in Parallel Algorithm Derivation and Program Transformation,
Eds. R. Paige, J. Reif, and R. Wachter,
Kluwer Academic Publishers, Boston, 1993, pages 55–69.

Derivation of Parallel Sorting Algorithms

Douglas R. Smith[1]
email: smith@condor.kestrel.edu
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304 USA

Abstract

Parallel algorithms can be derived from formal problem specifications by applying a sequence of transformations that embody information about algorithms, data structures, and optimization techniques. The KIDS system provides automated support for this approach to algorithm design. This paper carries through the salient parts of a formal derivation for a well-known parallel sorting algorithm – Batcher's Even-Odd sort. The main difficulty lies in building up the problem domain theory within which the algorithm is inferred.

1 Introduction

This paper is intended as an introduction to formal and automated design of parallel algorithms. The level of formality is somewhat lessened in order to concentrate on the main issues. We derive Batcher's Odd-Even sort [2] and discuss the derivation of several other well-known parallel sorting algorithms.

Algorithms can be treated as a highly optimized composition of information about the problem being solved, algorithm paradigms, data structures, target architectures, and so on. An attempt to provide automated support for algorithm design must be based on a formal model of the composition process and

1. *representation of problem domain knowledge* - expressing the basic and derived concepts of the problem and the laws for reasoning about them. We formalize knowledge about a particular application domain as a parameterized *domain theory*.

2. *representation of programming knowledge* - we also use theories to capture knowledge of algorithms and data structures. The logical concept of interpretation between theories is the basis for applying programming knowledge in the form of theories [3, 12, 10].

[1]This research was supported in part by the Office of Naval Research under Grant N00014-90-J-1733 and in part by Air Force Office of Scientific Research under Contract F49620-91-C-0073.

Most, if not all, sorting algorithms can be derived as interpretations of the divide-and-conquer paradigm. Accordingly, we present a simplified divide-and-conquer theory and show how it can be applied to design the sort algorithms mentioned above.

There are a variety of reasons for turning to a derivational approach to algorithm design. First, a derivation is structured *proof of correctness*, so a derivational approach is in accordance with modern programming methodology that insists that programs and proofs be developed at the same time. Second, the compositional view provides an *explanation* of an algorithm in terms that are common to many algorithms. This description shows the commonalities between algorithms and how a small collection of general principles suffice to generate a large variety of algorithms. All too often, the published explanation of an algorithm is just a post-hoc proof of correctness that sheds little light on the process of inventing the algorithm in the first place. Such proofs are too specific to the algorithm and use overly general proof techniques, such as induction. The reader may wish to compare our derivation with the presentation of Even-Odd sort in textbooks such as [1, 5]. Third, derivations often come in families – the design decisions that are dependent on the target language and architecture can be separated out. This allows *retargeting* a abstract algorithm for a problem to a variety of concrete programs in different languages for different machines. Finally, *automated support* can be provided for formal derivations. The machine handles many of the lower-level details, freeing the designer to concentrate on developing the problem domain theory and making high-level design decisions.

2 KIDS Model of Design

The Kestrel Interactive Development System (KIDS) has served as a testbed for our experiments in automated program derivation [11]. The user typically goes through the following steps in using KIDS. We do not claim this to be a complete model of software development; however, this model is supported in KIDS and has been used to design and optimize over 60 algorithms. Applications areas have included scheduling, combinatorial design, sorting and searching, computational geometry, pattern matching, and linear and nonlinear programming.

1. *Develop a domain theory* – The user builds up a domain theory by defining appropriate types and operations. The user also provides laws that allow high-level reasoning about the defined operations. Our experience has been that laws describing the preservation of properties under various operations provide most of the laws that are needed to support design and optimization. In particular, distributive and monotonicity laws have turned out to be so important that KIDS has a theory development component that supports their automated derivation.

2. *Create a specification* – The user enters a specification stated in terms of the underlying domain theory.

3. *Apply a design tactic* – The user selects an algorithm design tactic from a menu and applies it to a specification. Currently KIDS has tactics for

simple problem reduction (reducing a specification to a library routine), divide-and-conquer, global search (binary search, backtrack, branch-and-bound), local search (hillclimbing), and problem reduction generators (dynamic programming and generalized branch-and-bound).

4. *Apply optimizations* – KIDS supports program optimization techniques such as simplification, partial evaluation, finite differencing, and other transformations. The user selects an optimization method from a menu and applies it by pointing at a program expression.

5. *Apply data type refinements* – The user can select implementations for the high-level data types in the program. Data type refinement rules carry out the details of constructing the implementation [3].

6. *Compile* – The resulting code is compiled to executable form. In a sense, KIDS can be regarded as a front-end to a conventonal compiler.

Actually, the user is free to apply any subset of the KIDS operations in any order – the above sequence is typical of our experiments in algorithm design. In this paper we mainly concentrate on the first three steps.

3 Derivation of a Mergesort

3.1 Domain Theory for Sorting

Suppose that we wish to sort a collection of objects belonging to some set α that is linearly-ordered under $\leq$. Here is a simple specification of the sorting problem:

$$Sort(x : bag(\alpha) \mid true)$$
$$returns(\ z : seq(\alpha) \mid x = Seq\text{–}to\text{–}bag(z)\ \wedge\ Ordered(z)\ )$$

Sort takes a bag (multiset) x of α objects and returns some sequence z such that the following *output condition* holds: the bag of objects in sequence z is the same as x and z must be ordered under $\leq$. The predicate *true* following the parameter x is called the *input condition* and specifies any constraints on inputs.

In order to support this specification formally, we need a domain theory of sorting that includes the theory of sequences and bags, has the linear-order $\langle \alpha, \leq \rangle$ as a parameter, and defines the concepts of *Seq–to–bag* and *Ordered*. The following parameterized theory accomplishes these ends:

Theory $Sorting(\langle \alpha, \leq \rangle : linear - order)$
  Imports $integer,\ bag(\alpha), seq(\alpha)$
  Operations
    $Ordered : seq(\alpha) \rightarrow Boolean$

  Axioms
    $\forall (S : seq(\alpha))\ (Ordered(S)$
      $\Leftrightarrow \forall (i)(i \in \{1..length(S) - 1\}\ \Longrightarrow\ S(i) \leq S(i+1)))$

Theorems

$Ordered([]) = true$
$\forall(a : \alpha)\ (Ordered([a]) = true)$
$\forall(y_1 : seq(\alpha), y_2 : seq(\alpha))$
$\quad (Ordered(y_1 +\!\!+ y_2) \Leftrightarrow Ordered(y_1)$
$\qquad\qquad\qquad\qquad \wedge\ Seq\text{–}to\text{–}bag(y_1) \leq Seq\text{–}to\text{–}bag(y_2)$
$\qquad\qquad\qquad\qquad \wedge\ Ordered(y_2))$

end–theory

Sorting theory imports *integer*, *bag*, and *sequence* theory. Sequences are constructed via [] (empty sequence), $[a]$ (singleton sequence), and $A +\!\!+ B$ (concatenation). For example,

$$[1, 2, 3] +\!\!+ [4, 5, 6] = [1, 2, 3, 4, 5, 6].$$

Several parallel sorting algorithms are based on an alternative set of constructors which use interleaving in place of concatenation: the *ilv* operator

$$[1, 2, 3]\ ilv\ [4, 5, 6] = [1, 4, 2, 5, 3, 6]$$

interleaves the elements of its arguments. We assume that the arguments to *ilv* have the same length, typically denoted n, and that it is defined by

$$A\ ilv\ B = C \Leftrightarrow \forall(i)(i \in \{1..n\} \implies C_{2i-1} = A_i\ \wedge\ C_{2i} = B_i).$$

In Section 4 we develop some of the theory of sequences based on the *ilv* constructor.

Bags have an analogous set of constructors: $\{\!\{\}\!\}$ (empty bag), $\{\!\{a\}\!\}$ (singleton bag), and A d B (associative and commutative bag union). The operator *Seq-to-bag* coerces sequences to bags by forgetting the ordering implicit in the sequence. *Seq-to-bag* obeys the following distributive laws:

$$Seq\text{–}to\text{–}bag([]) = \{\!\{\}\!\}$$

$$\forall(a : \alpha)\ Seq\text{–}to\text{–}bag([a]) = \{\!\{a\}\!\}$$

$$\forall(y_1 : seq(\alpha), y_2 : seq(\alpha))$$
$$\quad Seq\text{–}to\text{–}bag(y_1 +\!\!+ y_2) = Seq\text{–}to\text{–}bag(y_1)\ \text{d}\ Seq\text{–}to\text{–}bag(y_1)$$

$$\forall(y_1 : seq(\alpha), y_2 : seq(\alpha))$$
$$\quad Seq\text{–}to\text{–}bag(y_1\ ilv\ y_2) = Seq\text{–}to\text{–}bag(y_1)\ \text{d}\ Seq\text{–}to\text{–}bag(y_1)$$

In the sequel we will omit universal quantifiers whenever it is possible to simplify the presentation without sacrificing clarity.

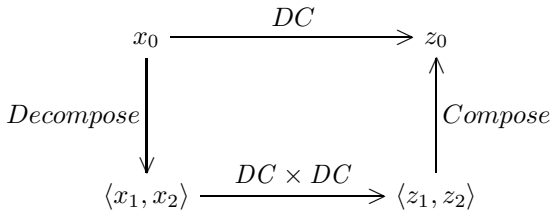
3.2 Divide-and-Conquer Theory

Most sorting algorithms are based on the divide-and-conquer paradigm: If the input is primitive then a solution is obtained directly, by simple code. Otherwise a solution is obtained by decomposing the input into parts, independently solving the parts, then composing the results. Program termination is guaranteed by requiring that decomposition is monotonic with respect to a suitable well-founded ordering. In this paper we focus on divide-and-conquer algorithms that have the following general form:

$DC(x_0 : D \mid I(x_0))$
$returns(\ z : R \mid O(x_0, z))$
$=$ **if** $Primitive(x_0)$
then $Directly\text{–}Solve(x_0)$
else $let\ \langle x_1, x_2\rangle = Decompose(x_0)$
$Compose(DC(x_1),\ DC(x_2))$

We refer to *Decompose* as a decomposition operator, *Compose* as a composition operator, *Primitive* as a control predicate, and $Directly{-}Solve$ as a primitive operator.

The essence of a divide-and-conquer algorithm can be presented via a *reduction diagram*:



which should be read as follows. Given input x_0, an acceptable solution z_0 can be found by decomposing x_0 into two subproblems x_1 and x_2, solving these subproblem recursively yielding solutions z_1 and z_2 respectively, and then composing z_1 and z_2 to form z_0.

In the derivations of this paper we will usually ignore the primitive predicate and *Directly-Solve* operator – the interesting design work lies in calculating compatible pairs of *Decompose* and *Compose* operators.

The following mergesort program is an instance of this scheme:

$MSort(b_0 : bag(integer))$
$returns(\ z : seq(\alpha) \mid x = Seq\text{–}to\text{–}bag(z)\ \wedge\ Ordered(z)\ )$
$=$ **if** $size(b_0) \leq 1$
then b_0
else $let\ \langle b_1, b_2\rangle = Split(b_0)$
$Merge(MSort(b_1),\ MSort(b_2))$

Here *Split* decomposes a bag into two subbags of roughly equal size and $Merge$ composes two sorted sequences to form a sorted sequence.

The characteristic that subproblems are solved independently gives the divide-and-conquer notion its great potential in parallel environments. Another aspect of divide-and-conquer is that the recursive decomposition can often be performed implicitly, thereby enabling a purely bottom-up computation. For example, in the Mergesort algorithm, the only reason for the recursive splitting is to control the order of composition (merging) of sorted subproblem solutions. However the pattern of merging is easily determined at design-time and leads to the usual binary tree computation pattern.

To express the essence of divide-and-conquer, we define a divide-and-conquer theory comprised of various sorts, function, predicates, and axioms that assure

that the above scheme correctly solves a given problem. A simplified divide-and-conquer theory is as follows (for more details see [8, 9]):

Theory *Divide–and–Conquer*

Sorts D, R	domain and range of a problem
Operations	
$I : D \rightarrow Boolean$	input condition
$O : D \times R \rightarrow Boolean$	output condition
$primitive : D \rightarrow Boolean$	control predicate
$O_{Decompose} : D \times D \times D \rightarrow Boolean$	output condition for Decompose
$O_{Compose} : R \times R \times R \rightarrow Boolean$	output condition for Compose
$\succ : D \times D \rightarrow Boolean$	well-founded order

Soundness Axiom

$$O_{Decompose}(x_0, x_1, x_2) \wedge O(x_1, z_1) \wedge O(x_2, z_2) \wedge O_{Compose}(z_0, z_1, z_2) \Longrightarrow O(x_0, z_0)$$

...

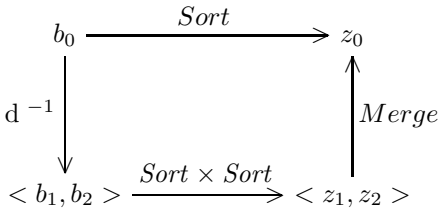
end–theory

The intuitive meaning of the Soundness Axiom is that if input x_0 decomposes into a pair of subproblems $\langle x_1, x_2 \rangle$, and z_1 and z_2 are solutions to subproblems x_1 and x_2 respectively, and furthermore solutions z_1 and z_2 can be composed to form solution z_0, then z_0 is guaranteed to be a solution to input x_0. There are other axioms that are required: well-foundedness conditions on $\succ$ and *admissability conditions* that assure that Decompose and Compose can be refined to total functions over their domains. We ignore these in order to concentrate on the essentials of the design process.

The main difficulty in designing an instance of the divide-and-conquer scheme for a particular problem lies in constructing decomposition and composition operators that work together. The following is a simplified version of a tactic in [8].

1. Choose a simple decomposition operator and well-founded order.
2. Derive the control predicate based on the conditions under which the decomposition operator preserves the well-founded order and produces legal subproblems.
3. Derive the input and output conditions of the composition operator using the Soundness Axiom of divide-and-conquer theory.
4. Design an algorithm for the composition operator.
5. Design an algorithm for the primitive operator.

Mergesort is derived by choosing d^{-1} as a simple (nondeterministic) decomposition operator. A specification for the well-known merge operation is derived using the Soundness Axiom.



A similar tactic based on choosing a simple composition operator and then solving for the decomposition operator is also presented in [8]. This tactic can be used to derive selection sort and quicksort-like algorithms.

Deriving the output condition of the composition operator is the most challenging step and bears further explanation. The *Soundness Axiom* of divide-and-conquer theory relates the output conditions of the subalgorithms to the output condition of the whole divide-and-conquer algorithm:

$$\begin{array}{l} O_{Decompose}(x_0, x_1, x_2) \\ \quad \wedge\, O(x_1, z_1) \ \wedge\ O(x_2, z_2) \\ \quad \wedge\, O_{Compose}(z_0, z_1, z_2) \\ \quad \Longrightarrow\ O(x_0, z_0) \end{array}$$

For design purposes this constraint can be treated as having three unknowns: O, $O_{Decompose}$, and $O_{Compose}$. Given O from the original specification, we supply an expression for $O_{Decompose}$ then reason backwards from the consequent to an expression over the program variables z_0, z_1, and z_2. This derived expression is taken as the output condition of $Compose$.

Returning to Mergesort, suppose that we choose d^{-1} as a simple decomposition operator. To proceed with the tactic, we instantiate the Soundness Axiom with the following substitutions

$$\begin{array}{rcl} O_{Decompose} & \mapsto & \lambda(b_0, b_1, b_2)\, b_0 = b_1\ \mathrm{d}\ b_2 \\ O & \mapsto & \lambda(b, z)\, b = Seq\text{–}to\text{–}bag(z) \ \wedge\ Ordered(z) \end{array}$$

yielding

$$\begin{array}{l} b_0 = b_1\ \mathrm{d}\ b_2 \\ \quad \wedge\, b_1 = Seq\text{–}to\text{–}bag(z_1) \ \wedge\ Ordered(z_1) \\ \quad \wedge\, b_2 = Seq\text{–}to\text{–}bag(z_2) \ \wedge\ Ordered(z_2) \\ \quad \wedge\, O_{Compose}(z_0, z_1, z_2) \\ \quad \Longrightarrow\ b_0 = Seq\text{–}to\text{–}bag(z_0) \ \wedge\ Ordered(z_0) \end{array}$$

To derive $O_{Compose}(z_0, z_1, z_2)$ we reason backwards from the consequent $b_0 = Seq\text{–}to\text{–}bag(z_0) \ \wedge\ Ordered(z_0)$ toward a sufficient condition expressed over the variables $\{z_0, z_1, z_2\}$ modulo the assumptions of the antecedent:

$$b_0 = \mathit{Seq\text{–}to\text{–}bag}(z_0) \ \wedge \ \mathit{Ordered}(z_0)$$

$$\iff \qquad \text{using assumption } b_0 = b_1 \ \mathrm{d} \ b_2$$

$$b_1 \ \mathrm{d} \ b_2 = \mathit{Seq\text{–}to\text{–}bag}(z_0) \ \wedge \ \mathit{Ordered}(z_0)$$

$$\iff \qquad \text{using assumption } b_i = \mathit{Seq\text{–}to\text{–}bag}(z_i), \ i = 1, 2$$

$$\mathit{Seq\text{–}to\text{–}bag}(z_1) \ \mathrm{d} \ \mathit{Seq\text{–}to\text{–}bag}(z_2) = \mathit{Seq\text{–}to\text{–}bag}(z_0) \ \wedge \mathit{Ordered}(z_0).$$

This last expression is a sufficient condition expressed in terms of the variables $\{z_0, z_1, z_2\}$ and so we take it to be the output condition for *Compose*. In other words, we ensure that the Soundness Axiom holds by taking this expression as a constraint on the behavior of the composition operator.

The input condition to the composition operator is obtained by forward inference from the antecedent of the soundness axiom; here we have the (trivial) consequences $\mathit{Ordered}(z_1)$ and $\mathit{Ordered}(z_2)$. Only consequences expressed in terms of the input variables z_1 and z_2 are useful.

Thus we have derived a formal specification for *Compose*:

$$\begin{array}{l} \mathit{Merge}(A : \mathit{seq}(\mathit{integer}), B : \mathit{seq}(\mathit{integer}) \mid \mathit{Ordered}(A) \ \wedge \ \mathit{Ordered}(B)) \\ \quad \mathit{returns}(\ z : \mathit{seq}(\mathit{integer}) \\ \qquad\qquad \mid \mathit{Seq\text{–}to\text{–}bag}(A) \ \mathrm{d} \ \mathit{Seq\text{–}to\text{–}bag}(B) = \mathit{Seq\text{–}to\text{–}bag}(z) \\ \qquad\qquad \wedge \mathit{Ordered}(z) \ ). \end{array}$$

Merge is now a derived concept in *Sorting* theory. We later derive laws for it, but now we proceed to design an algorithm to satisfy this specification. The usual sequential algorithm for merging is based on choosing a simple "cons" composition operator and deriving a decomposition operator [8]. However this algorithm is inherently sequential and requires linear time.

4 Batcher's Odd-Even Sort

Batcher's Odd-Even sort algorithm [2] is a mergesort algorithm in which the merge operator itself is a divide-and-conquer algorithm. The Odd-Even merge is derived by choosing a simple decomposition operator based on ilv and deriving constraints on the composition operator.

Before proceeding with algorithm design we need to develop some of the theory of sequences based on the ilv constructor. Generally, we develop a domain theory by deriving laws about the various concepts of the domain. In particular we have found that distributive, monotonicity, and invariance laws provide most of the laws needed to support formal design. This suggests that we develop laws for various sorting concepts, such as *Seq-to-bag* and *Ordered*. From Section 3 we have

Theorem 1. *Distributing Seq-to-bag over sequence constructors.*

1.1. $\mathit{Seq\text{–}to\text{–}bag}([]) = \ \{\!\{\}\!\}$
1.2. $\mathit{Seq\text{–}to\text{–}bag}([a]) = \ \{\!\{a\}\!\}$
1.3. $\mathit{Seq\text{–}to\text{–}bag}(S_1 \ \mathit{ilv} \ S_2) = \mathit{Seq\text{–}to\text{–}bag}(S_1) \ \mathrm{d} \ \mathit{Seq\text{–}to\text{–}bag}(S_2)$

It is not obvious how to distribute *Ordered* over ilv , so we try to derive it. In this derivation let n denote the length of both A and B.

$$
\begin{array}{ll}
Ordered(A\ ilv\ B) & \\
\iff & \text{by definition of } Ordered \\
& \forall(i)(i \in \{1..2n-1\} \implies (A\ ilv\ B)_i \leq (A\ ilv\ B)_{i+1}) \\
\iff & \text{change of index} \\
& \forall(j)(j \in \{1..n\} \implies (A\ ilv\ B)_{2j-1} \leq (A\ ilv\ B)_{2j}) \\
& \wedge\ \forall(j)(j \in \{1..n-1\} \implies (A\ ilv\ B)_{2j} \leq (A\ ilv\ B)_{2j+1}) \\
\iff & \text{by definition of } ilv \\
& \forall(j)(j \in \{1..n\} \implies A_j \leq B_j) \\
& \wedge\ \forall(j)(j \in \{1..n-1\} \implies B_j \leq A_{j+1}).
\end{array}
$$

These last two conjuncts are similar in form and suggest the need for a new concept definition and perhaps new notation. Suppose we define $A \leq^* B$ *iff* $A_i \leq B_i$ for $i \in \{1 \ldots n\}$. This allows us to express the first conjunct as $A \leq^* B$, but then we cannot quite express the second concept – we need to generalize to allow an offset in the comparison:

Definition 1. A pair of sequences A and B of length n are *pairwise-ordered with offset k*, written $A \leq_k^* B$, iff $A_i \leq B_{i+k}$ for $i \in \{1 \ldots n-k\}$.

Then the derivation above yields the following simple law

Theorem 2. *Conditions under which an interleaved sequence is Ordered.*
For all sequences A, B,
$Ordered(A\ ilv\ B) \iff A \leq_0^* B\ \wedge\ B \leq_1^* A.$

Note that this definition provides a proper generalization of the notion of orderedness:

Theorem 3. *Ordered as a diagonal specialization of* $\leq_i^*$.
For all sequences S,
$Ordered(S) \iff S \leq_1^* S$

Other laws are easily derived:

Theorem 4. *Transitivity of* $\leq_i^*$.
For all sequences A, B, C of equal length and integers i and j,
$A \leq_i^* B\ \wedge\ B \leq_j^* C \implies A \leq_{i+j}^* C$

As a simple consequence we have

Corollary 1. *Only Ordered sequences interleave to form Ordered sequences.*
For all sequences A, B,
$Ordered(A\ ilv\ B) \implies Ordered(A)\ \wedge\ Ordered(B).$

Proof:

$Ordered(A\ ilv\ B)$

$\iff$ by Theorem 2

$A \leq_0^* B\ \wedge\ B \leq_1^* A$

$\implies$ applying Theorem 4 twice

$A \leq_1^* A\ \wedge\ B \leq_1^* B$

$\iff$ by Theorem 3

$Ordered(A)\ \wedge\ Ordered(B).\ \Lambda$

Theorem 5. *Monotonicity of $\leq_i^*$ with respect to merging.*
For all sequences A_1, A_2, B_1, and B_2 and integers i,
$A_1 \leq_i^* A_2\ \wedge\ B_1 \leq_i^* B_2\ \implies\ Merge(A_1, B_1) \leq_{2i}^* Merge(A_2, B_2)$

We can apply the basic sort operation $sort2(x, y) = \langle min(x, y), max(x, y)\rangle$ over parallel sequences, just as we did with the comparator $\leq$.

Definition 2. *Pairwise-sort of sequences with offset k.*
Define $sort2_k^*(A, B) = \langle A', B'\rangle$ such that
(1) for $i \leq k$, $B'_i = B_i$
(2) for $i = 1, \ldots, n-k$, $\langle A'_i, B'_{i+k}\rangle = sort2(A_i, B_{i+k})$
(3) for $i > n-k$, $A'_i = A_i$

For example, $sort2_1^*([2, 3, 8, 9], [0, 1, 4, 5]) = \langle[1, 3, 5, 9], [0, 2, 4, 8]\rangle$. Laws for $sort2_k^*$ can be developed:

Theorem 6. *$sort2_k^*$ establishes $\leq_k^*$.*
For all sequences A, B, A', and B', and integer k,
$sort2_k^*(A, B) = \langle A', B'\rangle\ \implies\ A' \leq_k^* B'$.

This theorem is a trivial consequence of the definition of $sort2_k^*$. The following theorems give conditions under which important properties of the domain theory ($\leq_i^*$, $Ordered$) are preserved under under the $sort2_k^*$ operation. They can be proved using straightforward analysis of cases.

Theorem 7. *Ordered is invariant under $sort2_k^*$.*
For all sequences A, B and integer k,
$Ordered(A)\ \wedge\ Ordered(B)\ \wedge\ sort2_k^*(A, B) = \langle A', B'\rangle$
$\implies\ Ordered(A')\ \wedge\ Ordered(B')$

Theorem 8. *Invariance of $A \leq_i^* B$ with respect to $sort2_k^*(A, B)$.*
For all sequences A, B and integers i and k,
$A \leq_i^* B\ \wedge\ sort2_k^*(A, B) = \langle A', B'\rangle\ \implies\ A' \leq_i^* B'$

Theorem 9. *Invariance of $A \leq_i^* B$ with respect to $sort2_k^*(B, A)$.*
For all sequences A, B and $0 \leq i < k$,
$A \leq_{i+k}^* A\ \wedge\ B \leq_{i+k}^* B\ \wedge\ A \leq_i^* B\ \wedge\ B \leq_{i+2k}^* A\ \wedge\ sort2_k^*(B, A) = \langle B', A'\rangle$
$\implies\ A' \leq_i^* B'$

With these concepts and laws in hand, we can proceed to derive Batcher's Odd-Even mergesort. It can be derived simply by choosing to decompose the inputs to Merge by uninterleaving them.

$$\begin{array}{ccc} \langle A_0, B_0\rangle & \xrightarrow{\quad Merge \quad} & S_0 : seq(integer) \\ \Big\downarrow ilv^{-2} & & \Big\uparrow ? \\ \langle\langle A_1, B_1\rangle, \langle A_2, B_2\rangle\rangle & \xrightarrow{Merge \times Merge} & \langle S_1, S_2\rangle \end{array}$$

where ilv^{-2} means $A_0 = A_1\ ilv\ A_2$ and $B_0 = B_1\ ilv\ B_2$. Note how this decomposition operator creates subproblems of roughly the same size which provides good opportunities for parallel computation. Note also that this decomposition operator must ensure that the subproblems $\langle A_1, B_1\rangle$ and $\langle A_2, B_2\rangle$ satisfy the input conditions of *Merge*. This property is assured by Corollary 1.

We proceed by instantiating the Soundness Axiom as before:

$$\begin{array}{l} A_0 = A_1\ ilv\ A_2\ \wedge\ Ordered(A_0) \\ \wedge\ B_0 = B_1\ ilv\ B_2\ \wedge\ Ordered(B_0) \\ \wedge\ Seq\text{–}to\text{–}bag(S_1) = Seq\text{–}to\text{–}bag(A_1)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(B_2)\ \wedge\ Ordered(S_1) \\ \wedge\ Seq\text{–}to\text{–}bag(S_2) = Seq\text{–}to\text{–}bag(A_2)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(B_2)\ \wedge\ Ordered(S_2) \\ \wedge\ O_{Compose}(S_0, S_1, S_2) \\ \Longrightarrow\ Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(A_0)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(B_0) \\ \qquad \wedge\ Ordered(S_0) \end{array}$$

Constraints on $O_{Compose}$ are derived as follows:

$$Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(A_0)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(B_0)\ \wedge\ Ordered(S_0)$$

$\Longleftrightarrow$ by assumption

$$\begin{array}{l} Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(A_1\ ilv\ A_2) \\ \qquad\qquad\qquad\quad \mathrm{d}\ Seq\text{–}to\text{–}bag(B_1\ ilv\ B_2) \\ \wedge\ Ordered(S_0) \end{array}$$

$\Longleftrightarrow$ distributing $Seq\text{–}to\text{–}bag$ over ilv

$$\begin{array}{l} Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(A_1)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(A_2) \\ \qquad\qquad\qquad\quad \mathrm{d}\ Seq\text{–}to\text{–}bag(B_1)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(B_2) \\ \wedge\ Ordered(S_0) \end{array}$$

$\Longleftrightarrow$ by assumption

$$\begin{array}{l} Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(S_1)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(S_2) \\ \wedge\ Ordered(S_0). \end{array}$$

The input conditions on $Merge$ are derived by forward inference from the

assumptions above:

$$A_0 = A_1\ ilv\ A_2\ \wedge\ Ordered(A_0)$$
$$\wedge\ B_0 = B_1\ ilv\ B_2\ \wedge\ Ordered(B_0)$$
$$\wedge\ Ordered(S_1)\ \wedge\ Ordered(S_2)$$

$\Longrightarrow$ distributing $Ordered$ over ilv

$$A_1 \leq_0^* A_2\ \wedge\ A_2 \leq_1^* A_1$$
$$\wedge\ B_1 \leq_0^* B_2\ \wedge\ B_2 \leq_1^* B_1$$
$$\wedge\ Ordered(S_1)\ \wedge\ Ordered(S_2)$$

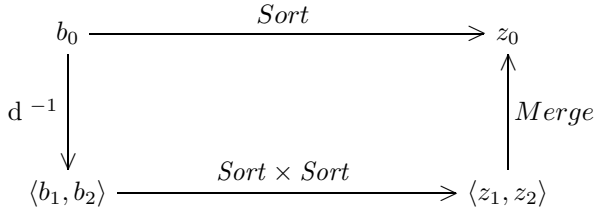
$\Longrightarrow$ by monotonicity of $\leq_i^*$ with respect to $Merge$

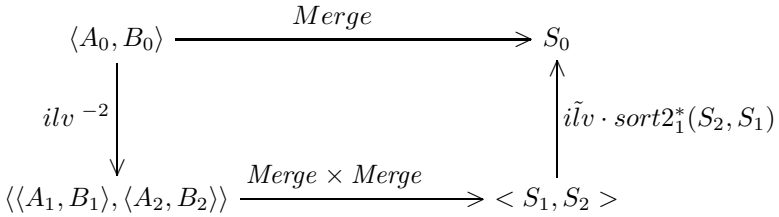
$$S_1 \leq_0^* S_2\ \wedge\ S_2 \leq_2^* S_1$$
$$\wedge\ Ordered(S_1)\ \wedge\ Ordered(S_2).$$

Thus we have derived the specification

$$Merge\text{–}Compose(S_1 : seq(integer), S_2 : seq(integer)$$
$$\mid S_1 \leq_0^* S_2\ \wedge\ S_2 \leq_2^* S_1\ \wedge\ Ordered(S_1)\ \wedge\ Ordered(S_2))$$
$$returns(\ S_0 : seq(integer)$$
$$\mid Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(S_1)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(S_2)$$
$$\wedge\ Ordered(S_0)\ ).$$

How can this specification be satisfied? Theorems 1.3 and 2 suggest ilv since it would establish the output conditions of $Merge\text{–}Compose$. Theorem 2 requires that we achieve the input condition $S_1 \leq_0^* S_2\ \wedge\ S_2 \leq_1^* S_1$ first. But Theorem 6 ($sort2_k^*$ establishes $\leq_k^*$) enables us to apply $Sort2_1^*(S_2, S_1)$ in order to achieve the second conjunct. Theorems 7, 8, and 9 ensure that $S_1 \leq_0^* S_2$ remains invariant. So $Merge\text{–}Compose$ is satisfied by $\tilde{ilv} \cdot sort2_1^*(S_2, S_1)$. The final algorithm in diagram form is





To simplify the analysis, assume that the input to *Sort* has length $n = 2^m$. Given n processors, *Merge* runs in time

$$\begin{aligned} T_{Merge}(n) &= max(T_{Merge}(n/2), T_{Merge}(n/2)) + O(1) \\ &= O(log(n)) \end{aligned}$$

since the decomposition and composition operators both can be evaluated in constant time and the recursion goes to depth $O(log(n))$.

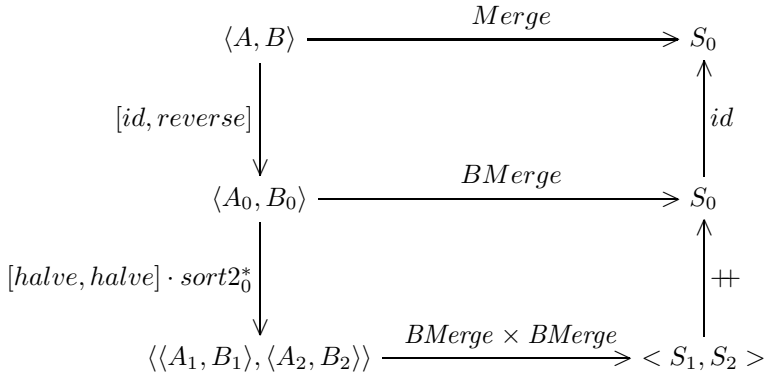
The decomposition operator d^{-1} in *Sort* is nondeterministic. This is an advantage at this stage of design since it allows us to defer committments and make choices that will maximize performance. In this case the complexity of *Sort* is calculated via the recurrence

$$T_{Sort}(n) = max(T_{Sort}(a(n)), T_{Sort}(b(n))) + O(log(n))$$

which is optimized by taking $a(n) = b(n) = n/2$ – that is, we split the input bag in half. Given n processors this algorithm runs in $O(log^2(n))$ time, so it is suboptimal for sorting. However, according to [7], Batcher's Odd-Even sort is the most commonly used of parallel sort algorithms.

5 Related Sorting Algorithms

Several other parallel sorting algorithms can be developed using the techniques above. Batcher's bitonic sort [2] and the Periodic Balanced Sort [4] are also basically mergesort algorithms. They differ from Odd-Even sort in that the merge operation is a divide-and-conquer based on concatenation as the composition operator. For example, bitonic merge can be diagrammed as follows:



The essential fact about using $+\!\!+$ as a composition operator is that $\langle\langle A_1, B_1\rangle, \langle A_2, B_2\rangle\rangle$ must be a partition in the sense that no element of A_1 or B_1 is greater than any element of A_2 and B_2. The cleverness of the algorithm lies in a special property of sequences that allows a simple operation ($sort2_0^*$ here) to effectively produce a partition. This property is called "bitonicity" for bitonic sort and "balanced" for the periodic balanced sort. (The operation $\langle A_0, B_0\rangle = \langle id(A), reverse(B)\rangle$ establishes the bitonic property and decomposition preserves it). The challenge in deriving these algorithms lies in discovering these properties given that one wants a divide-and-conquer algorithm

based on ++ as composition. Is there a systematic way to discover these properties or must we rely on creative invention? Admittedly, there may be other frameworks within which the discovery of these properties is easier.

Another well-known parallel sort algorithm is odd-even transposition sort. This can be viewed as a parallel variant of bubble-sort which in turn is derivable as a selection sort (local search is used to derive the selection subalgorithm). See the paper by Partsch in this volume.

The *ilv* constructor for sequences has many other applications including polynomial evaluation, discrete fast fourier transform, and matrix transposition. Butterfly and shuffle networks are natural architectures for implementing algorithms based on *ilv* [6].

6 Concluding Remarks

The Odd-Even sort algorithm is simpler to state than to derive. The properties of a *ilv*-based theory of sequences are much harder to understand and develop than a concatenation-based theory. However, the payoff is an abundance of algorithms with good parallel properties.

The derivation presented here requires a closer, more intensive development of the domain theory than most published derivations in the literature. The development was guided by some higher-level principles – invariance properties, distributive laws, and monotonicity laws provide most of the inference rules needed to support algorithm design.

We have used KIDS to derive a variant of the usual sequential mergesort algorithm [8]. However, simplifying assumptions in the implemented design tactic for divide-and-conquer disallow the derivation of the *ilv*-based merge described in Section 4. We are currently implementing a new algorithm design system based on [10] which overcomes these (and other) limitations and we see no essential difficulty in deriving the Odd-Even sort once the domain theory is place. Support for developing domain theories has not yet received enough serious attention in KIDS. For some theories, we have used KIDS to derive almost all of the laws needed to support the algorithm design process; other theories have been developed entirely by hand. In the current system, the theory development presented in Sections 3.1 and 4 would be done mostly manually.

The general message of this paper is that good parallel algorithms can be formally derived and that such derivations depend on the systematic development of the theory underlying the application domain. Furthermore, machine support can envisioned for both the theory development and algorithm derivation processes and this kind of support can be partially demonstrated at present.

Key elements of theory development are (1) defining basic concepts, operations, relations and the laws (axioms) that constrain their interpretation, (2) developing derived concepts, operations, and relations and important laws governing their behavior. The principle of seeking properties that are invariant under change or, conversely, operations that preserve important properties, provides strong guidance in theory development. In particular, distributive, monotonicity, and fixpoint laws are especially valuable and machine support for their acquisition is an important research topic.

References

[1] Akl, S. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1989.

[2] Batcher, K. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference* (1968), vol. 32, pp. 307–314.

[3] Blaine, L., and Goldberg, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.

[4] Dowd, M., Perl, Y., Rudolph, L., and Saks, M. The periodic balanced sorting network. *Journal of the ACM 36*, 4 (October 1989), 738–757.

[5] Gibbons, A., and Rytter, W. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, 1988.

[6] Jones, G., and Sheeran, M. Collecting butterflies. Tech. Rep. PRG-91, Oxford University, Programming Research Group, February 1991.

[7] Leighton, F. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.

[8] Smith, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (September 1985), 43–96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

[9] Smith, D. R. Structure and design of problem reduction generators. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 91–124.

[10] Smith, D. R. Constructing specification morphisms. Tech. Rep. KES.U.92.1, Kestrel Institute, January 1992. in Journal of Symbolic Computation, Special Issue on Automatic Programming, May-June 1993.

[11] Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.

[12] Smith, D. R., and Lowry, M. R. Algorithm theories and design tactics. *Science of Computer Programming 14*, 2-3 (October 1990), 305–321.