joint work with Wei-Ngan Chin, Hideya Iwasaki, Masato Takeichi

Parallel Programming with Data Parallel Skeletons

Parallel Data Structures + Parallel Computation Patterns

Polynomial computation

Bracket matching problem

The Advantages of Skeletal Parallel Programming

- Single control thread $\Rightarrow$ Understandable
- Explicit parallelism $\Rightarrow$ Parallelizable
- Abstract skeleton $\Rightarrow$ Architecture independent

BMF Parallel Model

- A kind of *program calculus* for calculating efficient programs (Bird:89)
- *Parallel data structure*: parallel lists $[x_1, x_2, \ldots, x_n]$
- *Primitives* (Skillicorn:90)

$$
\begin{array}{llcll}
map: & f * [x_1, \ldots, x_n] & = & [f\ x_1, \ldots, f\ x_n] & O(1) \\
reduce: & \oplus/ [x_1, \ldots, x_n] & = & x_1 \oplus \cdots \oplus x_n & O(\log n) \\
scan: & \oplus\!\!\!/\!\!\!/ [x_1, \ldots, x_n] & = & [\iota_\oplus, x_1, \ldots, x_1 \oplus \cdots \oplus x_n] & O(\log n) \\
zip: & zip\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] & = & [(x_1, y_1), \ldots, (x_n, y_n)] & O(1)
\end{array}
$$

Architecture Independent Parallel Model!

Developing **Efficient** Skeletal Parallel Programs is Difficult!

- Difficult to choose *appropriate* skeletons and to find a good *combination* of them.
  - Having few primitive skeletons ⇒ difficult to combine them
  - Having too many skeletons ⇒ difficult to choose appropriate ones
- Difficult to optimize skeletal programs while *preserving* parallelism
  - *composition* of skeletons ⇒ communication cost
  - *nested* use of skeletons ⇒ load unbalance

XML Tag Matching Problem

<Person>
<Name> *Zhenjiang Hu* </Name>
<Affiliation>
<Name>*University of Tokyo*</Name>
<Address>*Hongo 7-3-1, Bunkyo, Tokyo* <Address>
</Affiliation>
<Affiliation> *Presto 21, JST*</Affiliation>
<Email> *hu@mist.i.u-tokyo.ac.jp* </Email>
</Person>

Difficult to develop $O(\log N)$ skeletal parallel program. (Cole:95)

Maximum Segment Sum Problems

- 1-Dimensional MSS Problem:

$$mss\ [3, -4, \underline{2, -1, 6}, -3] = 7$$

- 2-Dimensional MSS Problem:

$$\text{mss2} \quad \begin{array}{l} [\,[\ 4, -1, -5], \\ \ \ [-1, \ \ 9, \ \ 6], \\ \ \ [-1, -2, \ \ 3]\,] \end{array} \quad = \quad 16$$

Deriving efficient skeletal parallel programs for them is hard [Gorlatch96,Hu97].

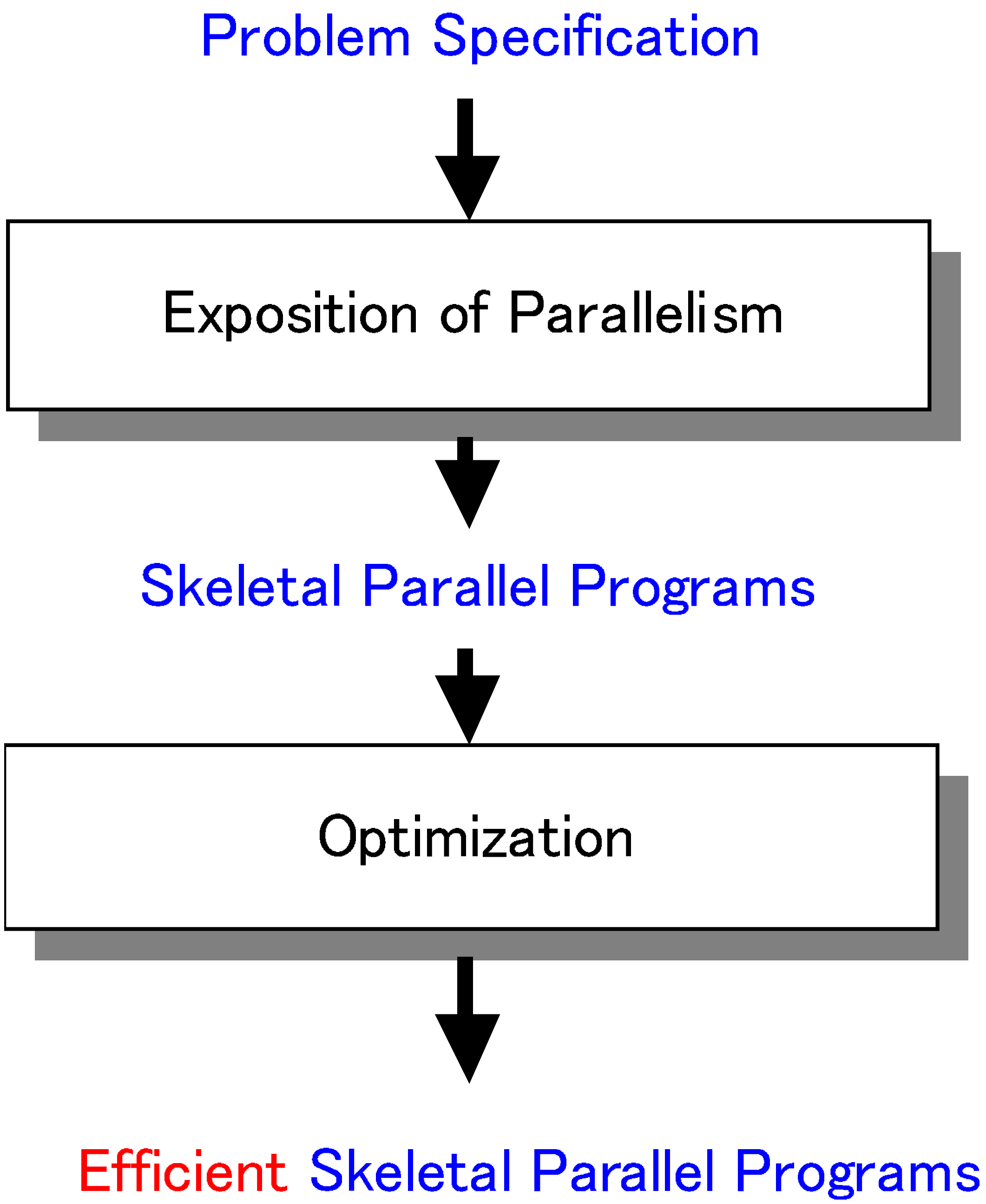
Objective

We are aiming at

$\boxed{\textit{a general and mechanical framework}}$

for optimizing skeletal parallel programs

- *General*: suitable for guiding human to develop efficient skeletal parallel programs;
- *Mechanical*: Suitable for being implemented in a compiler.


Problem Specification
Exposition of Parallelism
Skeletal Parallel Programs
Optimization
Efficient Skeletal Parallel Programs

Specification to Skeletal Program

Problem Specification

$\Downarrow$

Exposition of Parallelism

– compositional style –

– decomposition of single recursive definitions –

$\Downarrow$

Skeletal Parallel Program

Compositional Style

$$
\begin{array}{rcl}
mss & = & maximum \circ (\underline{sum*}) \circ segs \\
 & & \text{where} \\
 & & \quad segs = concat \circ (tails*) \circ inits \\
 & & \quad sum = +/ \\
 & & \quad maximum = \uparrow / \\
 & & \quad inits = +\!\!\!+\!\!\not\!\!\!\!\!\!\!\!\!/\!\!-\, \circ [\cdot]* \\
 & & \quad \ldots
\end{array}
$$

Compositional programming is not that easy!

- Associative operators should be used in *reduce* and *scan*.
- Some problems are more natural to be described *recursively*.

Example: Solving Bracket Matching Problem using a Stack (Cole95):

$$
\begin{array}{lcl}
bm\ [\,]\ s & = & isEmpty\ s \\
bm\ (a : x)\ s & = & \text{if } isOpen\ a \text{ then } bm\ x\ (push\ a\ s) \\
 & & \text{elseif } isClose\ a \text{ then } noEmpty\ s\ \wedge\ match\ a\ (top\ s)\ \wedge \\
 & & \qquad\qquad\qquad\qquad bm\ x\ (pop\ s) \\
 & & \text{else } bm\ x\ s
\end{array}
$$

Diffusion Transformation (PEPM99)

Recursive Definitions

$$\Downarrow$$

$$\boxed{\textit{Diffusion}}$$

$$\Downarrow$$

Programs in terms Parallel Skeletons

Simple Diffusion: Homomorphism Lemma (Bird87)

$$\begin{array}{lcl} h\ [a] & = & f\ a \\ h\ (x +\!\!+\, y) & = & h\ x \oplus h\ y \end{array}$$

$$\Updownarrow$$

$$h = \ (\oplus/) \ \circ \ (f*)$$

Parallel Programming via Homomorphism

A (Sequential) Recursive Function

$$\Downarrow$$

$$\begin{array}{lcl} h\ [a] & = & f\ a \\ h\ (x +\!\!+\, y) & = & h\ x \oplus h\ y \end{array}$$

(Cole:95, Gorlatch:96-97, Hu&Chin:98)

An Example

Deriving Homomorphism by Context Preservation (POPL98,ICCL98)

$$\begin{array}{lcl} \mathit{length}\ [a] & = & 1 \\ \mathit{length}\ ([a] +\!\!+\, x) & = & 1 + \mathit{length}\ x. \end{array}$$

- Generalization.

$$\mathit{length}\ (x' +\!\!+\, x) = uG\ x' + \mathit{length}\ x.$$

- Induction on x'.

$$uG\ [a] = 1$$

$$\begin{array}{l} length\ ((x'_1 \mathbin{+\!\!+} x'_2) \mathbin{+\!\!+} x) \\ = \underline{uG\ (x'_1 \mathbin{+\!\!+} x'_2)} + length\ x \end{array} \qquad \begin{array}{l} length\ (x'_1 \mathbin{+\!\!+} (x'_2 \mathbin{+\!\!+} x)) \\ = uG\ x'_1 + length\ (x'_2 \mathbin{+\!\!+} x) \\ = \underline{uG\ x'_1 + uG\ x'_2} + length\ x \end{array}$$

$$uG\ (x'_1 \mathbin{+\!\!+} x'_2) = uG\ x'_1 + uG\ x'_2$$

- Tupling *length* with *uG* gives a homomorphism.

A More Involved Example: mss

$$
\begin{array}{lcl}
mss\ [a] & = & [a] \\
mss\ ([a] +\!\!+\, x) & = & mis\ ([a] +\!\!+\, x) \uparrow\ mss\ x
\end{array}
$$

$$
\begin{array}{lcl}
mis\ [a] & = & [a] \\
mis\ ([a] +\!\!+\, x) & = & \mathit{if}\ mis\ x + \underline{a} < \underline{a}\ \mathit{then}\ mis\ x + \underline{a}\ \mathit{else}\ \underline{a}
\end{array}
$$

$$
\begin{array}{lcl}
mss([x]) & = & x \\
mss(xr \mathbin{+\!\!+} xs) & = & min2(mss(xr), min2(uT(xr) + mis(xs), mss(xs))) \\[1em]
mis([x]) & = & x \\
mis(xr \mathbin{+\!\!+} xs) & = & \textbf{if}\ mis(xs) \le uH(xr)\ \textbf{then}\ mis(xs) + uG(xr)\ \textbf{else}\ mis(xr) \\[1em]
uH([x]) & = & 0 \\
uH(xr \mathbin{+\!\!+} xs) & = & min2(uH(xs), uH(xr) - uG(xs)) \\[1em]
uT([x]) & = & x \\
uT(xr \mathbin{+\!\!+} xs) & = & min2(uT(xr) + uG(xs), uT(xs)) \\[1em]
uG([x]) & = & x \\
uG(xr \mathbin{+\!\!+} xs) & = & uG(xs) + uG(xr)
\end{array}
$$

Diffusing Recursion with Accumulation Parameter

$$
\begin{array}{lcl}
\textsf{accumulate}\ [\,]\ c & = & g_1\ c \\
\textsf{accumulate}\ (a:x)\ c & = & k(a,c)\ \oplus\ \textsf{accumulate}\ x\ (c \otimes g_2\ a)
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lcl}
\textsf{accumulate}\ x\ c & = & \textbf{let}\ cs' \mathbin{+\!\!+} [c'] = (c\otimes) * (\otimes\#_{(}g_2 * x)) \\
 & & \qquad ac = zip\ x\ cs' \\
 & & \textbf{in}\ (\oplus/\,(k * ac)) \oplus (g_1\ c')
\end{array}
$$

$\boxed{\text{Trick: use } scan \text{ to specify dependency!}}$

Example: Simple Bracket Matching Problem

Initial Specification

$$
\begin{array}{lcl}
sbp\ [\,]\ c & = & c == 0 \\
sbp\ (a : x)\ c & = & \textbf{if}\ a == \text{'('}\ \textbf{then}\ sbp\ x\ (c+1) \\
 & & \textbf{else if}\ a == \text{')'}\ \textbf{then}\ c > 0\ \wedge\ sbp\, x\, (c-1) \\
 & & \textbf{else}\ sbp\ x\ c.
\end{array}
$$

Merging Recursive Calls

$$
\begin{array}{lcl}
sbp\ [\,]\ c & = & g_1\ c \\
sbp\ (a:x)\ c & = & k(a,c)\ \wedge\ sbp\ x\ (c + g_2\ a)
\end{array}
$$

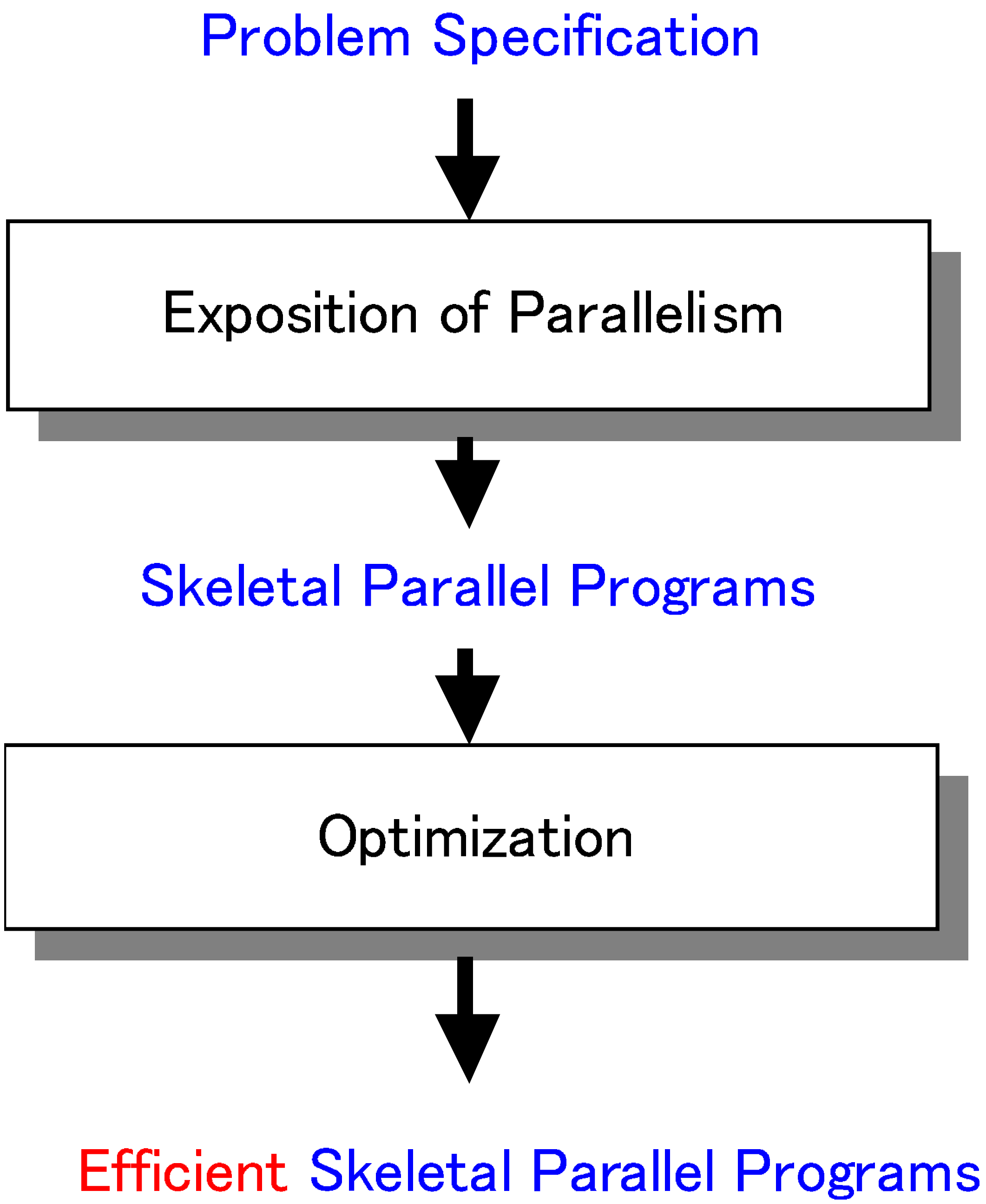
where

$$
\begin{array}{lcl}
g_1\ c & = & c == 0 \\
k(a,c) & = & \textbf{if}\ a == \text{'('}\ \textbf{then}\ True \\
 & & \textbf{else if}\ a == \text{')'}\ \textbf{then}\ c > 0\ \textbf{else}\ True \\
g_2\ a & = & \textbf{if}\ a == \text{'('}\ \textbf{then}\ 1 \\
 & & \textbf{else if}\ a == \text{')'}\ \textbf{then}\ (-1)\ \textbf{else}\ 0
\end{array}
$$

Diffusion

$$
\begin{array}{rcl}
sbp\ x\ c & = & \textbf{let}\ cs' +\!\!+ [c'] = (c+) * ((+) \#_{(} g_2 * x)) \\
 & & \qquad ac = zip\ x\ cs' \\
 & & \textbf{in}\ (\wedge/\ (k * ac)) \oplus (g_1\ c')
\end{array}
$$

$\boxed{\text{A } O(\log N) \text{ skeletal parallel program!}}$


Problem Specification
Exposition of Parallelism
Skeletal Parallel Programs
Optimization
Efficient Skeletal Parallel Programs

Optimization of Skeletal Parallel Programs

Skeletal Parallel Programs

$\Downarrow$

Mechanical Optimization

– *fusion*: eliminating cost due to composition of skeletons –

– *flattening*: eliminating cost due to nested use of skeletons –

$\Downarrow$

Efficient Skeletal Parallel Programs

Skeletal Programs for mss: Revisited

$$
\begin{array}{lcl}
mss & = & maximum \circ (\underline{sum*}) \circ segs \\
& & \text{where} \\
& & \quad segs = concat \circ (tails*) \circ inits \\
& & \quad sum = +/ \\
& & \quad maximum = \uparrow / \\
& & \quad inits = +\!\!+ /\!\!\!\!/\!\!- \circ [\cdot]* \\
& & \quad \ldots
\end{array}
$$

Inefficiency:

- Two many unnecessary intermediate data structures $\Rightarrow$ High communication cost
- Nested use of skeletons like $sum*$ $\Rightarrow$ Load unbalance

Optimization by Transformation

Rather than building a set of (ad-hoc) transformation rules for program fusion (Lengauer&Gorlatch&Wedler:99,Keller&Chakravarty:99)

$\Downarrow$

We propose *two cheap rules* for optimization:

Cheap Fusion Rule

Segmented Flattening Rule

(ESOP 2002)

A General Parallel Skeleton: Intermediate Form

Let g, p, q be functions, and $\oplus$ and $\otimes$ be associative operators.

$$
\begin{array}{lcl}
\mathsf{accumulate}\ [\,]\ e & = & g\ e \\
\mathsf{accumulate}\ (a : x)\ e & = & p(a, e)\ \oplus\ \mathsf{accumulate}\ x\ (e \otimes q\ a).
\end{array}
$$

We write $[\![g, (p, \oplus), (q, \otimes)]\!]$ for the function $\mathsf{accumulate}$.

Optimization by Fusion Transformation

- foldr-build Rule (Gill et al: ICFP'93)

  Assume that $gen : \forall\beta.\,(A \to \beta \to \beta) \to \beta \to \beta$ for some fixed A, then

$$\mathsf{foldr}\ (\oplus)\ e\ (\mathsf{build}\ gen) = gen\ (\oplus)\ e,$$

  where foldr and build are defined by

$$\begin{array}{lcl} \mathsf{foldr}\ (\oplus)\ e\ [] & = & e \\ \mathsf{foldr}\ (\oplus)\ e\ (a : x) & = & a \oplus \mathsf{foldr}\ (\oplus)\ e\ x \\ \mathsf{build}\ gen & = & gen\ (:)\ []. \end{array}$$

The accumulate: a consumer of join lists (POPL'98)

$$
\begin{array}{lcl}
[\![g, (p, \oplus), (q, \otimes)]\!]\ x & = & \mathit{fst} \circ \mathsf{accumulate}' \\
\mathsf{accumulate}'\ [\,] & = & \lambda e \rightarrow (g\ e, \_, \_) \\
\mathsf{accumulate}'\ [a] & = & \lambda e \rightarrow (p\ (a, e) \oplus g\ (e \otimes q\ a), p\ (a, e), q\ a) \\
\mathsf{accumulate}'\ (x +\!\!+\ y) & = & \mathsf{accumulate}'\ x \odot_{\oplus,\otimes} \mathsf{accumulate}'\ y
\end{array}
$$

where

$$
\begin{array}{lcl}
(u \odot_{\oplus,\otimes} v)\ e & = & \textbf{let}\ (r_1, s_1, t_1) = u\ e \\
 & & \qquad (r_2, s_2, t_2) = v\ (e \otimes t_1) \\
 & & \textbf{in}\ (s_1 \oplus r_2,\ s_1 \oplus s_2,\ t_1 \otimes t_2).
\end{array}
$$

Our Fusion Law

If for some fixed A we have

$$gen : \forall \beta. (\beta \to \beta \to \beta) \to (A \to \beta) \to \beta \to \beta$$

then

$$\begin{array}{l} [\![g, (p, \oplus), (q, \otimes)]\!] \ (\textsf{buildJ}\ gen)\ e \\ \quad = fst\ (gen\ (\odot_{\oplus,\otimes}) \\ \qquad\qquad (\lambda a \to (\lambda e \to (p\ (a, e) \oplus g\ (e \otimes q\ a), p\ (a, e), q\ a))) \\ \qquad\qquad (\lambda e \to (g\ e, \_, \_) \\ \qquad\qquad e) \end{array}$$

where buildJ $gen = gen\ (+\!\!+)\ [\cdot]\ []$.

Warm Fusion: Basic Skeletons in buildJ-accumulate Form

$$
\begin{aligned}
f * x &= \mathsf{buildJ}\ (\lambda c \lambda s \lambda n \rightarrow [\![\lambda \_ \rightarrow n, (\lambda(a, \_) \rightarrow s\ (f\ a), c), (\_, \_)]\!]\ x\ \_) \\
\oplus / x &= [\![\lambda \_ \rightarrow \iota_{\oplus}, (\lambda(a, \_) \rightarrow a, \oplus), (\_, \_)]\!]\ x\ \_ \\
\oplus \#_e x &= \mathsf{buildJ}\ (\lambda c \lambda s \lambda n \rightarrow [\![s, (\lambda(a, e) \rightarrow s\ e, c), (id, \oplus)]\!]\ x\ e)
\end{aligned}
$$

An Example

$$
\begin{aligned}
& \wedge / (even * x) \\
= & \quad \{ \text{ def. of } \mathit{reduce} \text{ and } \mathit{map} \} \\
& [\![\lambda\_ \to \mathsf{T}, (\lambda(a,\_) \to a, \wedge), (\_,\_)]\!] \\
& \quad (\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \to [\![\lambda\_ \to n, (\lambda(a,\_) \to s\ (even\ a), c), (\_,\_)]\!]\ x\ \_))\ \_ \\
= & \quad \{ \text{ Fusion } \} \\
& [\![\lambda\_ \to \mathsf{T}, (\lambda(a,\_) \to even\ a \wedge \mathsf{T}, \wedge), (\_,\_)]\!]\ x\ \_ \\
= & \quad \{ \text{ simplification } \} \\
& [\![\lambda\_ \to \mathsf{T}, (\lambda(a,\_) \to even\ a, \wedge), (\_,\_)]\!]\ x\ \_
\end{aligned}
$$

Segmented Flattening Transformation

$$sum * \;\; [[], [1], [2], [], [3,4,5,6]] \;\Rightarrow\; ?\; [(T\ 1, 1), (T\ 0, 2), (T\ 1, 3), (F, 4), (F, 5), (F, 6)]$$

The general question is under what condition of f, the program

$$f*$$

can be implemented efficiently? With fusion, f could be expected to be described in accumulate. So how to deal with

$$(\lambda x \rightarrow [\![k, (p, \oplus), (q, \otimes)]\!]\ x\ e)*?$$

Cheap Flattening Rule

$$(\lambda x.\, [\![g, (p, \oplus), (q, \otimes)]\!]\ x\ c) * xs \ = \ [\![g', (p', \oplus'), (q', \otimes')]\!](\mathit{flatten}\ xs)(F, c, \mathit{True}, 0)$$

where

$$
\begin{aligned}
& g'(\_, \_, \mathit{True}, i) = \mathit{copy}\ i\ (g\ c) \\
& g'(\_, a, \mathit{False}, i) = g\ a : \mathit{copy}\ i\ (g\ c) \\
& p'((T\ m, x), (\_, \_, \mathit{True}, \_)) = \mathit{copy}\ m\ (g\ c) +\!\!+ [p\ (x, c)] \\
& p'((T\ m, x), (\_, a, \mathit{False}, \_)) = [g\ a] +\!\!+ \mathit{copy}\ m\ (g\ c) +\!\!+ [p\ (x, c)] \\
& p'((F, x), (\_, a, \_, \_)) = [p\ (x, a)] \\
& us \oplus' vs = \mathit{init}\ us +\!\!+ [\mathit{last}\ us \oplus \mathit{head}\ vs] +\!\!+ \mathit{tail}\ vs \\
& q'(T\ m, x) = (T\ m, c \otimes q\ x, \mathit{False}, 0) \\
& q'(F, x) = (F, q\ x, \mathit{False}, 0) \\
& (\_, \_, \_, i) \otimes' (T\ m, b, v, \_) = (T\ m, b, v, i) \\
& (t, a, \_, i) \otimes' (F, b, v, \_) = (t, a \otimes b, v, i)
\end{aligned}
$$

An Experiment: XML Tag Matching Problem

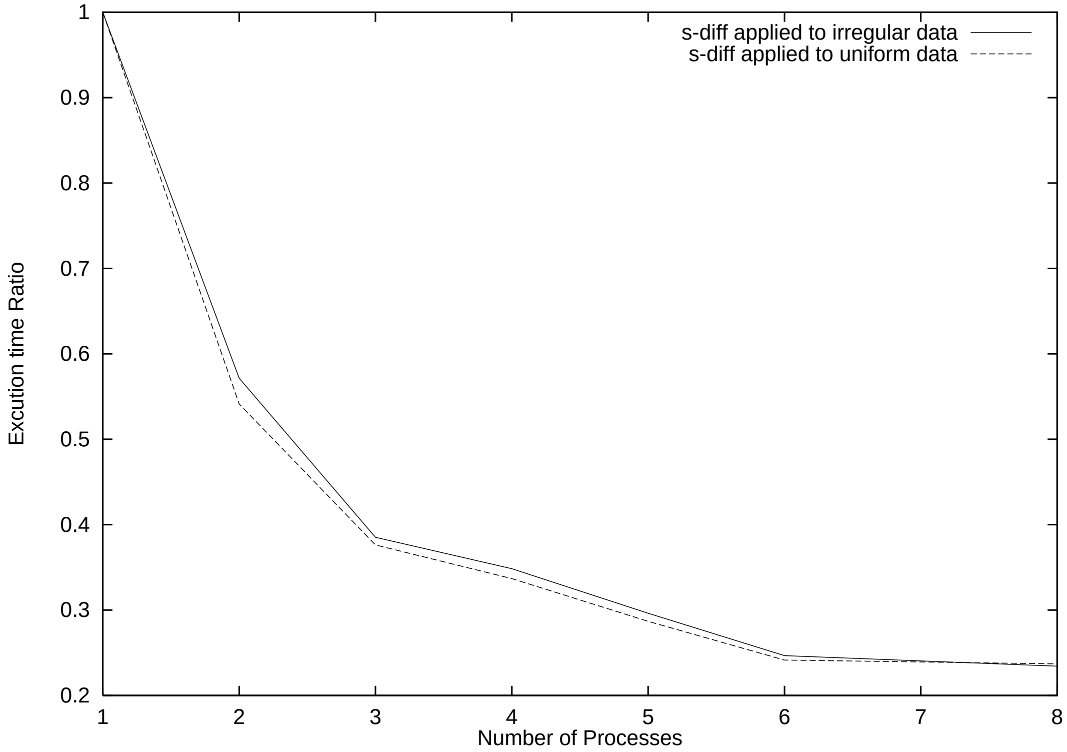
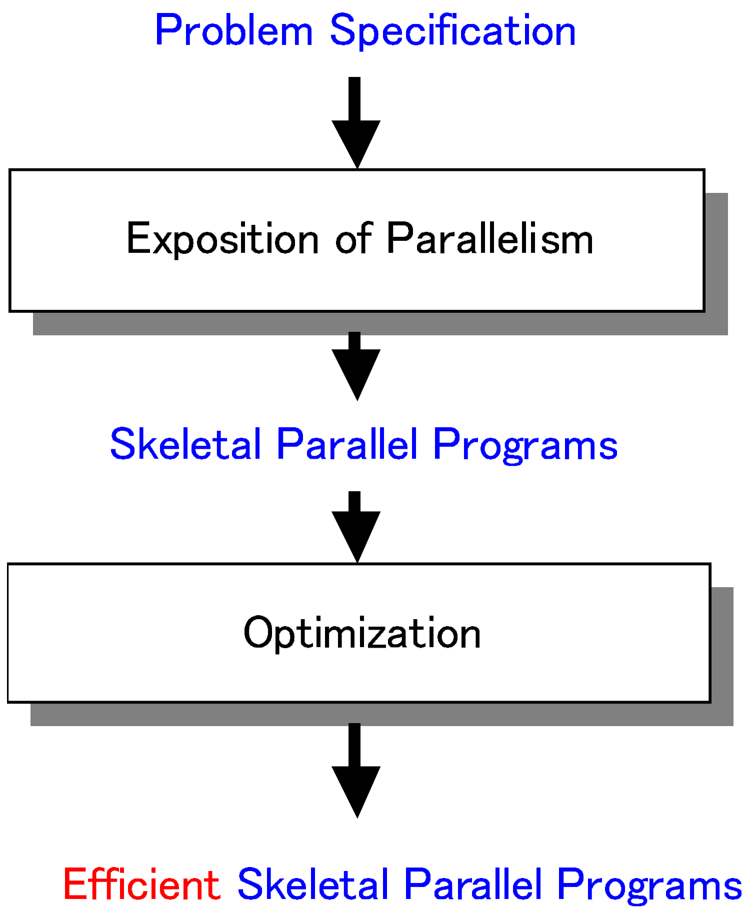


Figure 1: S-Diff applied to uniform and irregular data

Summary



Diffusion (Context Preservation), A General Skeleton, Cheap Fusion, Segmented Flattening

On-going and Future Work

- *Implementing* the idea: the optimizing system is under development!
- Extending the idea to develop skeletal parallel programs on *trees*

$\boxed{\text{Optimization of Polytypic Skeletal Parallel Programs}}$

`http://www.ipl.t.u-tokyo.ac.jp/~hu/`