Universität Ulm
Fakultät für Informatik

Transformational Development of Data-Parallel Algorithms: an Example

Helmuth Partsch
Universität Ulm

Nr. 94-18
Ulmer Informatik-Berichte
Dezember 1994

Transformational Development of Data-Parallel Algorithms: an Example

Helmuth Partsch
Fakultät für Informatik
Universität Ulm
89069 Ulm
email: partschh@informatik.uni-ulm.de

abstract

This paper deals with a case study in the formal derivation of data-parallel algorithms by means of program transformations. Particular emphasis is on the observation that a careful choice of suitable operations on abstract data structures and a thorough investigation of their algebraic properties can reduce substantial parts of the development activities to pure algebraic calculation.

0. Introduction

Transformational development summarizes the idea of deriving an algorithm from a formal problem specification by stepwise application of semantics-preserving transformation rules. For a brief introduction to and overview of the approach, cf., e.g., [Bauer et al. 89] or [Boiten et al. 92]. A comprehensive treatment of the subject can be found in [Partsch 90].

0.1 Transformational development

For the following, a slightly more precise characterization of the idea of transformational development can be given, if we assume the following scenario:

C set of language constructs
$\mathcal{L}(C)$: set of syntactically correct programs over C
C_s: "source constructs"
C_t: "target constructs"
C_m: "machine characteristics"
} (non-empty) subsets of C
$\mathcal{M}(p)$: semantics of $p \in \mathcal{L}(C)$

As a suitable interpretation of these sets, we could view C_s to consist of typical applicative constructs (inclusive of recursion) and C_t to comprise variables, assignment, and loops.

In the scenario as sketched above, a **formal specification** is the description of a concrete problem in terms of source constructs, i.e., as a $p \in \mathcal{L}(C_s)$.

Program transformations are (partial) mappings

$$\mathcal{T}_i: \mathcal{L}(C') \rightarrow \mathcal{L}(C'),$$

where $C', C'' \subseteq C$ and $\mathcal{M}(p) \equiv \mathcal{M}(\mathcal{T}_i(p))$ holds for all "admissible" $p \in \mathcal{L}(C')$, i.e., all those p for which $\mathcal{T}_i$ is defined. A typical example of a program transformation is given by the well-known transition from a tail-recursive function to an (equivalent) **while**-loop.

A **transformational development**, finally, is a mapping

$$\mathcal{T}: \mathcal{L}(C_s) \rightarrow \mathcal{L}(C_t \cup C_m) \text{ such that } \mathcal{M}(p) \equiv \mathcal{M}(\mathcal{T}(p)) \text{ holds for all "admissible" } p \in \mathcal{L}(C_s)$$

viewed as a stepwise process described by a composition of transformations.

By means of this fairly vague characterization it is already possible to give an account of relevant research topics within this approach. On the one hand, there are various linguistic aspects such as the identification of suitable source and target constructs C_s and C_t inclusive of their semantic definition, an aspect utterly important for the practical use of the methodology. On the other hand, there are, of course, the major problems of finding suitable transformations $\mathcal{T}_i$ and adequately representing them by appropriate rules, as well as the identification of generally applicable forms of compositions of transformation rules ("development strategies"). It is especially this latter aspect our case study will focus on.

0.2 Sequential and parallel algorithms

In case of sequential algorithms a distinction between the sets C_t and C_m in the above scenario is "irrelevant" resp. superfluous, since all sequential machines are characterized by essentially the same constructs. This is certainly not the case for parallel algorithms, where it is well neccessary to represent the different machine characteristics (of different classes of parallel architectures) by means of appropriate language constructs and to take this information into account in the development process.

In its widest sense, parallel execution means the "simultaneous" application of "operations" (by means of several processors) to (possibly additionally structured) collections of data. An important aspect in this context is the "processor distribution", i.e., the assignment of the available processors to the "operations" to be executed. A rough, well-known classification (for architectures as well as for algorithms targeted at these architectures) results from a more precise characterization of the term "operations". If it is required that the "operations" are identical and elementary, we get a class usually termed SIMD (single instruction multiple data). If different (and not just elementary) operationes are allowed, the class MIMD (multiple instruction multiple data) is obtained. In between there is a class sometimes called SPMD (single program multiple data) where all operations are required to be identical ("data parallelism"), but not neccesarily elementary. Within these classes a finer classification can be obtained, if the connections between the individual processors ("processor topology") are taken into account. Typical topologies for the SIMD or SPMD class are the (linear) processor array, the processor ring, or an arrangement of the processors in a mesh or a hypercube.

0.3 Formulation of parallel algorithms

In order to characterize specific hardware structures, it would, of course, be possible to develop new language constructs and to integrate them with established data and control structures into a new (parallel) target language. In terms of the scenario sketched in 0.1, this would mean a (new) definition of C_m independent of other available language constructs. The disadvantage of proceeding this way, however, is obvious: for each new construct, not only new semantics has to be defined (consistently with the semantics of existing constructs), but also, and even more important with respect to the development process, programming knowledge codified by corresponding transformation rules has to be explored, in order to be integrated with the existing formalism - in all, a laborious, error-prone, and, above all, rather inflexible approach.

Therefore, we follow an approach based on the idea of using available language constructs for the description of hardware characteristics, i.e., making C_m a set of definitions composed of constructs from $C_s \cup C_t$. Taking into account that processor topologies as well as data collections may be viewed as finite mappings on suitable index domains, the use of functional constructs is at hand. This leads in a natural way to the notion of "*skeletons*" (cf. [Harrison 92], [Darlington et al. 93], [Boiten et al. 93]): particular higher-order functions that abstract elementary operations of specific parallel architectures and allow a simple and straightforward transition to the corresponding machine commands. A typical example of such a skeleton is the *apply-to-all*, i.e., a higher-order function for the application of an operation to all elements of a data collection represented by a finite mapping, under the additional assumption that there is a unique correspondence between the index domain of the data collection and the available processors.

Aspects of processor topology also allow for an adequate representation by means of appropriate skeletons (cf., [Geerling x]). The important issue from an algorithmic point of view, however, is not the topology itself, but rather the "communication structure" induced by it, which appropriately can be abstracted into suitable skeletons. In this way, for example, the linear shift (of a data collection) is a typical communication skeleton for a linear processor array.

The advantages of using skeletons as abstractions of elementary parallel operations and communication aspects are at hand. By using available language constructs, viz. higher-order functions, it is to be expected that the effort for developing new methodical knowledge can be reduced to a minimum, since higher-order functions are already used in the development of sequential algorithms, and are, therefore, fairly extensively explored. Further (obvious) advantages are the following: Having an explicit goal (viz., the skeletons) for a development, supports a more clearly goal-directed development process. Relying on a common basis of description is a good starting point to investigate the portability of parallel algorithms for some class of architectures to another one. Integrating future architectures into the approach is straightforward and not a principal problem.

0.4 Problem-oriented skeletons

In earlier experiments (cf., e.g., [Geerling 92, 94a, 94b], [Partsch 93]) machine characteristic skeletons (based on finite mappings on index domains) were involved in the development process at a rather early stage. This caused some of the derivations to become unnecessarily burdened by extensive index calculations which, although straightforward in most cases, are

cumbersome to read, decrease clarity of the algorithms involved, and, above all, entail long and fatiguing derivations.

Meanwhile, recent research (cf., e.g., [Achatz, Schulte 94]) has shown that a careful choice of suitable operations on abstract data types ("problem-oriented skeletons"), a thorough investigation of their algebraic properties, and their use in the development process leads to much more elegant - since more abstract and thus more concise - and architecture-independent derivations.

Our subsequent sample derivation is essentially intended to back this claim. We deal once more with a problem that already has been tackled in [Partsch 93], viz. the odd-even-transposition sort algorithm. In our previous treatment indices were introduced fairly early in the development, which certainly obscured some of the subsequent development steps. In the present case study, we follow the idea of "problem-oriented skeletons", i.e., we use exlusively abstract operations on sequences and their algebraic properties in our development, and defer the transition to machine-oriented skeletons to the very last step in our derivation. In this way, the majority of the development becomes purely calculational und thus much more lucid. In order to convince himself of this latter claim, the reader is invited to compare the derivation in this paper with the one in [Partsch 93].

1. The problem specification

In order to be able to specify the problem we want to deal with, we have to introduce some basic notions, notably sequences.

1.1 Sequences

The primitive data type our problem is based on is the type of **sequences** (over some basic type α) which we abbreviate by **sequ**. As elementary, totally defined operations on sequences we assume to have available (for s, t of type **sequ**, x of type α):

ε	empty sequence
$\langle x \rangle$	sequence former
$s +\!\!+ t$	concatenation, and
$\lvert s \rvert$	length.

Whenever clear from the context, explicit sequence formers will be avoided, i.e., concatenation will also be used for adding (single) elements to a sequence.

In addition to the totally defined operations, we also assume to have available some partially defined ones:

s_i	indexed access (only defined for $1 \le i \le \lvert s \rvert$, otherwise undefined)
s_1	first element (undefined for $s \equiv \varepsilon$)
$s_\#$	last element (undefined for $s \equiv \varepsilon$)
$\overleftarrow{s}$	rest (undefined for $s \equiv \varepsilon$)
$\overrightarrow{s}$	starter (undefined for $s \equiv \varepsilon$).

For all operations, characteristic properties such as $s \neq \varepsilon \Rightarrow s \equiv s_1 \mathbin{+\!\!+} \lceil s$, usually explicitly given by an algebraic type definition (cf. [Partsch 90]), are also assumed to be available.

1.2 Sorting

Based on the notion of sequences, a specification of the sorting problem is straightforward:

sort(*S*) **where**

sort: **sequ** → **sequ**,
sort(*s*) =$_{\text{def}}$ **some sequ** *x*: *sameels*(*x*, *s*) ∧ *issorted*(*x*),

sameels: (**sequ** × **sequ**) → **bool**,
sameels(*x*, *s*) =$_{\text{def}}$ ‹‹ *x* and *s* have the same elements ››,

issorted: **sequ** → **bool**,
issorted(*s*) =$_{\text{def}}$ ∀ (**inat** *i*, *j*): $i < j \Rightarrow s[i] \leq s[j]$,

inat =$_{\text{def}}$ (**nat** *i*: $1 \leq i \leq |S|$).

Above, the function *sameels* has not been formalized. This was done on purpose, since for our derivation we only need intuitively obvious properties of this function.

2. Development of an algorithm

In the following an algorithm for the sorting problem will be developed. As to the rules used in the development, the reader is referred to [Partsch 90].

2.1 Preparatory considerations

Our first goal is to transform the specification in order to have a better starting point which clearly motivates the central idea finally leading to the intended algorithm. Starting with the definition of the predicate *issorted*, we can simply reason as follows:

∀ (**inat** *i*, *j*): $i < j \Rightarrow s[i] \leq s[j]$
  ≡ [relationship between ∀ and ∃]
¬ ∃ (**inat** *i*, *j*): $\neg(i < j \Rightarrow s[i] \leq s[j])$
  ≡ [def. of ⇒ in terms of ∨; de Morgan's law]
¬ ∃ (**inat** *i*, *j*): $i < j \wedge s[i] > s[j]$
  ≡ [relationship between ∃ and set comprehension]
$\{$(**inat** *i*, *j*): $i < j \wedge s[i] > s[j]\} = \emptyset$
  ≡ [trivial property of sets]
$|\{$(**inat** *i*, *j*): $i < j \wedge s[i] > s[j]\}| = 0$.

Thus, by abstraction, we have as a new specification of the predicate *issorted*:

$issorted(s) =_{def} \#invs(s) = 0,$

$\#invs$: **sequ** → **nat**,
$\#invs(s) =_{def} |\{(\mathbf{inat}\ i, j): i < j \wedge s[i] > s[j]\}|.$

Our next efforts aim at deriving an algorithm from this modified specification. Obviously, we have

$$\sum_{i=1}^{|s|-1} i = (|s| \times (|s|-1))/2$$

as an upper bound for $\#invs(s)$, and, by rearranging the elements of s, we want to achieve that $\#invs(s) = 0$ becomes true. This motivates the basic idea for an algorithm by successively reducing the upper bound on the number of inversions (until it becomes ≤ 0) by means of suitable operations. Naively, we could eliminate just one inversion per step by swapping mispositioned elements. This, however, would lead to an algorithm with worst case complexity $O(|s|^2)$. Therefore, as an additional idea to obtain an algorithm with better performance, we try to find an operation that reduces the upper bound for $\#invs(s)$ by $|s|-1$ in each step. Obviously, this has to be an operation that changes the sequence as a whole, e.g., by multiple swaps.

2.2 Development

As motivated in the previous section, we start our development by adding the upper bound on $\#invs(s)$ as an additional parameter to the function sort. Technically, this means, we use an *embedding* (with assertion) which results in

$sort'(S, |S|)$ **where**

$sort'$: (**sequ** × **int**) → **sequ**,
$sort'(s, i) =_{def} sort(s)$

with

$\#invs(s) \leq (i \times (|s|-1))/2$

as the assertion (invariant) for *sort'*.

The following steps follow the "generalized unfold-fold strategy" from [Partsch 90] applied to the definition of *sort'*. By unfolding the definition of *sort*, applying a *case introduction* (based on the tautology $i \leq 0 \vee i > 0$), and exploiting the *distributivity* of **some** over **if - fi**, we get

$sort'(s, i) =_{def}$
if $i \leq 0$ **then some sequ** x: $sameels(x, s) \wedge issorted(x)$
else some sequ x: $sameels(x, s) \wedge issorted(x)$ **fi**.

Next, both branches of the conditional are simplified exploiting the condition $i \leq 0$ and the assertion of *sort'*.

In the **then**-branch (under premise $\#invs(s) \leq (i \times (|s|-1))/2 \wedge i \leq 0$) we simply have:

$\#invs(s) \leq (i \times (|s|-1))/2 \wedge i \leq 0) \Rightarrow \#invs(s) \leq 0 \Rightarrow issorted(s)$; and
$sameels(s, s) \equiv$ **true**,

i.e., s satisfies the required properties.

In the **else**-branch (under premise $\#invs(s) \leq (i \times (|s|-1))/2 \wedge i > 0$) we assume to be able to find an operation

$transp$: **sequ** $\rightarrow$ **sequ**,

with the properties (necessary to maintain the assertion of *sort'* and to allow a subsequent folding)

$s \equiv (transp(S))^{(|s|-i)/2} \Rightarrow (\#invs(transp(s)) \leq ((i\text{-}2) \times (|s|-1))/2$; and
$sameels(transp(s), s) \equiv$ **true**.

Together, we thus obtain:

$sort'(s, i) =_{def}$
if $i \leq 0$ **then** s
else some sequ x: $sameels(x, transp(s)) \wedge issorted(x)$ **fi**,

and folding with the original definition of *sort'* results in

$sort'(s, i) =_{def}$ **if** $i \leq 0$ **then** s **else** $sort'(transp(s), i\text{-}2)$ **fi**.

2.3 Additional operations

Of course, we still have to justify the essential assumption in our development by providing a definition of *transp* and proving the postulated properties. To this end, we first introduce some additional operations on sequences:

$\langle s_i: P(i)\rangle$	sequence comprehension
$o(s) =_{def} \langle s_i:$ **odd** $i\rangle$	subsequence of elements with odd index
$e(s) =_{def} \langle s_i:$ **even** $i\rangle$	subsequence of elements with even index
$split(s) =_{def} (o(s), e(s))$	split of s into the pair of subsequences with odd and even indices, resp.
$ilv(s, t)$	inverse of *split* , i.e. $ilv(split(s)) \equiv s$ and $split(ilv(s, t)) \equiv (s, t)$ hold (for **abs**$(\|s\| - \|t\|) \leq 1$)

$s \circledleft t =_{def} (\langle min(s_i, t_i): 1 \leq i \leq |s|\rangle, \langle max(s_i, t_i): 1 \leq i \leq |s|\rangle)$ (provided $|s| = |t|$)

$s \circledright t =_{def} (\langle max(s_i, t_i): 1 \leq i \leq |s|\rangle, \langle min(s_i, t_i): 1 \leq i \leq |s|\rangle)$ (provided $|s| = |t|$)

$\uparrow s =_{def}$ **max**$\{s_i: 1 \leq i \leq |s|\}$ (provided $|s| > 0$)

$\downarrow s =_{def}$ **min**$\{s_i: 1 \leq i \leq |s|\}$ (provided $|s| > 0$)

Among these operations, we are primarily interested in *split*, *ilv*, $\circledleft$ and $\circledright$.These operations have a number of interesting algebraic properties which will be used in our further derivations. Some of these properties are listed below where, for brevity, the definedness of all subexpressions is assumed without explixitly stating the respective conditions, in particular $|s| = |t|$. In addition, in order to save parentheses, we also assume $\circledleft$ to have lower priority than $\dashv\vdash$. Thus, for **sequ** s, t, u, v, and elements a, b we have:

(a) $o(ilv(s, t)) \equiv s$, $\qquad e(ilv(s, t)) \equiv t$

(b) $o(\overrightarrow{ilv(s, t)}) \equiv s, \quad e(\overrightarrow{ilv(s, t)}) \equiv \vec{t}, \quad e(\overleftarrow{ilv(s, t)}) \equiv t, \quad o(\overleftarrow{ilv(s, t)}) \equiv \overleftarrow{s}$

(c) $a \leq b \Rightarrow a \dashv\vdash b \equiv ilv(a \oslash b)$

(d) $ilv(s \oslash t) \dashv\vdash ilv(u \oslash v) \equiv ilv(s \dashv\vdash u \oslash t \dashv\vdash v)$

(e) $a \dashv\vdash ilv(s, t) \dashv\vdash b \equiv ilv(a \dashv\vdash t, s \dashv\vdash b)$

(f) $o(ilv(s \oslash t)) \equiv e(ilv(t \oslash s)), \qquad e(ilv(s \oslash t)) \equiv o(ilv(t \oslash s))$

(g) $a \geq b \Rightarrow a \dashv\vdash o(ilv(s \oslash t)) = o(ilv(a \dashv\vdash s \oslash b \dashv\vdash t))$

(h) $e(ilv(s \oslash t)) \equiv e(ilv(a \dashv\vdash s \oslash b \dashv\vdash t))$

(i) **odd** $|s| \Rightarrow o(s) \equiv o(\vec{s}) \dashv\vdash s_{\#}, \qquad$ **even** $|s| \Rightarrow o(s) \equiv o(\vec{s})$

(k) **odd** $|s| \Rightarrow e(s) \equiv e(\vec{s}), \qquad$ **even** $|s| \Rightarrow e(s) \equiv e(\vec{s}) \dashv\vdash s_{\#}$

The proofs of these properties are fairly straightforward and left to the interested reader.

2.4 The odd-even-transposition sort algorithm

Based on the operations introduced above, we are now able to complete the definition of our sorting algorithm (for a proof that the definition of *transp* satisfies the postulated conditions, cf. [Völker 92]):

sort'(*S*, |*S*|) **where**

sort': (**sequ** × **int**) → **sequ**,
sort'(*s*, *i*) =$_{\text{def}}$ **if** $i \leq 0$ **then** *s* **else** *sort'*(*transp*(*s*), *i*-2) **fi**,

transp: **sequ** → **sequ** ,
transp(*s*) =$_{\text{def}}$ *transpe*(*transpo*(*s*)),

transpo: **sequ** → **sequ** ,

$$transpo(s) =_{\text{def}} \begin{cases} ilv(o(s) \oslash e(s)), \text{ if } \textbf{even } |s| \\ ilv(o(\vec{s}) \oslash e(s)) \dashv\vdash s_{\#}, \text{ if } \textbf{odd } |s| \end{cases}$$

transpe: **sequ** → **sequ** ,

$$transpe(s) =_{\text{def}} \begin{cases} s_1 \dashv\vdash ilv(\overrightarrow{e(s)} \oslash \overleftarrow{o(s)}) \dashv\vdash s_{\#}, \text{ if } \textbf{even } |s| \\ s_1 \dashv\vdash ilv(e(s) \oslash \overleftarrow{o(s)}), \text{ if } \textbf{odd } |s| \end{cases}$$

This algorithm works as follows. Assume as input, the sequence

<3, 7, 5, 2, 1, 4, 6 >.

Then the algorithm performs the following computation:

3		3		3		2		2		1		1
7		7		2		3		1		2		2
5	*transpo*	2	*transpe*	7	*transpo*	1	*transpe*	3	*transpo*	3	*transpe*	3
2	→	5	→	1	→	7	→	4	→	4	→	4
1		1		5		4		7		5		5
4		4		4		5		5		7		6
6		6		6		6		6		6		7

Obviously, if we assume an operational realization of the operation $\lessdot$, which (by means of sufficiently many processors) is able to perform the necessary comparisons betwen and interchanges of sequence elements "simultaneously", we already have an algoithm which is parallel in principle. A key issue in this respect, from a methodical point of view, is the fact that an algorithm has been derived from the specification which uses powerful operations that change the data structure as a whole, but are elementwise executable.

3. Improvement of the algorithm for parallel execution

The algorithm given in the previous section, however, still has some unwanted peculiarities that should be eliminated: On the one hand there are case distinctions involving scalar operations (viz. $+\!\!+$) in the definitions of *transpo* and *transpe* which should be avoided for parallel execution. On the other hand, we immediately see that in each iteration only part of the processors are active which is a waste of resources. In both cases, we have phenomina that are important for parallel algorithms in general: Avoiding scalar operations by means of "uniformization" of data and optimal exploitation of the available processors. In both cases it is also possible to obtain the desired form by appliction of suitable transformation rules which essentially exploit the algebraic properties of the operations used.

3.1 Elimination of scalar operations

As a first step towards eliminating the scalar operations in *transpo*/*transpe* we try to eliminate the case distinctions. To this end, we introduce auxiliary operations (essentially to obtain sequences of uniform, even length):

$$s^* =_{\text{def}} \begin{cases} s +\!\!+ \uparrow s, & \text{if } \mathbf{odd}\ |s| \\ s +\!\!+ \uparrow s +\!\!+ \uparrow s, & \text{if } \mathbf{even}\ |s| \end{cases}$$

$$\mathit{unstar}(s^*, |s|) =_{\text{def}} \begin{cases} \overline{s^*}, & \text{if } \mathbf{odd}\ |s| \\ \overline{\overline{s^*}}, & \text{if } \mathbf{even}\ |s| \end{cases}$$

Again, these operations have interesting algebraic properties that profitably can be used later in our development. Some of these properties (the proofs of which are again straightforward and left to the reader) are (for arbitrary **sequ** s):

- **even** $|s^*|$
- **odd** $|s| \Rightarrow s \equiv \overline{s^*}$
- **even** $|s| \Rightarrow s \equiv \overline{\overline{s^*}}$
- $(s^*)_{\#} \equiv \uparrow s$
- $|s| > 0 \Rightarrow (s^*)_1 \equiv s_1 \equiv (o(s^*))_1$
- $e(s^*) \equiv e(s) +\!\!+ \uparrow s$
- **even** $|s| \Rightarrow o(s^*) \equiv o(s) +\!\!+ \uparrow s$
- **odd** $|s| \Rightarrow o(s^*) \equiv o(s)$

- **odd** |s| $\Rightarrow$ $\overline{e(s^*)} \equiv e(s)$
- **odd** |s| $\Rightarrow$ $\overline{o(s^*)} \equiv \overline{o(s)}$.

Our next efforts aim at "adapting" *sort'* to these "uniform" sequences. By an *embedding* we define

sort"(*s**, |*S*|) **where**

sort": (**sequ** × **int**) → **sequ**,
sort"(*s**, *i*) $=_{\text{def}}$ *unstar*(*sort'*(*s*, *i*))*),

and then calculate as follows:

unstar(*sort'*(*s*, *i*))*)
≡ [def. of *sort'* ; distr. of * over **if** - **fi**]
unstar(**if** *i* ≤ 0 **then** *s** **else** *sort'*(*transp*(*s*), *i*-2)* **fi**)
≡ [distr. of *unstar* over **if** - **fi**]
if *i* ≤ 0 **then** *unstar*(*s**) **else** *unstar*(*sort'*(*transp*(*s*), *i*-2))*) **fi**
≡ [def. of *sort"*; def. of *transp*]
if *i* ≤ 0 **then** *unstar*(*s**) **else** *sort"*((*transpe*(*transpo*(*s*)))*, *i*-2) **fi**
≡ [new def. see below]
if *i* ≤ 0 **then** *unstar*(*s**) **else** *sort"*(*transpe**(*transpo**(*s**)), *i*-2) **fi**
where
*transpo**(*s**) $=_{\text{def}}$ (*transpo*(*s*))*
*transpe**(*s**) $=_{\text{def}}$ (*transpe*(*s*))*.

Similar to the above calculation, explicit defintions for *transpo** and *transpe** can be obtained by further calculation:

For the case **odd** |s| we have

*transpo**(*s**)
≡ [def. of *transpo**]
(*transpo*(*s*))*
≡ [def. of *transpo*, *]
ilv(*o*($\vec{s}$) ⋉ *e*(*s*)) ⫫ $s_{\#}$ ⫫ ↑*s*
≡ [(c)]
ilv(*o*($\vec{s}$) ⋉ *e*(*s*)) ⫫ (*s*# ⋉ ↑*s*)
≡ [(d)]
ilv((*o*($\vec{s}$) ⫫ *s*#) ⋉ (*e*(*s*) ⫫ ↑*s*))
≡ [def. of $\vec{s}$, *]
ilv(*o*(*s**) ⋉ *e*(*s**))

and

$transpe^*(s^*)$

$\equiv$ [def. of $transpe^*$]

$(transpe(s))^*$

$\equiv$ [def. of $transpe$, *]

$s_1 \dashv\!\vdash ilv(e(s) \oslash \overline{o(s)}) \dashv\!\vdash \uparrow s$

$\equiv$ [properties of $\vec{s}$ and *]

$s_1 \dashv\!\vdash ilv(\overline{e(s^*)} \oslash \overline{o(s^*)}) \dashv\!\vdash \uparrow s$

Likewise, for the case **even** |s| we have

$transpo^*(s^*)$

$\equiv$ [def. of $transpo^*$]

$(transpo(s))^*$

$\equiv$ [def. of $transpo$, *]

$ilv(o(s) \oslash e(s)) \dashv\!\vdash \uparrow s \dashv\!\vdash \uparrow s$

$\equiv$ [(c)]

$ilv(o(s) \oslash e(s)) \dashv\!\vdash (\uparrow s \oslash \uparrow s)$

$\equiv$ [(d)]

$ilv((o(s) \dashv\!\vdash \uparrow s) \oslash (e(s) \dashv\!\vdash \uparrow s))$

$\equiv$ [def. of *]

$ilv(o(s^*) \oslash e(s^*))$

and

$transpe^*(s^*)$

$\equiv$ [def. of $transpe^*$]

$(transpe(s))^*$

$\equiv$ [def. of $transpe$, *]

$s_1 \dashv\!\vdash ilv(\overline{e(s)} \oslash \overline{o(s)}) \dashv\!\vdash s_\# \dashv\!\vdash \uparrow s \dashv\!\vdash \uparrow s$

$\equiv$ [(c)]

$s_1 \dashv\!\vdash ilv(\overline{e(s)} \oslash \overline{o(s)}) \dashv\!\vdash (s_\# \oslash \uparrow s) \dashv\!\vdash \uparrow s$

$\equiv$ [(d)]

$s_1 \dashv\!\vdash ilv((\overline{e(s)} \dashv\!\vdash s_\#) \oslash (\overline{o(s)} \dashv\!\vdash \uparrow s)) \dashv\!\vdash \uparrow s$

$\equiv$ [properties of $\vec{s}$ and *]

$s_1 \dashv\!\vdash ilv(\overline{e(s^*)} \oslash \overline{o(s^*)}) \dashv\!\vdash \uparrow s.$

Putting these pieces together, we get as an intermediate result

$transpo^*(s^*) =_{def} ilv(o(s^*) \oslash e(s^*))$

$transpe^*(s^*) =_{def} s_1 \dashv\!\vdash (ilv(\overline{e(s^*)} \oslash \overline{o(s^*)}) \dashv\!\vdash \uparrow s.$

Still, however, there are scalar operations in the definition of *transpe**. Therefore, we calculate further as follows:

$s_1 \dashv\!\vdash ilv(\overline{e(s^*)} \circledcirc \overline{o(s^*)}) \dashv\!\vdash \uparrow s$

$\equiv$ [def. of *ilv*]

$s_1 \dashv\!\vdash ilv(o(ilv(\overline{e(s^*)} \circledcirc \overline{o(s^*)}), e(ilv(\overline{e(s^*)} \circledcirc \overline{o(s^*)}))) \dashv\!\vdash \uparrow s$

$\equiv$ [(f)]

$s_1 \dashv\!\vdash ilv(e(ilv(\overline{o(s^*)} \oslash \overline{e(s^*)}), o(ilv(\overline{o(s^*)} \oslash \overline{e(s^*)}))) \dashv\!\vdash \uparrow s$

$\equiv$ [(e)]

$ilv(s_1 \dashv\!\vdash o(ilv(\overline{o(s^*)} \oslash \overline{e(s^*)}), e(ilv(\overline{o(s^*)} \oslash \overline{e(s^*)})) \dashv\!\vdash \uparrow s)$

$\equiv$ [(g), (h)]

$ilv(o(ilv(s_1 \dashv\!\vdash \overline{o(s^*)} \oslash \downarrow s \dashv\!\vdash \overline{e(s^*)}), \overline{e(ilv(s_1 \dashv\!\vdash \overline{o(s^*)} \oslash \downarrow s \dashv\!\vdash \overline{e(s^*)}))} \dashv\!\vdash \uparrow s)$

$\equiv$ [def. of $\overline{s}$]

$ilv(o(ilv(o(s^*) \oslash \downarrow s \dashv\!\vdash \overline{e(s^*)}), \overline{e(ilv(o(s^*) \oslash \downarrow s \dashv\!\vdash \overline{e(s^*)}))} \dashv\!\vdash \uparrow s)$

$\equiv$ [**where**-abstraction]

$ilv(o(ilv(o', e'), \overline{e(ilv(o', e'))} \dashv\!\vdash \uparrow s)$ **where** $(o', e') =_{\text{def}} o(s^*) \oslash (\downarrow s \dashv\!\vdash \overline{e(s^*)})$

$\equiv$ [(a)]

$ilv(o', \overline{e'} \dashv\!\vdash \uparrow s)$ **where** $(o', e') =_{\text{def}} o(s^*) \oslash (\downarrow s \dashv\!\vdash \overline{e(s^*)})$

3.2 Optimal exploitation of available processors

Although successful in eliminating scalar operations, we still have the problem that in each step of the computation processors with even (resp. odd) index are inactive. The basic idea to solve this problem is to have two separate sequences for even and odd indices and to appropriately modify the sorting algorithm.

Again by an *embedding*, we transform our previous problem statement into

*sort**(*split*(*S**), |*S*|, |*S*|) **where**

*sort**: (**sequ** × **sequ** × **int**) → **sequ**,
*sort**(*o*, *e*, *i*) $=_{\text{def}}$ *sort"*(*ilv*(*o*, *e*), *i*)

and calculate an explicit definition for *sort** as follows:

sort"(*ilv*(*o*, *e*), *i*)

$\equiv$ [def. of *sort"*]

if $i \leq 0$ **then** *unstar*(*ilv*(*o*, *e*)) **else** *sort"*(*transpe**(*transpo**(*ilv*(*o*, *e*))), *i*-2) **fi**

$\equiv$ [define *transp** by *ilv*(*transp**(*o*, *e*)) $=_{\text{def}}$ *transpe**(*transpo**(*ilv*(*o*, *e*)))]

if $i \leq 0$ **then** *unstar*(*ilv*(*o*, *e*)) **else** *sort"*(*ilv*(*transp**(*o*, *e*)), *i*-2) **fi**

$\equiv$ [def. of *sort**]

if $i \leq 0$ **then** *unstar*(*ilv*(*o*, *e*)) **else** *sort**(*transp**(*o*, *e*), *i*-2) **fi**.

As in the previous section, an explicit definition of *transp** is obtained by further, straightforward calculation:

*transp**(o, e)

≡ [property of *split/ilv*]

split(*ilv*(*transp**(o, e)))

≡ [defining property of *transp**(o, e)]

split(*transpe**(*transpo**(*ilv*(o, e))))

≡ [def. of *transpo**]

split(*transpe**(*ilv*(o', e'))) **where** $(o', e') =_{\text{def}} o(ilv(o, e)) \circledcirc e(ilv(o, e)) = o \circledcirc e$

≡ [def. of *transpe**; simplification]

split(*ilv*$(o'', \overline{e''} \shortparallel \uparrow s)$ **where** $(o'', e'') =_{\text{def}} o' \circledcirc (\downarrow s \shortparallel \overline{e'})$; $(o', e') =_{\text{def}} o \circledcirc e$

≡ [property of *split/ilv*]

o'', $\overline{e''} \shortparallel \uparrow s$ **where** $(o'', e'') =_{\text{def}} o' \circledcirc (\downarrow s \shortparallel \overline{e'})$; $(o', e') =_{\text{def}} o \circledcirc e$

3.3 Final algorithm

Putting all pieces together, we have as the final result of our derivation:

*sort**(*split*(*S**), |*S*|, |*S*|) **where**

*sort**: (**sequ** × **sequ** × **int**) → **sequ**,

*sort**$(o, e, i) =_{\text{def}}$

if $i \leq 0$ **then** *unstar*(*ilv*(o, e)) **else** *sort**$(o'', \overline{e''} \shortparallel \uparrow s\ ,\ i\text{-}2)$

where $(o'', e'') =_{\text{def}} o' \circledcirc (\downarrow s \shortparallel \overline{e'})$; $(o', e') =_{\text{def}} o \circledcirc e$ **fi**.

In order to illustrate how this algorithm works, we consider again the sequence

<3, 7, 5, 2, 1, 4, 6 >.

Now, the computation proceeds as follows (with SHR as abbreviation for $(\downarrow s \shortparallel \overline{e})$ and SHL for $\overline{e} \shortparallel \uparrow s$):

$i = 7$

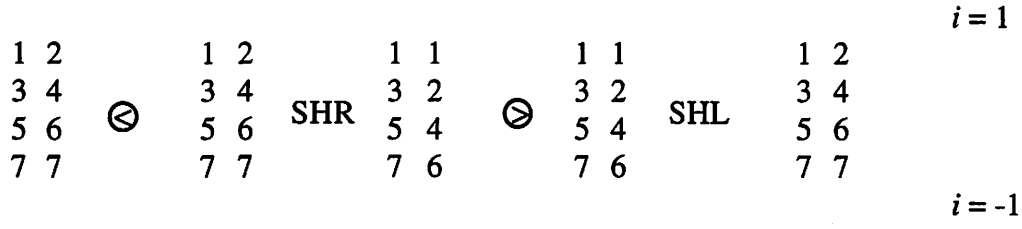
3 7 5 2 1 4 6 7	⊚	3 7 2 5 1 4 6 7	SHR	3 1 2 7 1 5 6 4	⊚	3 1 7 2 5 1 6 4	SHL	3 2 7 1 5 4 6 7

$i = 5$

3 2 7 1 5 4 6 7	⊚	2 3 1 7 4 5 6 7	SHR	2 1 1 3 4 7 6 5	⊚	2 1 3 1 7 4 6 5	SHL	2 1 3 4 7 5 6 7

$i = 3$

2 1 3 4 7 5 6 7	⊚	1 2 3 4 5 7 6 7	SHR	1 1 3 2 5 4 6 7	⊚	1 1 3 2 5 4 7 6	SHL	1 2 3 4 5 6 7 7



Hence, our algorithm results in

$unstar(ilv(<1, 3, 5, 7>, <2, 4, 6, 7>, 7) = <1, 2, 3, 4, 5, 6, 7>$

as expected.

3.4 Final remarks on the development and the algorithm

The algorithm as given in the previous section still admits a sequential interpretation. There is, however, also a straightforward interpretation as a parallel algorithm for a linear processor array, if we interprete

sequ	by	**vector**
$(\downarrow s \dashv\!\!\!+ \vec{e})$	by	$SHR(e, \downarrow s)$
$\overleftarrow{e} \dashv\!\!\!+ \uparrow s$	by	$SHL(e, \uparrow s)$
$o \circledcirc e$	by	$MAP_{2\text{-}2}(<, o, e)$
$o \circledcirc e$	by	$MAP_{2\text{-}2}(>, o, e)$.

Here, $SHR(e, \downarrow s)$ and $SHL(e, \uparrow s)$ denote machine-oriented skeletons describing a linear shift (by one position) to the left and right, respectively, where the empty position is filled with $\downarrow s$ and $\uparrow s$, respectively. Also, $MAP_{2\text{-}2}(op, v, v')$ denotes denotes a skeleton which formalizes the elementwise application of a binary operation *op* with two results to the elements of v and v', respectively. For formal definitions of these skeletons, we refer the reader to [Partsch 93].

Of course, some further, rather straightforward optimizations are possible, too. For instance, the case $|s| = 0$ could be treated separately, already in the definition of *sort**. Also, $\uparrow s$ and $\downarrow s$ could be computed in advance (with complexity $O(|s|)$) and replaced in *sort** by corresponding global constants.

4. Concluding remarks

In comparing the derivation in the previous sections with the one in [Partsch 93], it can be seen that both are of approximately the same length. The major difference, however, is not in the length of the derivations, but rather in their complexity. Whereas in our previous attempt to tackle the same problem we had to use a number of inventive steps additionally burdened by a too early introduction of index calculations, the present one consists mainly of pure algebraic calculations which benefit from a suitable choice of high-level operators and an investigation of their algebraic properties. In addition, several of these calculations (e.g. those where the defintion of the *sort* function is adapted to a new domain) follow similar, fairly straightforward patterns such that a compactification by suitable abstraction would be reasonable and, moreover, substantially reduce the length of the derivations.

The key step in the approach illustrated by our sample derivation is the one from the original problem statement into a form which allows the parallel execution of suitably structured collections of data. In our sample development this was not problem at all, since already the original problem statement contained such a collection of data, viz. sequences. Other case studies (cf., e.g., [Partsch 93], [Geerling 92, 94a, 94b], [Boiten et al. 93]) where this was not the case, have shown that well-know program transformations (such as *currying* or *tabulation*, cf. [Partsch 90]) are suited to achieve this important transition.

From the considerations in this paper, one might get the (wrong) impression that all problems with respect to the development of parallel algorithms are already solved, and that not much still needs to be investigated. Of course, this is not the case. Even if the central problem of characterizing different classes of parallel architectures is solved by the definition of suitable sets of skleletons, there is still a major task to be solved, viz. the exploration and acquisition of methodical knowledge necessary for the derivation of parallel algorithms from formal specifications in a much more goal-directed fashion. Another open problem is the aspect of "data partitioning" which we ignored in our sample development. This aspect deals with a suitable distribution of the data to be processed to the available processors, in case the size of the data collection exceeds the number of available processors. First attempts towards solving this problem in a transformational setting can be found in [Pepper et al. 93] and [Südholt 94].

References

[Achatz, Schulte 94]
Achatz, K., Schulte, W.: Architecture independent massive parallelization of divide-and-conquer algorithms. Submitted for publication, 1994

[Bauer et al. 89]
Bauer, F.L., Möller, B., Partsch, H., Pepper, P.: Programming by formal reasoning - computer-aided intuition-guided programming. IEEE Transactions on Software Engineering **15**:2, 165-180 (1989)

[Boiten et al. 92]
Boiten, E.A., Partsch, H.A., Tuijnman, D., Völker, N.: How to produce correct software - an introduction to formal specification and program development by transformations. The Computer Journal **35**:6, 547-554 (1992)

[Boiten et al. 93]
Boiten, E.A., Geerling, A.M., Partsch, H.A.: Transformational derivation of (parallel) programs using skeletons. In: Wijshoff, H.A.G. (ed.): Computing Science in the Netherlands 1993, pp. 97-108

[Darlington et al. 93]
Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q., While, R.L.: Parallel programming using skeleton functions. In: PARLE '93: Parallel Architectures and Languages Europe. Lecture Notes in Computer Science **694**, Berlin: Springer 1993, pp. 146-160

[Boiten et al. 93]
Boiten, E.A., Geerling, A.M., Partsch, H.A.: Transformational derivation of (parallel) programs using skeletons. In: Wijshoff, H.A.G. (ed.): Computing Science in the Netherlands 1993, pp. 97-108

[Geerling 92]
Geerling, A.M.: Two examples in parallel program derivation: Parallel-prefix and matrix multiplication. Imperial College London, Research Report DoC 92/33, 1992

[Geerling 94a]
Geerling, A.M.: Formal derivation of SIMD parallelism from non-linear recursive specifications. In: Buchberger, B., Volkert, J. (eds.): CONPAR '94 - VAPP VI International Conference on Parallel and Vector Processing. Lecture Notes in Computer Science **854**, Berlin: Springer 1994, pp.136-147

[Geerling 94b]
Geerling, A.M.: Program transformations and skeletons: formal derivation of parallel programs. Computing Science Institute, University of Nijmegen, Technical Report CSI-R9411, 1994

[Harrison 92]
Harrison, P.G.: A higher-order approch to parallel algorithms. The Computer Journal 29:12, 555-566 (1992)

[Partsch 93]
Partsch, H.: Some experiments in transforming towards parallel executability. In: Paige, R., Reif, J., Wachter, R. (eds.): Parallel algorithm derivation and program transformation. Kluwer Academic Publishers 1993, pp.

[Pepper et al. 93]
Pepper, P., Exner, J., Südholt, M.: Functional development of massively parallel programs. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.): Formal methods in programming and their applications. Lecture Notes in Computer Science **735**, Berlin: Springer 1990, pp.217-238

[Südholt 94]
Südholt, M.: Data distribution algebras - a formal basis for using skeletons. In: Olderog, E.-R. (ed.): Programming concepts, methods and calculi. Amsterdam: North-Holland 1994, pp. 19-38

[Völker 92]
Völker, N.: A formal derivation of odd-even transposition sort. Department of Informatics, University of Nijmegen, Technical Report 92-07, 1992

Liste der bisher erschienenen Ulmer Informatik-Berichte

Einige davon sind per FTP von `ftp.informatik.uni-ulm.de` erhältlich
Die mit * markierten Berichte sind vergriffen

List of technical reports published by the University of Ulm

Some of them are available by FTP from `ftp.informatik.uni-ulm.de`
Reports marked with * are out of print

91-01 *Ker-I Ko, P. Orponen, U. Schöning, O. Watanabe*
Instance Complexity

91-02* *K. Gladitz, H. Fassbender, H. Vogler*
Compiler-Based Implementation of Syntax-Directed Functional Programming

91-03 *Alfons Geser*
Relative Termination

91-04* *J. Köbler, U. Schöning, J. Toran*
Graph Isomorphism is low for PP

91-05 *Johannes Köbler, Thomas Thierauf*
Complexity Restricted Advice Functions

91-06 *Uwe Schöning*
Recent Highlights in Structural Complexity Theory

91-07 *F. Green, J. Köbler, J. Toran*
The Power of Middle Bit

91-08* *V.Arvind, Y. Han, L. Hamachandra, J. Köbler, A. Lozano, M. Mundhenk, A. Ogiwara, U. Schöning, R. Silvestri, T. Thierauf*
Reductions for Sets of Low Information Content

92-01* *Vikraman Arvind, Johannes Köbler, Martin Mundhenk*
On Bounded Truth-Table and Conjunctive Reductions to Sparse and Tally Sets

92-02* *Thomas Noll, Heiko Vogler*
Top-down Parsing with Simulataneous Evaluationof Noncircular Attribute Grammars

92-03 *Fakultät für Informatik*
17. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen

92-04 *V. Arvind, J. Köbler, M. Mundhenk*
Lowness and the Complexity of Sparse and Tally Descriptions

92-05 *Johannes Köbler*
Locating P/poly Optimally in the Extended Low Hierarchy

92-06 *Armin Kühnemann, Heiko Vogler*
Synthesized and inherited functions -a new computational model for syntax-directed semantics

92-07 *Heinz Fassbender, Heiko Vogler*
A Universal Unification Algorithm Based on Unification-Driven Leftmost Outermost Narrowing

92-08 *Uwe Schöning*
On Random Reductions from Sparse Sets to Tally Sets

92-09 *Hermann von Hasseln, Laura Martignon*
Consistency in Stochastic Network

92-10 *Michael Schmitt*
A Slightly Improved Upper Bound on the Size of Weights Sufficient to Represent Any Linearly Separable Boolean Function

92-11 *Johannes Köbler, Seinosuke Toda*
On the Power of Generalized MOD-Classes

92-12 *V. Arvind, J. Köbler, M. Mundhenk*
Reliable Reductions, High Sets and Low Sets

92-13 *Alfons Geser*
On a monotonic semantic path ordering

92-14* *Joost Engelfriet, Heiko Vogler*
The Translation Power of Top-Down Tree-To-Graph Transducers

93-01 *Alfred Lupper, Konrad Froitzheim*
AppleTalk Link Access Protocol basierend auf dem Abstract Personal Communications Manager

93-02 *M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch*
The COCOON Object Model

93-03 *Thomas Thierauf, Seinosuke Toda, Osamu Watanabe*
On Sets Bounded Truth-Table Reducible to P-selective Sets

93-04 *Jin-Yi Cai, Frederic Green, Thomas Thierauf*
On the Correlation of Symmetric Functions

93-05 *K.Kuhn, M.Reichert, M. Nathe, T. Beuter, C. Heinlein, P. Dadam*
A Conceptual Approach to an Open Hospital Information System

93-06 *Klaus Gaßner*
Rechnerunterstützung für die konzeptuelle Modellierung

93-07 *Ullrich Keßler, Peter Dadam*
Towards Customizable, Flexible Storage Structures for Complex Objects

94-01 *Michael Schmitt*
On the Complexity of Consistency Problems for Neurons with Binary Weights

94-02 *Armin Kühnemann, Heiko Vogler*
A Pumping Lemma for Output Languages of Attributed Tree Transducers

94-03 *Harry Buhrman, Jim Kadin, Thomas Thierauf*
On Functions Computable with Nonadaptive Queries to NP

94-04 *Heinz Faßbender, Heiko Vogler, Andrea Wedel*
Implementation of a Deterministic Partial E-Unification Algorithm for Macro Tree Transducers

94-05 *V. Arvind, J. Köbler, R. Schuler*
On Helping and Interactive Proof Systems

94-06 *Christian Kalus, Peter Dadam*
Incorporating record subtyping into a relational data model

94-07 *Markus Tresch, Marc H. Scholl*
A Classification of Multi-Database Languages

94-08 *Friedrich von Henke, Harald Rueß*
Arbeitstreffen Typtheorie: Zusammenfassung der Beiträge

94-09 *F.W. von Henke, A. Dold, H. Rueß, D. Schwier, M. Strecker*
Construction and Deduction Methods for the Formal Development of Software

94-10 *Axel Dold*
Formalisierung schematischer Algorithmen

94-11 *Johannes Köbler, Osamu Watanabe*
New Collapse Consequences of NP Having Small Circuits

94-12 *Rainer Schuler*
On Average Polynomial Time

94-13 *Rainer Schuler, Osamu Watanabe*
Towards Average-Case Complexity Analysis of NP Optimization Problems

94-14 *Wolfram Schulte, Ton Vullinghs*
Linking Reactive Software to the X-Window System

94-15 *Alfred Lupper*
Namensverwaltung und Adressierung in Distributed Shared Memory-Systemen

94-16 *Robert Regn*
Verteilte Unix-Betriebssysteme

94-17 *Helmuth Partsch*
Again on Recognition and Parsin of Context-Free Grammars: Two Exercises in Transformational Programming

94-18 *Helmuth Partsch*
Transformational Development of Data-Parallel Algorithms: an Exemple

Ulmer Informatik-Berichte

ISSN 0939-5091

Herausgeber: Fakultät für Informatik

Universität Ulm, Oberer Eselsberg, D-89069 Ulm