# CoPa: a Parallel Programming Language for Collections

Dan Suciu[*] Val Tannen[†]

**Abstract**

In this paper we propose a new framework for parallel processing of collections. We define a high-level language called CoPa for processing nested sets, bags, and *sequences* (a generalization of arrays and lists). CoPa includes most features found in query languages for object-oriented or object-relational databases, and has, in addition, a powerful form of recursion not found in query languages. CoPa has a formal *declarative* definition of parallel complexity, as part of its specification. We prove the existence of a *complexity-preserving* compilation for CoPa, i.e. one which offers upper-bound guarantees for the parallel complexity of the compiled code. The majority of the compilation process is architecture-independent, using a parallel vector machine model (BVRAM). The BVRAM instructions form a sequence-algebra which is of independent interest, and have been carefully chosen to reconcile two conflicting demands: supporting the complexity-preserving compilation of CoPa's high-level constructs, and efficient implementability on a variety of architectures. The latter allows us to establish comparisons with some of the parallel algorithms work through a provably optimal implementation of the BVRAM on butterfly networks. In targeting more practical architectures we use the LogP model. Here we prove that monotone data communications admit optimal implementations on the LogP model, and use that to implement the BVRAM efficiently. Finally, we tested the feasibility of our entire approach to compilation by running some experiments on a LogP simulator. Their goal is to compare, for both speedup and scaleup, the three components of the total running time: the data communication cost, the control communication cost, and the local computations.

## 1 Introduction

This work focuses on the parallel processing of collection data structures. The main motivation comes from a database perspective. Relational database queries have proven to be highly suitable for parallel evaluation. On one hand relational data is uniform, so it can be easily partitioned uniformly across the nodes of a parallel architecture. On the other hand, the operators in the relational algebra are simple and can be efficiently computed in parallel with good load balance. Both *shared-memory* and *shared-nothing* architectures have been studied intensively for parallel databases, with the latter having more potential for scaleup. Today's commercial database management systems (DBMS) achieve almost linear speedup and almost constant scaleup[1] on both architectures.

Modern application however often require data which is more complex than the relational model. To

[*]AT&T Labs, `suciu@research.att.com`. This work was done while the author was at the University of Pennsylvania and was supported by NSF Grant CCR-90-57570, ONR Contract NOOO14-93-11284, and by a fellowship from the Institute for Research in Cognitive Science.

[†]University of Pennsylvania, `val@cis.upenn.edu`. The author was supported by NSF Grant CCR-90-57570 and ONR Contract NOOO14-93-11284.

[1]Scaleup measures the running time when both the database size and the number of nodes on a parallel architecture increase by the same factor.

meet these new demands, new data models have been proposed. *Object-oriented* databases [25, 20, 4] view the data as a collection of interlinked objects. The focus of query evaluation switches from processing massive, but uniform data sets to link-traversal in a highly interconnected collection of objects. On a parallel shared-nothing architecture, when a link traverses node boundaries, either the query or the data has to migrate between nodes [33, 18]. As a consequence, efficient parallel evaluation is harder to achieve for this model than for the relational one. An additional complication was caused by the absence of genuine query languages. With the important exception of O2 [25], OODBMS programming has been done only in general-purpose object-oriented languages extended with some collection datatypes and with a persistence mechanism (eg. ObjectStore [44]). Any attempt to parallelize such a language would face the usual problems of dealing with side-effects, etc. The situation has improved with the ODMG standard proposal of OQL [20], based on the O2 query language and that has been adopted by most commercial systems. Still, these difficulties have hindered the emergence of parallel implementations of OODBMS.

By contrast the *object-relational* [63] model shifts the focus back to sets. Data is again represented as relations, but now the relations' attributes are allowed to be user-defined types or other relations. Besides sets, the model also includes bags (multisets), arrays and lists, generically called *collections*. In short, the object-relational model extends the relational one by allowing (a) additional kinds of collections, (b) nested collections, and (c) user-defined types. This model also reaffirms the predominance of a query language for accessing and processing data, as opposed to a general-purpose language. Typically such a query language [63] is obtained by extending SQL to the new data features, and is related to the ongoing SQL-3 standardization process [28]. We note however that with a few simple modifications OQL could serve as a superior object-relational query language.

Existing parallel evaluation techniques for relational query languages rely mainly on efficient parallel join algorithms, often based on hashing techniques [61, 53] These do not appear to extend to object-oriented or object-relational languages because of the new features. First, nested collections give rise naturally to *nested parallelism*. We will shortly (section 1.1) give an example that illustrates the desirability of nested parallelism, and also some of the difficulties it creates for compilation. The main challenge is to avoid data skew. In fact, in relational databases, hash-based techniques also perform poorly in the presence of data skew [27]. Second, object-relational queries often involve user-defined functions: when these are expensive, they create additional data skew problems. And finally ordered collections, like arrays and lists, require new processing techniques, different from those for sets.

There has been a good deal of work on data-parallel techniques and languages that are all or in part data-parallel: there exist parallel extensions of FORTRAN, like High Performance Fortran [43] and PTRAN [3, 24], parallel extensions of C, like C* [69],$C^{**}$ [45], of Lisp, like CM List [39] and Paralation Lisp [60, 51], and applicative parallel programming languages, like NESL [8, 10, 11], Sisal [31, 62, 30], Crystal [21], Proteus [54, 34, 59], and Data-parallel ML [32, 37, 38]. None of these have been concerned with query constructs and their integration in such languages.

In this paper we propose a new framework for parallel processing of collections. We define a high-level language called CoPa for processing nested sets, bags, and *sequences* (a generalization of arrays and lists). CoPa includes includes all the features found in query languages for object-oriented or object-relational databases except for link traversal. These features coexist naturally with a powerful form of recursion that allows expressing most parallel algorithms without slowdown. The **main contribution** of the paper is a parallel compilation technique for CoPa with the following characteristics (see Figure 1):

- The majority of the compilation process is *architecture-independent*, using intermediate languages that are of interest in themselves, such as SA, an algebra of operations on flat sequences,

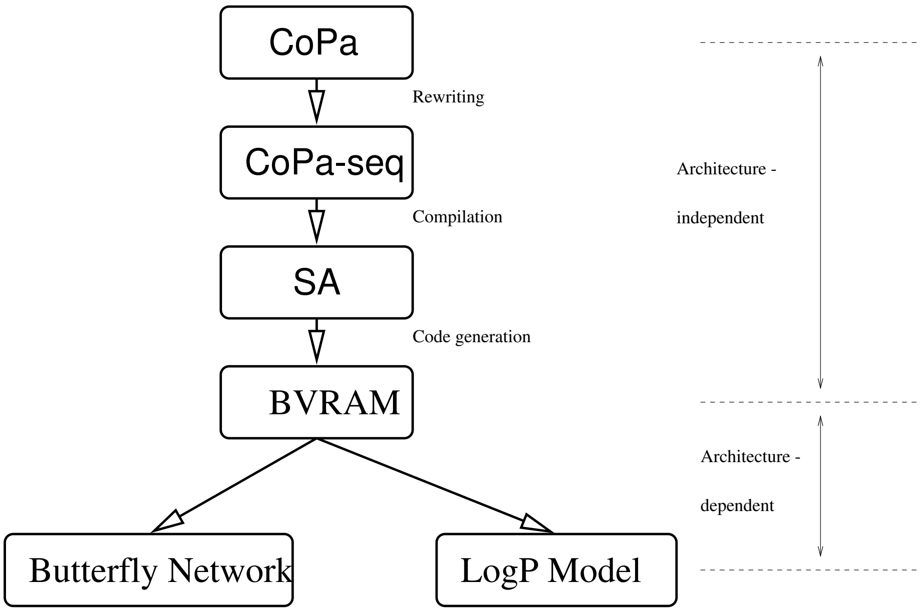


Figure 1: CoPa complexity preserving compilation

and BVRAM (below).

- Formal *declarative* definitions of time $T$ and work $W$ complexity are part of the specification of CoPa and of that of the intermediate languages. We prove that the corresponding compilation phases preserve the complexity of the code.

- The last architecture-independent compilation step targets a parallel vector machine model we called BVRAM (Bounded Vector Random Access Machine). The BVRAM *instructions* have been carefully chosen to reconcile two conflicting demands: supporting the complexity-preserving compilation of CoPa's high-level constructs, and efficient implementability on a variety of architectures.

- We establish comparisons with some of the parallel algorithms work through a *provably optimal* implementation of the BVRAM on butterfly networks.

- We maintain a measure of independence even in our targeting of practical architectures by using the LogP model. Here we also prove some *optimality* results.

- We have tested the feasibility of our entire approach to compilation by running some *experiments* on a LogP simulator. Their goal is to compare, for both speedup and scaleup, the three components of the total running time: the data communication cost, the control communication cost, and the local computations.

In what follows we discuss some of the details of the technical contributions.

**Declarative complexity** Traditionally, declarative query languages do not carry explicit information about the complexity of a query. The query cost is of concern only to the optimizer who has to choose between several physical execution plans with different costs. In CoPa however the parallel complexities are as much part of the specification of the language as its operational semantics is, and therefore complexity is as declarative as the queries are. This is new and it allows us to provide formal proofs that our compilation techniques preserve complexity. One of the goals of this work therefore is to show that the high-level, declarative complexity can be actually achieved in implementation, even for a language as complex as CoPa.

In guaranteeing the preservation of complexity, we focus on the three main architecture-independent compilation phases: the representation of sets and bags as sequences, the elimination of recursion, and the flattening of nested parallelism. We are especially concerned with guaranteeing the complexity of the resulting flat program, such as to make sure that no potential for parallelism is lost in the flattening process. By choosing as compilation target flat sequences we forfeit the possibility of using existing parallel algorithms for the relational algebra operators. We made this choice for two reasons: we wanted our techniques to work uniformly across all kinds of collections, including lists and arrays, and we wanted to offer complexity guarantees, even in the presence of data skew.

**Recursive functions** Declarative query languages are limited in expressive power. For instance the relational algebra cannot express transitive closure. This limitation remains true for extensions of the relational algebra to nested relations [55, 78]. To increase computing power, applications use a query language like SQL *embedded* in a general-purpose programming language. There are limitations on the data structures that can be exchanged between the two languages (the "impedance mismatch" problem) as well as limitations in optimization opportunities (one SQL query at a time). We are interested in the design of a powerful language in which "queries" like, say, those in OQL [20] are just program phrases, and can be combined with a variety of parallel algorithm implementations, while maintaining the goal of complexity-preserving compilation. For that we introduced a special recursion schema in CoPa, called map-recursion. Note that map-recursion is stronger than traditional extensions of the relational algebra with fixpoints or DATALOG-recursion, because map-recursion makes our language Turing-complete, while fixpoint-like extensions are usually limited in expressive power. Nonetheless, we show that map-recursion can be eliminated while preserving complexity.

**The BVRAM and the Sequence Algebra** As a target for compilation, we designed a parallel vector machine model, called BVRAM (Bounded Vector Random Access Machine). The BVRAM consists of a fixed number of *vector registers*, each holding a flat sequence (i.e. a sequence of atomic values), and performs sequence instructions on these registers. Each instruction is "parallel", in the sense that it operates in parallel on the elements of the input sequence(s).

The BVRAM instructions form the basis for a new algebra for flat sequences, which we call the *Sequence Algebra*, SA for which we also provide declarative parallel complexities. We believe this algebra to be of independent interest, since it is a natural extension of the relational algebra from sets to sequences. Such extensions have been considered before for bags [49, 35], but the order inherent in sequences seems to be harder to deal with. Typical properties considered for the set or bag algebras are inexpressibility results like "transitive closure is not expressible in the algebra" [49]. Here we prove a new kind of inexpressibility results for SA: we give lower bounds on the complexity of certain sequence operations (sequence reversal and matrix transposition) expressible in SA. Moreover, we have found that factoring the compilation to the BVRAM through the SA provides a clearer specification of the translation and better proofs of complexity preservation.

**Monotone communications** For the architecture-dependent step we target two parallel architectures: a butterfly network, and the LogP model. We consider the butterfly network for theoretical purposes only. We show that all BVRAM instructions admit an optimal implementation on the butterfly network. As a result one can use CoPa and its parallel complexities $T, W$ as a high-level formalism to express algorithms for the butterfly network, and derive their low-level parallel complexity from $T, W$.

The LogP model was proposed by Culler et al. [22] as a way to model more accurately existing parallel shared-nothing architectures. The model consists of a fixed number $P$ of nodes, each node being a processor with local memory. The nodes communicate via a network, and the model focuses on the communication cost, which it captures by three parameters: the latency $L$, the overhead $o$, and the gap $g$ (hence LogP). The focus in the LogP model lies in programming parallel algorithms such that the communications do not interfere with each other. Karp et al. [42] proved some of the first results about the LogP model, namely that some simple parallel algorithms, like broadcast, summation, and prefix-scan, can be implemented in optimal parallel time [42]: here "optimal" means absolute minimum running time, not within a constant factor. In all problems considered in [42] the size of the input data depends only on $P$, the number of processors. In our work we extend this list of results by proving that *monotone communications* admit an optimal implementation in the LogP model. Thus all BVRAM instructions, except one, admit optimal implementations in the LogP model.

In some sense this result differs from the previous ones in that here the data size is much larger than the number $P$ of processors and dominates the parallel running time. In a monotone communication problem each of the $P$ nodes holds a sequence of data items, and each item has to be sent to some other node(s) (possible 0): the communication is *monotone* if the items do not change their order in the global sequence. We give an absolute optimal algorithm for the one-to-one monotone communication. In a one-to-many monotone communication the same algorithm needs to be followed by a parallel broadcast step: we can only prove that it is optimal up to the additive cost of one broadcast operation. We call this algorithm *near-optimal*, since the communication time here is dominated by the size of the data, while the additive cost depends only on $P$.

**Some experiments** Finally, we tested the feasibility of these implementation techniques by running some simple experiments on a LogP simulator. The goal of the experiments is to compare three components of the total running time: the data communication cost, the control communication cost, and the local computations. These comparisons are important, since our implementation techniques achieve good processor balance basically at the expense of massive data communications. The results show that for medium or large data sets, and for a number of nodes in the range $1 - 64$, the data communication costs are low enough, and hence are a price worthwhile paying for guaranteed processor balance. But as the number of nodes grows beyond some threshold, the cost of the control messages exceeds the total communication cost, making communications too expensive.

## 1.1 An Example

We start with an example illustrating of the features of a parallel language for collections, like nested parallelism, user-defined functions, and monotone communications. Consider a relation $Stores(name, sales)$ storing the name and a set of sale transactions for each store, where each sale transaction consists of an item name and a price. An object-oriented or object-relational system would allow us to declare such data as:

class $sale$
{ Float $price$;
String $item$;
};
class $store$
{ String $name$;
Bag$<$ $sale$ $>$ $sales$
};
val $Stores$:Bag$<$ $store$ $>$

Now consider the following query $Q$:

$$\begin{aligned}\text{fun } Q(stores) &= \text{select } f(x.name, x.sales)\\ &\quad\ \text{from } x \text{ in } stores\\ &\quad\ \text{where } p(x.name)\end{aligned}$$

which applies (in parallel) the function $f$ to each *store*, $x$, satisfying the predicate $p$, and returns the bag of results. Suppose now that the function $f(n, s)$ in turn applies some other function, $g$, in parallel to all *sales* of $x$, say $f(n, s)$ is:

$$\begin{aligned}\text{fun } f(n, s) &= \text{select } g(n, y)\\ &\quad\ \text{from } y \text{ in } s\end{aligned}$$

Then $Q$ has *nested parallelism*. Note that the user-defined function $g(n, y)$ has access both to the name $n$ of a store, and to one of its sale transactions $y$. We have used here CoPa syntax to specify the query, but queries are expressible with minor syntax changes in virtually all languages proposed for object-oriented or object-relational databases, like OQL (ODMG [20]) and an SQL3-related language [63].

Let us take a close look at a parallel evaluation of the query $Q$. Of course, we start by applying $p$ in parallel to all stores: here we would like to have the stores uniformly distributed on the nodes, in order to get good load balance. Next we want to apply $f$ in parallel to all selected stores: to ensure good load balance, we redistribute first the selected stores on the nodes. But now we face a dilemma, due to the fact that stores are nested collections: should we distribute the stores uniformly, or rather their sales ? This dilemma is typical for nested collection, and for the associated nested parallelism.

Our solution is based on a flattening technique using *segment descriptors* [12, 7]. It only works for sequences, not for sets or bags, so a part of this paper deals with encodings of (nested) sets and bags in terms of sequences. To illustrate the technique, suppose that our database has 6 stores, with $5, 6, 4, 0, 1, 2$ sales respectively (for a total of 18). First we split *stores* into a flat sequence *names*, and a nested sequence *sales*:

$$sales = [[s_{00}, s_{01}, s_{02}, s_{03}, s_{04}], [s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}], [s_{20}, s_{21}, s_{22}, s_{23}], [], [s_{40}], [s_{50}, s_{51}]]$$

The nested sequence *sales* is further split into two flat sequences $s$ and $ss$:

$$\begin{aligned} ss &= [5, 6, 4, 0, 1, 2]\\ s &= [s_{00}, s_{01}, \ldots, s_{51}]\end{aligned}$$

The numbers in $ss$ are the lengths of the subsequences in *sales* and are called segment descriptors. Finally, once the whole database is represented only with flat sequences, we can distribute each of the sequences uniformly on the nodes. Figure 2 illustrates this for 3 nodes.

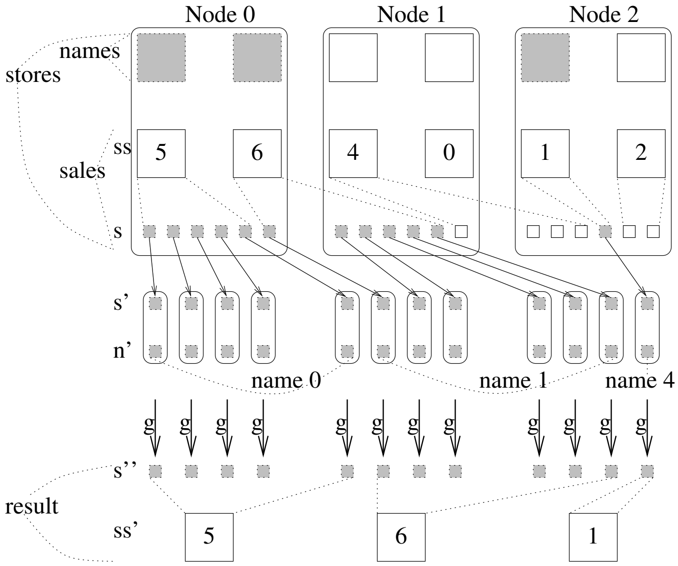


Figure 2: Implementation of the query $Q$.

Now we can explain how the query $Q$ can be computed on this data representation, step by step:

1. Apply the predicate $p$ to each store. Assume that $p$ returns true on stores $0, 1, 4$ (those shaded in Figure 2).
2. Send a 0 or 1 flag to every element in $s$, saying whether its store has been selected or not: this step involves data communication. The resulting subsequence of $s$, call it $s'$, has 12 elements (which are shaded in Figure 2).
3. Redistribute $s'$ uniformly on the 3 nodes.
4. We cannot apply $g$ on the elements in $s'$ yet, because $g$ needs access to the name of the store too. For that, replicate the names of the selected stores: the resulting sequence $n'$ contains the names of stores 0, 1, and 4, with multiplicities 5, 6, and 1 respectively.
5. Now apply $g$ on each name-sale pair from $n'$ and $s'$. This is again done locally, with perfect load balance. Call $s''$ the sequence of 12 results.
6. Let $ss'$ be the subsequence of $ss$ corresponding to the selected stores: $ss' = [5, 6, 1]$. Distribute $ss'$ uniformly on the nodes.
7. The final result is a nested sequence represented by the two sequences $s''$ and $ss'$.

Each step is a perfectly balanced parallel computation, but the price paid for that is that of frequent data communications. Here the insight is that all these communications are of a particular, simple form: they are *monotone* communications. In this paper we describe an optimal algorithm for monotone communications.

In this example, a key step consisted in expressing nested collections in terms of flat sequences. An interesting question is whether this generalizes to more complex queries on nested collections. We show that this is the case, by identifying a minimal set of flat sequence operations which can support the nested queries. These are simple enough to parallelize, and yet sufficient to implement "efficiently" OQL-style queries on nested collections.

## 1.2 Related Work

**Bubba,FAD and SVP** Bancilhon et al. [5] describe a database language FAD, designed for a parallel database machine Bubba. FAD is a functional language, featuring nested sets with object identifiers and two main parallel constructs: a parallel map (called **filter**), and a parallel divide and conquer construct (called **pump**). Parker et al. [56] describe a value-based model of parallelism in bulk data processing, SVP, in which both sets and streams are represented as trees. It has a complex parallel operator, called *transducer*, generalizing parallel map, parallel divide-and-conquer, and parallel stream processing. Neither FAD nor SVP have explicit sequence operations.

**Query languages for collections** Many of the collection operations we consider are inspired by the theory of *monads* [50]. The idea that monads could be used to organize semantics of programming constructs is due to Moggi [52]. Wadler [75] showed that they are also useful in organizing syntax, in particular they explain the "list-comprehension" syntax of functional programming. Moreover Trinder and Wadler [71] showed that an extension of comprehension can implement the (flat) relational calculus. Trinder and Watt [70, 76], have also sought after a uniform algebra for several different collection types. This was followed by work at Penn [14, 15, 77, 17, 48, 47, 68, 67] and elsewhere [29].

**NESL** NESL is a powerful, general-purpose parallel functional languages designed by Guy Blelloch [8], which comes with a high-level definition of the parallel complexity. It is compiled into a Vector Random Access Machine (VRAM) by flattening nested parallelism and using segment descriptors (these techniques are described in earlier work [12, 7]). Our flattening technique uses the same segment descriptors for encoding nested sequences in terms of flat sequences. The complexity-preserving compilation improves over that of NESL in several ways: first it does not need a stack of sequences, it is more economical in the communication primitives needed, and improves the complexity of while-iterations. Moreover, we are able to guarantee complexity of any compiled program, by a combination of language design (CoPa) and compilation techniques, while in NESL the preservation holds only for "contained" programs (see Sec. 4.3 for a more detailed discussion).

**LogP** We base our communication cost analysis on the LogP model, proposed by Culler et al. [22]. While precise enough to model virtually all modern multiprocessors [22, 2, 9, 72], the LogP model is simple enough to serve as a framework for the design and analysis of parallel algorithms [42]. Our optimal algorithm for monotone communication is for the LogP model.

**Our previous work** Some results in this work have been previously announced in [66] and [64]. Moreover, this work can be seen as a materialization of the theoretical result in [65, 67]. Indeed, we give there a characterization of NC in terms of a query language for sets featuring the monad constructs seen also here and *divide-and-conquer recursion* (dcr) and it is easy to see that dcr admits an NC implementation with map-recursion.

## 1.3 Overview

This paper is organized as follows. We describe CoPa in Sec. 2 and give some examples. CoPa's declarative complexities are described in Sec. 3. Sec. 4 describes the architecture-independent part of the compilation: Sec. 4.1 shows the encoding of sets and bags in terms of sorted sequences, Sec. 4.2

shows how to eliminate recursion, and Sec. 4.3, 4.4 show the actual compilation on a BVRAM. As an application of the complexity-preserving compilation, Sec. 4.5 gives lower bound on the cost of certain permutations expressed in CoPa. We describe architecture-dependent implementations in Sec. 5: implementation on the butterfly network is discussed in 5.1, and on the LogP model in 5.2. In Sec. 5.3 we give optimal and near-optimal LogP algorithms for the one-to-one and one-to-many monotone communications, and in Sec. 5.4 we report on some LogP simulation experiments. We conclude in Sec. 6

# 2 The Specification of CoPa

We present here our language CoPa. The language is inspired by recent developments in query language [25, 17, 20] and adds a novel kind of recursion, map-recursion. It is a monomorphic functional language, with emphasis on operations on collections, their combinations, and user-defined external functions. The language is first-order in its treatment of functions (functions cannot be arguments or results of functions) but it is higher-order in its treatment of collections allowing for arbitrary nesting. Its restricted form of recursion is actually quite general. The only major restriction in CoPa is that it does not have object references or pointers. Indeed, these features are meant to enable state and side-effects and their presence would completely change the problem.

Since we intend to formally prove that our compilation technique preserves complexity, we have an interest in controlling the size of the formalism. Therefore, CoPa is specified by a relatively small number of essential constructs, and we use additional syntactic sugar in examples that demonstrate the practicality of the language.

## 2.1 Language Definition

**Types** CoPa is *strongly typed.* There are two kinds of types: the types of (complex) values, and the types of functions. The latter play a rather auxiliary role and could be avoided altogether at the price of a heavier syntax for expressions: all functions are first-order. The value types are constructed from a set of base types by combinations of *product type*, *sum type*, and the three *collection type constructors*: sets, bags, and sequences. To simplify the presentation we consider tuples instead of records: of course, records can be treated similarly, without major changes. The base types are denoted by $\mathbb{D}_1, \mathbb{D}_2, \ldots$ and include types like *string*, *real*, *int*, $\mathbb{N}$ as well as user-defined types. We view objects of base types as black boxes, in that we do not attempt to parallelize operations on such values.

$$\text{(complex value types)} \qquad t ::= \mathbb{D} \mid \ldots \mid t \times \ldots \times t \mid t + \ldots + t \mid \{t\} \mid \{\!|t|\!\} \mid [t]$$

$$\text{(function types)} \qquad u ::= t_1 \to t_2$$

$\mathbb{D}$ stands for some arbitrary domain of the base type (one of $\mathbb{D}_i$, $i \geq 1$). As usual, $t_1 \times \ldots \times t_n$ denotes the **product type** and contains all $n$-tuples $\langle x_1, \ldots, x_n \rangle$ where $x_1 \in t_1, \ldots, x_n \in t_n$. Dually, $t_1 + \ldots + t_n$, is the **sum (disjoint union) type** of $t_1, \ldots, t_n$; its values are of the form $in_1(x_1)$, or $in_2(x_2)$, ..., or $in_n(x_n)$, with $x_1 \in t_1, \ldots, x_n \in t_n$.

We denote with *unit* the empty product type (obtained by taking $n = 0$ in $t_1 \times \ldots \times t_n$) with a single value, the empty tuple $\langle\rangle$. We define the **boolean** type $\mathbb{B} \stackrel{\text{def}}{=} unit + unit$, and identify its values $in_1\langle\rangle$ and $in_2\langle\rangle$ with true and false respectively. We could take $n = 0$ in a disjoint sum type $t_1 + \ldots + t_n$ too: the resulting type has no values at all, and is of no interest for us in the sequel.

| Expression | Type | Meaning |
|---|---|---|
| $x$ | $t$ | variable |
| $p$ | $d_p \to c_p$ | external function $p \in \Sigma$ |
| $e_1 = e_2$ | $\mathbb{B}$ | only for $e_1, e_2$ of base type |
| $\langle e_1, \ldots, e_n \rangle$ | $t_1 \times \ldots \times t_n$ | tuple |
| $\pi_i^{t_1 \times \ldots \times t_n}(e)$ | $t_i$ | projection |
| $in_i^{t_1 + \ldots + t_n}(e)$ | $t_1 + \ldots + t_n$ | injection |
| case $e$ of<br>$in(x_1) \Rightarrow e_1 \mid$<br>$\ldots$<br>$in(x_n) \Rightarrow e_n$ | $t$ | See text |
| $\lambda x : t_1.e$ | $t_1 \to t_2$ | $\lambda$-abstraction |
| $f(e)$ | $t_2$ | function application |

Figure 3: General-purpose operators in CoPa.

The interesting type constructs in our languages are the three collection types, sets, bags, and sequences. They differ in how they treat multiplicities and order. Thus, in a *set* both multiplicities and order are irrelevant: $\{1, 2, 2, 4, 6\}$ is the same as $\{1, 2, 4, 6\}$ and as $\{6, 2, 1, 4\}$. In *bags* multiplicities matter: $\{\!|1, 2, 2, 4, 6|\!\}$ is different from $\{\!|1, 2, 4, 6|\!\}$. Finally, sequences are lists: both order and multiplicities matter, e.g. $[1, 2, 4, 6] \neq [6, 2, 1, 4]$. Since some operations are similar across all three collection types, we consider some limited form of "collection polymorphism", in that we may define functions which work uniformly across all three collections. We will use the notation $\lceil t \rfloor$ for any of the three collection types $\{t\}$, $\{\!|t|\!\}$, or $[t]$. There is nothing deep here: our collection polymorphism is just a convenient way to avoid repeating the same notation three times.

**Language constructs** The constructs (expressions) of CoPa are listed in Figures 3, 4 and 5. A complete definition of the language is in Appendix A. As usual, we distinguish between *free* and *bound* variables: (eg. $x$ becomes bound in $\lambda x{:}t.e$). As we explained earlier, this language is first-order from a functional point of view, hence bound variables have value type and only range over (complex) values. The language is parameterized by a set of *external functions* $p : d_p \to c_p$ and *constants* $c : t_c$. Examples of constants are $0, 1, 2, \ldots$, "$abc$", ... External functions play a major role in CoPa programs. All external functions are sequential and without sideffects: CoPa will not attempt to parallelize them. The simplest external functions are *scalar functions*, whose domain and codomain are base types, like $+, -, *, /$. Next we have *aggregates*, like $\mathsf{max} : \{\!|\mathbb{N}|\!\} \to \mathbb{N}$, $\mathsf{sum} : \{\!|\mathbb{N}|\!\} \to \mathbb{N}$: in general, an aggregate function has type $agg : \lceil s \rfloor \to s'$, where $s, s'$ are base types. More complex, any user-defined function, of any type, can be an external function. We do not attempt to parallelize external functions: as a consequence aggregates are performed sequentially.

CoPa follows the design principles of [17] and considers product, sum, and collection types as orthogonal. Hence, there are are constructs associated to the product types, and others associated to the sum types (Fig. 3) and finally others associated to collections (Fig. 4). Recursion and iteration are in Fig. 5.

The meaning of product and sum CoPa expressions is standard: we briefly describe them for completeness: $\langle e_1, \ldots, e_n \rangle$ denotes an $n$-tuple, $\pi_i^{t_1 \times \ldots \times t_n}$, for $i = 1, n$, are the projections: $\pi_i^{t_1 \times \ldots \times t_n} \langle x_1, \ldots, x_n \rangle \stackrel{\text{def}}{=} x_i$. For the sum type, $in_i(e)$ denotes the value $in_i(v)$, where $v$ is $e$'s value. The meaning of case $e$ of $in_1(x_1) \Rightarrow e_1 \mid \ldots$ is: if the meaning of $e$ is $in_i(v)$, then the meaning of the case ex-

| Expression | Type | Meaning |
|---|---|---|
| $\lceil\rfloor$ | $\lceil t\rfloor$ | empty collection |
| $\lceil e\rfloor$ | $\lceil t\rfloor$ | singleton collection |
| $e_1 \cup e_2$ | $\{t\}$ | set union |
| $e_1 \oplus e_2$ | $\{\!\mid t\mid\!\}$ | bag addition |
| $e_1 @ e_2$ | $[t]$ | sequence concatenation |
| $\mathsf{ext}(f)$ | $\lceil t_1\rfloor \rightarrow \lceil t_2\rfloor$ | extension (see text) |
| $\mathsf{map}(f)$ | $\lceil t_1\rfloor \rightarrow \lceil t_2\rfloor$ | map (see text) |
| flatten | $\lceil\lceil t\rfloor\rfloor \rightarrow \lceil t\rfloor$ | flattening nested collections |
| length | $\lceil t\rfloor \rightarrow \mathbb{N}$ | number of elements |
| get | $\lceil t\rfloor \rightarrow t$ | get single element (see text) |
| set2bag | $\{t\} \rightarrow \{\!\mid t\mid\!\}$ | |
| $\mathsf{bag2seq}(p)$ | $\{\!\mid t\mid\!\} \rightarrow [t]$ | $p : t \times t \rightarrow \mathbb{B}$ is an order predicate |
| seq2bag | $[t] \rightarrow \{\!\mid t\mid\!\}$ | |
| bag2set | $\{\!\mid t\mid\!\} \rightarrow \{t\}$ | |
| zip | $[t_1] \times [t_2] \rightarrow [t_1 \times t_2]$ | sequence zip (see text) |
| enumerate | $[t] \rightarrow [\mathbb{N}]$ | enumerates a sequence |
| split | $[t] \times [\mathbb{N}] \rightarrow [[t]]$ | inverse of flatten (see text) |

Figure 4: Collection-oriented operators in CoPa.

| Expression | Type | Meaning |
|---|---|---|
| $\mathsf{while}(p, f)$ | $t \rightarrow t$ | iteration, where $p : t \rightarrow \mathbb{B}$, $f : t \rightarrow t$ |
| $\mathsf{maprec}(d, c)$ | $t \rightarrow t'$ | map-recursion, where $d : t \rightarrow [t]$, $c : t \times [t'] \rightarrow t'$ |

Figure 5: Iteration and recursion in CoPa.

pression is that of $e_i[v/x_i]$. The type of the case-statement is $t$, provided that $e : t_1 + \ldots + t_n$, $x_1 : t_1, \ldots, x_n : t_n$, and $e_1 : t, \ldots, e_n : t$: see Appendix A for a complete definition. An if-statement, if $b$ then $e_1$ else $e_2$, where $b : \mathbb{B}$, can be expressed as case $b$ of $in_1(x_1) \Rightarrow e_1 \mid in_2(x_2) \Rightarrow e_2$. The lambda expression $\lambda x{:}t.e$ denotes a function in which $x$ is the input variable, while the construct $f(e)$ is function application.

For collections, $\lceil\rfloor$ is the empty collection, $\lceil e \rfloor$ denotes the singleton collection, and $e_1 \cup e_2$, $e_1 \oplus e_2$, $e_1 @ e_2$ are the set union, bag addition, and sequence concatenation respectively. In the context of a generic collection we write @ for all three operations, i.e. $@ : \lceil t \rfloor \times \lceil t \rfloor \to \lceil t \rfloor$. The "extension" of a function $f : t_1 \to \lceil t \rfloor$ is defined to be:

$$\mathsf{ext}(f)(\lceil x_0, \ldots, x_{n-1} \rfloor) \stackrel{\text{def}}{=} f(x_0) @ \ldots @ f(x_{n-1})$$

Map is defined for any function $f : t_1 \to t_2$:

$$\mathsf{map}(f)(\lceil x_0, \ldots, x_{n-1} \rfloor) \stackrel{\text{def}}{=} \lceil f(x_0), \ldots, f(x_{n-1}) \rfloor$$

while flatten "flattens" a nested collection in an obvious way. Note that ext can express map and flatten and conversely. These two alternatives of lifting functions to collections can be traced back to two alternative ways of defining monads: we refer the reader to [17] for a discussion of the relationship between collection languages and monads. Figure 6 proves the equivalence of $\mathsf{ext}(-)$ to $\mathsf{map}(-)$ and flatten. When proving properties about CoPa sometimes it is more conveniet to work with $\mathsf{ext}(-)$, other times with $\mathsf{map}(-)$ and flatten, and we will feel free to switch back and forth between the two presentations.

The operation get attempts to retrieve the unique element of a collection, and returns an error if that collection is not a singleton: $\mathsf{get}(\lceil x \rfloor) = x$. Next, zip and enumerate are defined by:

$$\begin{aligned}
\mathsf{zip}\langle [x_0, \ldots, x_{n-1}], [y_0, \ldots, y_{n-1}] \rangle &\stackrel{\text{def}}{=} [\langle x_0, y_0 \rangle, \ldots, \langle x_{n-1}, y_{n-1} \rangle] \\
\mathsf{enumerate}([x_0, \ldots, x_{n-1}]) &\stackrel{\text{def}}{=} [0, \ldots, n-1]
\end{aligned}$$

Here zip returns an error when the two sequences have different lengths. The next function, $\mathsf{split}(x, y)$, is a kind of an inverse to flatten. It expects $y$ to be a sequence of integers whose sum equals the length of $x$; then $\mathsf{split}(x, y)$ splits $x$ into a number of subsequences, whose lengths is given by the numbers in $y$. E.g. $\mathsf{split}([a, b, c, d, e], [2, 0, 3]) = [[a, b], [], [c, d, e]]$.

Finally, $\mathsf{while}(p, f)(x)$ iteratively computes $x := f(x)$ until $p(x)$ becomes $false$, then returns $x$: formally, $\mathsf{while}(p, f)(x) = f^{(k)}(x)$ where $k \geq 0$ is the smallest number s.t. $p(f^{(k)}(x)) = \mathsf{false}$. The map-recursion $\mathsf{maprec}(c, d)$ is discussed in Subsection 2.3 below.

We shall drop the type superscripts and the types in the $\lambda$-abstractions whenever they can be deduced from the context, and abbreviate $\lceil x_0 \rfloor @ \ldots, @ \lceil x_{n-1} \rfloor$ with $\lceil x_0, \ldots, x_{n-1} \rfloor$. Also we will freely use pattern matching in $\lambda$-expressions, like in the function $swap : t_1 \times t_2 \to t_2 \times t_1$ which we write as $swap \stackrel{\text{def}}{=} \lambda \langle x_1{:}t_1, x_2{:}t_2 \rangle . \langle x_2, x_1 \rangle$, or even $\lambda \langle x_1, x_2 \rangle . \langle x_2, x_1 \rangle$, instead of the official $\lambda z{:}t_1 \times t_2 . \langle \pi_2^{t_1 \times t_2}(z), \pi_1^{t_1 \times t_2}(z) \rangle$.

**Typing Rules** There are no novelties in CoPa's typing rules. But the typing rules are important in CoPa because they are associated with its parallel operational semantics and parallel complexities, to be discussed later, hence we briefly describe them here: the complete listing of the rules is given in Appendix A. The rules use contexts to keep track of the variables' types. A *context* is a set $\Gamma = \{x_1 : t_1, \ldots, x_n : t_n\}$, where $x_1, \ldots, x_n$ are distinct variables and $t_1, \ldots, t_n$ are types. Strictly

$$\begin{aligned}
\mathsf{map}(f) &= \mathsf{ext}(\lambda x{:}t_1.[f(x)]) \\
\mathsf{flatten} &= \mathsf{ext}(\lambda x{:}[t].x) \\
\mathsf{ext}(f)(x) &= \mathsf{flatten}(\mathsf{map}(f)(x))
\end{aligned}$$

Figure 6: Translations from $\mathsf{ext}(-)$ to $\mathsf{map}(-)$ and $\mathsf{flatten}$.

speaking, expressions in CoPa are of two tiers: *value* expressions, $e : t$, and *function* expressions $f : t_1 \to t_2$. Thus, the rules in Appendix A have consequences of either the form $\Gamma \rhd e : t$, denoting the fact that "expression $e$ has type $t$ under the context $\Gamma$, or of the form $\Gamma \rhd f : t_1 \to t_2$. For example, the rules for $\langle e_1, \ldots, e_n \rangle$ and $\lambda$-expressions are:

$$\frac{\Gamma \rhd e_1 : t_1, \ldots, \Gamma \rhd e_n : t_n}{\Gamma \rhd \langle e_1, \ldots, e_n \rangle : t_1 \times \ldots \times t_n} \qquad \frac{x : t_1, \Gamma \rhd e : t_2}{\Gamma \rhd \lambda x{:}t_1.e : t_1 \to t_2}$$

A somewhat special rule which will become important in the context of parallel evaluation is the *weakening* rule:

$$\frac{\Gamma \rhd e : t_2}{x : t_1, \Gamma \rhd e : t_2}$$

It allows us to add unused variables to the context $\Gamma$. This rule is needed to typecheck certain CoPa expressions. For example the rule for $\langle e_1, \ldots, e_n \rangle$ requires that all of $e_1, \ldots, e_n$ have the same context while, in fact, not all of them need to use exactly the same variables: we could "forget" some of the unused variables in $e_i$ in the typing derivation for that expression. Applying the weakening rule at different places yields different correct type derivations for the same CoPa expression. As we shall see, these may correspond to different parallel evaluations, and different parallel complexities, because the values of free variables have to be broadcast to the parallel threads.

Due to our interest in database applications, a closed (i.e. with no free variables) expression of function type $f : t_1 \to t_2$ will often be called a **query**.

**Example 2.1** For some simple examples, consider the three functions below, which we shall need in the sequel:

| Function | Definition | Comments |
|---|---|---|
| $\Pi_i : \lceil t_1 \times \ldots \times t_i \rfloor \to \lceil t_i \rfloor$ | $\mathsf{map}(\pi_i)$ | database projections $(i = 1, n)$ |
| $\mathsf{filter}(p) : [t] \to [t]$ | $\mathsf{ext}(\lambda x.\mathsf{if}\ \ p(x)\ \ \mathsf{then}\ \ [x]\ \ \mathsf{else}\ \ [])$ | $p : t \to \mathbb{B}$ is some predicate |
| $\mathsf{select} : [\mathbb{B} \times t] \to [t]$ | $\Pi_2(\mathsf{filter}(\lambda\langle b, \_\rangle.b)(x))$ | |

□

## 2.2 Making CoPa Practical: Syntactic Sugar

We extend the core CoPa with some concrete syntax, including:

- An ML-style let -expressions, allowing us to give names to values and functions.
- A select − from −where block, like in various query languages (SQL, O2, OQL, etc).
- Records, as opposed to tuples.

The extensions are merely syntactic sugar: we show how they can be easily compiled-away.

The ML-style let -expressions are:

$$\begin{array}{ll} \mathsf{let\ val}\ x : t = e & \mathsf{let\ fun}\ f(x : t) = e \\ \mathsf{in}\ e'\ \mathsf{end} & \mathsf{in}\ e'\ \mathsf{end} \end{array}$$

The first is equivalent to $(\lambda \mathtt{x} : \mathtt{t}.\mathtt{e}')(\mathtt{e})$, the second to $\mathtt{e}'[(\lambda \mathtt{x} : \mathtt{t}.\mathtt{e})/\mathtt{f}]$. We allow several `val` and `fun` declarations in the same `let`, and allow patterns in function arguments, like $\mathtt{fun\ f}(\mathtt{x} : \mathtt{t}, \mathtt{y} : \mathtt{t}', \mathtt{z} : \mathtt{t}'') = \mathtt{e}$ (meaning $\mathtt{fun\ f}(\mathtt{u} : \mathtt{t} \times \mathtt{t}' \times \mathtt{t}'') = \mathtt{e}[\Pi_1(\mathtt{u})/\mathtt{x}, \Pi_2(\mathtt{u})/\mathtt{y}, \Pi_3(\mathtt{u})/\mathtt{z}]$).

The select − from −where construct is:

$$\begin{array}{l} \mathsf{select}\ e \\ \mathsf{from}\ x_1\ \mathsf{in}\ e_1, \ldots, x_k\ \mathsf{in}\ e_k \\ \mathsf{where}\ C_1, \ldots, C_l \end{array}$$

Here $e_1, \ldots, e_k$ are collection expressions of the same kind (i.e. all are sets, or bags, or sequences), and $C_1, \ldots, C_l$ are boolean conditions. We drop the from- or the where-clause when either is empty ($k = 0$ or $l = 0$). Following [16] this is translated by the following two rules:

- When $k = 0$:

$$\left.\begin{array}{l} \mathsf{select}\ e \\ \mathsf{where}\ C_1, \ldots, C_l \end{array}\right\} \Longrightarrow \mathsf{if}\ C_1\ \mathsf{and} \ldots \mathsf{and}\ C_l\ \mathsf{then}\ \lceil e \rfloor\ \mathsf{else}\ \lceil \rfloor$$

- When $k > 0$:

$$\left.\begin{array}{l} \mathsf{select}\ e \\ \mathsf{from}\ x_1\ \mathsf{in}\ e_1, \ldots, x_k\ \mathsf{in}\ e_k \\ \mathsf{where}\ C_1, \ldots, C_l \end{array}\right\} \Longrightarrow \left\{\begin{array}{lll} \mathsf{let} & \mathsf{fun}\ f(x_1) = & \mathsf{select}\ e \\ & & \mathsf{from}\ x_2\ \mathsf{in}\ e_2, \ldots, x_k\ \mathsf{in}\ e_k \\ & & \mathsf{where}\ C_1, \ldots, C_l \\ \mathsf{in} & \mathsf{ext}(f)(e_1) & \\ \mathsf{end} & & \end{array}\right.$$

**Example 2.2** The function $distribute_{left} : t_1 \times \lceil t_2 \rfloor \to \lceil t_1 \times t_2 \rfloor$, with the meaning:

$$distribute_{left}\langle x, \lceil y_1, \ldots, y_n \rfloor \rangle \stackrel{\text{def}}{=} \lceil \langle x, y_1 \rangle, \ldots, \langle x, y_n \rangle \rfloor$$

is defined as:

```
fun distribute_left(x:t1, Y:[t2]) =
    select <x,y>
    from y in Y
```

Without syntactic sugar: $distribute_{left} = \lambda \langle x{:}t_1, Y{:}[t_2] \rangle.(\mathsf{ext}(\lambda y{:}t_2.\langle x, y \rangle)(Y))$. Note the essential role of the free variable $x$ in the function $\lambda y{:}t_2.\langle x, y \rangle$. Also, $distribute_{right} : [t_1] \times t_2 \to [t_1 \times t_2]$ can be defined similarly. □

**Example 2.3** We illustrate now a more realistic example in CoPa. Consider a set of customers each consisting of a `Name` and some `Orders`. The orders form a sequence, and have attributes `ItemNo`, `Qty`, and `Desc`: the latter is a free-text description of the order. The types can be defined by:

```
class Customer {                    class Order {
  Name : string[30];                  ItemNo : int;
  Orders : Sequence<Order>            Qty : int;
}                                     Description : description
                                    }
```

Due to some processing errors, some customers had to place orders twice. Such orders always have the same `ItemNo`, are are known to have been placed in sequence, but they may differ in quantities. We can tell if they are duplicate orders only by looking at the free-text `Description`. A user-defined function `related` takes two descriptions and tells if they are duplicate orders: it returns an integer, in the range 0..100, describing the likelihood of them being duplicates. We want to retrieve all customers who had duplicate orders with likelihood $> 80\%$. This is achieved by:

```
fun FindCustomer(CS : set<Customer>) =
  let fun P(C : Customer) =
        let val ord = C.Orders
            val n = length(ord)
            val ord1 = ord[0..n-2]
            val ord2 = ord[1..n-1]
            varl r = select related(o1.Description, o2.Description)
                     from <o1,o2> in zip(ord1,ord2)
                     where o1.ItemNo = o2.ItemNo
        in if length(r) > 0 then max(r)
           else 0
        end
  in  select C.Name
      from C in CS
      where P(C) > 80
  end
```

Here `ord[i..j]` is another shorthand for extracting a subsequence from a sequence: it can be expresses in CoPa as:

$$\Pi_1(\mathsf{filter}(\lambda\langle x,k\rangle.i \leq k \text{ and } k \leq j)(\mathsf{zip}\langle ord, \mathsf{enumerate}(ord)\rangle))$$

□

## 2.3 Recursion in CoPa

General recursion in a parallel function language is difficult to compile. Blelloch and Sabot [12] and Blelloch [7] describe a compilation technique for a parallel, functional programming language. That technique however does not preserve the parallel time complexity $T$ for *every* program, but only for certain programs which are *contained*. A compiler cannot check whether a program is contained, because this is an undecidable property.

Here we adopt a different approach. We impose *syntactic* restrictions on the recursion schema which a programmer is allowed to use. A compiler would simply have to verify that the function definition looks like that:

$$\mathsf{fun}\quad f(x) = c(x, \mathsf{map}(f)(d(x)))$$

$$
\begin{array}{lcl}
\mathsf{fun}\ \ g(x) & = & \mathsf{if}\ \ p(x)\ \ \mathsf{then}\ \ s(x)\ \ \mathsf{else}\ \ c\langle g(d_1(x)), g(d_2(x))\rangle \\
\mathsf{fun}\ \ h(x) & = & \mathsf{if}\ \ p(x)\ \ \mathsf{then}\ \ s(x)\ \ \mathsf{else}\ \ c(h(d(x))) \\
\mathsf{fun}\ \ k(x) & = & \mathsf{if}\ \ p(x)\ \ \mathsf{then}\ \ s(x)\ \ \mathsf{else} \\
 & & \mathsf{if}\ \ p'(x)\ \ \mathsf{then}\ \ c\langle k(d_1(x)), k(d_2(x))\rangle\ \ \mathsf{else}\ \ c'\langle k(d_1'(x)), k(d_2'(x)), k(d_3'(x))\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fun}\ \ d'(x) = \mathsf{if}\ \ p(x)\ \ \mathsf{then}\ \ [] \\
\qquad\qquad\quad \mathsf{else}\ \ [d_1(x), d_2(x)] \\
\mathsf{fun}\ \ c'\langle x, []\rangle = s(x) \\
\qquad |c'\langle \_, [y_1, y_2]\rangle = c\langle y_1, y_2\rangle \\
\mathsf{fun}\ \ g(x) = \mathsf{maprec}(d', c')(x)
\end{array}
$$

Figure 7: Other recursive schemas and their translations into map-recursion

Without syntactic sugar $f$ would be written as $\mathsf{maprec}(d, c)$, and its type is $t \to t'$, provided that $d : t \to [t]$ and $c : t \times [t'] \to t'$. Intuitively, this is a generalized form of divide-and-conquer. To compute $f(x)$, one *divides* $x$ first by computing $d(x) = [x_0, \ldots, x_{n-1}]$, then applies $f$ on each subproblem, $[f(x_0), \ldots, f(x_{n-1})]$, and finally applies the *conquer* function $c$. Note that $c$ may refer to the original input $x$ in addition to the results of the $n$ subproblems.

Several recursive schemes are technically speaking not map-recursive, but can be rephrased easily as such. Figure 7 contains three examples of recursive definitions. The first, $g(x)$, is a 2-way divide-and-conquer, and its translation into a map-recursive scheme is shown. The function $h(x)$ is a tail-recursive one: here $n = 1$ and the translation into map-recursion is straightforward and omitted. Finally $k(x)$ illustrates a more complex divide-and-conquer recursion, in which the input $x$ is divided either into 2 subproblems or into 3: the translation into map-recursion is, again, straightforward and omitted.

An informal examination of a variety of parallel algorithms [40, 7] persuaded us that map-recursion suffices to express in a natural way large classes of parallel algorithms. It is easy to write a preprocessor which translates recursive schemas like those in figure Figure 7 into map-recursive definitions, and in the process "parallelizes" them.

One wonders which functions *cannot* be easily translated into map-recursion. Primitive recursion on sequences is a particular case of map-recrusion; in particular using unary or binary representation, all arithmetical primitive recursive functions are representable. It turns out however that the definition of Ackerman's function: $A(x, y) = A(x-1, A(x, y-1))$ exhibits a kind of recursion scheme which is not map-recursion: in order to express $A$ in CoPa one would have to express it as a while program first, using a sequnce for a stack. Another, more practical, such example is the linear-time median-finding algorithm [13].

# 3 Parallel Evaluation and Parallel Complexity

CoPa is a parallel language: this statement is made precise by the the definition of CoPa's *parallel complexity*, which is part of the language definition. Exposing the complexity at the language level is novel in the context of declarative query languages since complexity is traditionally related to *how* a query is computed, while declarativeness presumes that the query only specifies *what* is to be computed. However, CoPa's complexity is computation independent: it is just a number the language definition assigns to each expression, and does not prescribe how that expression is to be evaluated. We say that the complexity is *declarative*. The burden of ensuring that the complexity is indeed preserved at running time falls on the *complexity preserving* compilation. This section describes CoPa's parallel complexity and work complexity together with its operational semantics: a full definition is given in Appendix A. The parallel and work complexities are in the spirit of those of Blelloch and Sabot [12] and Blelloch [7] although here we give a completely formal definition with rules that accompany those of operational semantics. We use a *structured operational semantics*, in the style pioneered by [57] and refined by [41].

The parallel complexity is a function $T$ which, for each CoPa expression $e$, gives a number $T(e)$ defining the parallel running time of the expression $e$. The function $T$ is defined inductively, but not on the structure of the expression $e$, but on its evaluation tree: that is, $T$ is defined simultanuosly with the definition of CoPa's operational semantics. It is $T$ who defines parallelism in CoPa. For example for an expression $\mathsf{map}(f)([e_0, \ldots, e_{n-1}])$, $T$ is defined by:

$$T(\mathsf{map}(f)([e_0, \ldots, e_{n-1}])) \stackrel{\text{def}}{=} 1 + \max_{i=0,n-1} T(f(e_i)) \tag{1}$$

in effect stipulating that $\mathsf{map}(f)(x)$ has to be computed by applying "in parallel" $f$ on each element of the sequence $x$: the $1 + \ldots$ term allows the evaluation to do some pre- and postprocessing ($T$ is meaningful only up to a constant factor).

Similarly, consider a second complexity associated to each CoPa expression, called *work complexity*, $W(e)$, which is essentially the sequential complexity of that expression.

A complexity-preserving compilation has to ensure that $T, W$ relate to the running time on a parallel machine. Hence, the exact definition may depend on the target architecture. For example assuming a shared-nothing architecture with $P$ processors, the running time $T_P$ could be required to be:

$$T_P \quad = \quad O(T + \frac{W}{P} \log P)$$

Complete definitions of the operational semantics, the parallel time complexity and the work complexity are given in Appendix A, in the same table with CoPa's type inference rules. Here we highlight the basic principles underlying those definitions.

**Operational Semantics** The operational semantics is a set of rules telling how CoPa expressions evaluate to *constants*. The latter are given by the grammar:

$$C ::= d \mid \langle C, \ldots, C \rangle \mid in_i^{t_1 + \ldots + t_m}(C) \mid \lceil C, \ldots, C \rfloor$$

Here $d$ is some encoding for an object of the base type $\mathtt{D}$. We only consider well-typed constants. During the evaluation of some expression $e$, the free variables occurring in $e$ are bound to constants. The association between variables and constants is captured by an **environment** [23], a finite set of the form $\rho = \{x_1 = C_1, \ldots, x_n = C_n\}$, where $x_1, \ldots, x_n$ are variables, and $C_1, \ldots, C_n$ are constants. The variables $x_1, \ldots, x_n$ must be distinct. Each environment $\rho$ defines a type context $\Gamma = \{x_1 : s_1, \ldots, x_n : s_n\}$, where $s_i$ is the type of $C_i$, for all $i = 1, n$: we say that $\Gamma$ is associated to $\rho$.

Following [41] (see eg. [36]) we use the *natural operational semantics* style, which defines a ternary relation $\rho \bullet e \Downarrow C$ meaning that the term $e$ under the environment $\rho$ evaluates to the constant $C$, and a 4-ary relation $\rho \bullet f(C) \Downarrow C'$ meaning that the function $f$ applied to the constant $C$ in environment

$\rho$ evaluates to $C'$. For a simple example, if $f = \lambda x.(\mathsf{flatten}(x)@[100])$ and $C = [[3,5],[2]]$, then $\emptyset \bullet f(C) \Downarrow [3,5,2,100]$. For an example with an environment, consider $e = \mathsf{zip}\langle x, \mathsf{enumerate}(x)\rangle$ and $\rho = \{x = [30,10,50]\}$: we have $\rho \bullet e \Downarrow [\langle 30,0\rangle, \langle 10,1\rangle, \langle 50,2\rangle]$. The relations $\rho \bullet e \Downarrow C$ and $\rho \bullet f(C) \Downarrow C'$ will only be defined for environments $\rho$ associated to some context $\Gamma$ for which $\Gamma \rhd e : t$, and $\Gamma \rhd f : t \to t'$ Therefore type judgments $\Gamma \rhd e : t$ and operational semantics rules $\rho \bullet e \Downarrow C$ are defined simultaneously in Appendix A. One can prove the expected fact that for any environment $\rho$ and any expression $e$ there exists at most one constant $C$ such that $\rho \bullet e \Downarrow C$ (there may none, if $e$ generates an error or diverges). Similarly, the $C'$ in $\rho \bullet f(C) \Downarrow C'$ is also unique. For that reason we will abbreviate $\rho \bullet e \Downarrow$ and $\rho \bullet f(C) \Downarrow$, when the evaluation result is understood from the context.

**Complexities $T, W$** Each rule in the operational semantics has an associated rule for computing $T$ and one for $W$. Hence, strictly speaking $T, W$ depend also on $\rho$, so we have $T(\rho, e), W(\rho, e)$ denoting the complexities of $\rho \bullet e \Downarrow$, and $T(\rho, f, C), W(\rho, f, C)$ denoting the complexities of $\rho \bullet f(C) \Downarrow$. $W$ follows a simple rule: it is always the sum of the sizes of all values involved in the rule, including all values in the environment $\rho$, For simple base types, like *int*, bounded-length *strings*, we adopt a unit size measure, i.e. $size(x) = 1$ for all objects $x$ of that type; other user-defined types may have arbitrary types. The more interesting rules are those for $T$, since they dictate what is to be computed in parallel and what not. We highlight here the key sources of parallelism in CoPa, as reflected in $T$'s definition.

- First-order operations on collections: Operations like flatten, @, or the specialized operations on bags and sequences all take $T = 1$ hence are inherently "parallel". For example:

$$T(\mathsf{flatten}(e)) \stackrel{\text{def}}{=} 1 + T(e)$$

- Map and Ext: $\mathsf{map}(f)$, and similarly $\mathsf{ext}(f)$, are the main source of parallelism. To execute $n$ threads in parallel, one constructs in CoPa a sequence of length $n$ then maps some function on it.

- map-recursion: Iterating map-parallelism is achieved through map-recursion or while. $T(\mathsf{maprec}(d,c), x)$ is defined as follows. Suppose

$$d(x) = [x_0, \ldots, x_{n-1}]$$

and let

$$\begin{aligned} y_i &\stackrel{\text{def}}{=} \mathsf{maprec}(d,c)(x_i) \quad \text{for } i = 0, n-1 \\ T_d &\stackrel{\text{def}}{=} T(d,x) \\ T_i &\stackrel{\text{def}}{=} T(\mathsf{maprec}(d,c), x_i) \quad \text{for } i = 0, n-1 \\ T_c &\stackrel{\text{def}}{=} T(c, \langle x, [y_1, \ldots, y_n]\rangle) \end{aligned}$$

Then:

$$T(\mathsf{maprec}(d,c)) \stackrel{\text{def}}{=} T_d + \max_{i=1,n}(T_i) + T_c \tag{2}$$

Intuitively, this definition suggests the following way of computing $\mathsf{maprec}(d,c)(x)$:

1. First compute $d(x) = [x_0, \ldots, x_{n-1}]$. This takes time $T_d$.
2. Next compute $\mathsf{maprec}(d,c)(x_0), \ldots, \mathsf{maprec}(d,c)(x_{n-1})$. Perform the $n$ computations *in parallel*. Therefore the total time for this step will be $\max_{i=1,n}(T_i)$.

3. Finally compute $c(\langle x, [y_0, \ldots, y_{n-1}]\rangle)$, which takes time $T_c$.

Note that the above three steps are done sequentially, hence the addition in Equation 2.

We comment now on the relationship between the weakening rule (Subsection 2.1) and parallel evaluation. Consider for example the expressions:

$$(\lambda(x : [\mathtt{N}]).\langle x, \mathsf{map}(f)(x)\rangle)[C_0, \ldots, C_{n-1}]$$

Assume that $f(z) = z + 1$. Starting with the empty environment, we end up evaluating:

$$\rho \bullet \langle x, \mathsf{map}(f)(x)\rangle \Downarrow$$

where $\rho = x = [C_0, \ldots, C_{n-1}]$. This in turn requires us to evaluate both $\rho \bullet x \Downarrow$ and $\rho \bullet \mathsf{map}(f)(x) \Downarrow$ which becomes $\rho \bullet \mathsf{map}(f)([C_0, \ldots, C_{n-1}])$. The first expression needs the environment $\rho$, and evaluates to $[C_0, \ldots, C_{n-1}]$ in $T = 1$ and $W = n$. For the second expression however we have to choices. The first is to evaluate it directly, by evaluating the $n$ expressions $\rho \bullet f(C_i) \Downarrow C_i + 1$: here we get $T = O(1), W = O(n^2)$ because we replicated $\rho$ $n$-times. The second choice is to apply the weakening rule before the $\mathsf{map}$-rule, and evaluate $\bullet\mathsf{map}(f)([C_0, \ldots, C_{n-1}]) \Downarrow$, with complexities $T = O(1), W = O(n)$. As this example illustrates, by applying the weakening rule early we can reduce the work complexity. We will always assume that the weakening rule is applied as early as possible.

Before illustrating some examples, we comment on the impact of the weakening rule (which, recall, allows us to drop useless variables from type contexts). The environment counts in the computation of $W$. Moreover the environment is replicated in the $\mathsf{map}$ and $\mathsf{ext}$ rules. Hence a useless variable in an environment may add up significantly to $W$, e.g. increasing from $O(n)$ to $O(n^2)$. It is important that typechecking in CoPa applies the weakening rule as close to the root as possible.

**Example 3.1** We illustrate these on a simple functions for indexing a sequence, which we will use in the sequel: $\mathit{Index} : [t] \times [\mathtt{N}] \to [t]$, with the meaning:

$$\mathit{Index}\langle X, I\rangle = [x_{i_0}, \ldots, x_{i_{m-1}}]$$

Here $m = \mathsf{length}(I)$, $n = \mathsf{length}(X)$. This can be sought of as a generalization of indexing in an array, $X[i]$. The complexities are $T = O(1), W = O(mn)$. The function first lets each element in the sequence $X$ know its position (in $XE$), then replicates the sequence $m$ times in selects in parallel, from each copy $k$, the element number $i_k$:

```
fun Index⟨X, I⟩ =
    let val XE = zip⟨X, enumerate(X)⟩
        fun s(i) = Π₁(filter(λ⟨x, p⟩.i = p)(XE))
    in flatten(map(s)(I))
    end
```

The replication is done by $\mathsf{map}$-ing the function $s$, which has the free variable $XE$, over $I$, hence we expect $W = O(mn)$. Fig. 8 illustrates the relevant fragment of the actual operational semantics. Here $x$ denotes the value $[x_0, \ldots, x_{n-1}]$, and $i$ the value $[i_0, \ldots, i_{m-1}]$. Recall that we have to apply the rewriting rules of Subsection 2.2 first: hence, the evaluation starts with $\emptyset \bullet \mathit{Index}\langle x, i\rangle \Downarrow$, then proceeds to $\{X = x, I = i\} \bullet \mathsf{zip}\langle X, \mathsf{enumerate}(X)\rangle \Downarrow$. Denoting the latter result with $xe$, the

$$
\dfrac{
  \dfrac{}{\{\dots\} \bullet I \Downarrow i}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{\dots}{XE = xe \bullet s(i_0) \Downarrow [x_{i_0}]}
      \quad \cdots \quad
      \dfrac{\dots}{XE = xe \bullet s(i_{m-1}) \Downarrow [x_{i_{m-1}}]}
    }{XE = xe \bullet \mathsf{map}(s)(i) \Downarrow [[x_{i_0}], \dots, [x_{i_{m-1}}]]}
  }{\{X = x, I = i, XE = xe\} \bullet \mathsf{map}(s)(i) \Downarrow [[x_{i_0}], \dots, [x_{i_{m-1}}]]}
  \over
  \{X = x, I = i, XE = xe\} \bullet \mathsf{map}(s)(I) \Downarrow [[x_{i_0}], \dots, [x_{i_{m-1}}]]
  \qquad
  \dfrac{}{\{\dots\} \bullet \mathsf{flatten}(\dots) \Downarrow \dots}
}{\{X = x, I = i, XE = xe\} \bullet \mathsf{flatten}(\mathsf{map}(s)(I)) \Downarrow [x_{i_0}, \dots, x_{i_{m-1}}]}
$$

Figure 8: Fragment of operational semantics tree

evaluation then proceeds with $\{X = x, I = i, XE = xe\} \bullet \mathsf{flatten}(\mathsf{map}(s)(I)) \Downarrow$, which is shown in Fig. 8. The top part of the tree is responsible for the $W = O(mn)$ complexity: there are $m$ rules, and each carries the environment $XE = xe$, whose size is $n$ — hence $W = O(mn)$. Note that we applied the weakening rule earlier to get rid of $X$ and $I$ in the environment: without that, the work complexity would have been $O(m(m + n))$. We could have applied weakening on the left-most and right-most branches (corresponding to $I \Downarrow i$ and $\mathsf{flatten}([[x_{i_0}], \dots, [x_{i_{m-1}}]]) \Downarrow [x_{i_0}, \dots, x_{i_{m-1}}]$), but this have only cluttered the figure without reducing the complexity. Also, since the height of the tree is independent of $m, n$, we have $T = O(1)$.

It is interesting to remark that if we assume $0 \leq i_0 \leq i_1 \leq \dots \leq i_{m-1} < n$, then *Index* can be expressed in $T = O(1), W = O(m + n)$: we refer the reader to [64] for details.

□

# 4 Complexity-Preserving Compilation

We present in this section a complexity-preserving compilation technique for CoPa. We target two kinds of parallel architectures. The first is a butterfly network: consequently, CoPa can be used as a high-level language to describe parallel algorithms for a butterfly network and to infer their running time from the high-level complexities $T, W$. The second architecture is the LogP model introduced by Culler et al. [22]. This model captures more accurately modern shared-nothing architectures: by choosing the LogP model as the target, CoPa can be used to express parallel programs on architectures of practical interest.

Most of the techniques described here are architecture-independent. Compilation is achieved in a number of steps, illustrated in Figure 1, of which only the last one is architecture dependent: we defer that step to the next section, and discuss here the other steps.

The first step is a preprocessing step encoding bags and sets as (sorted) sequences. We denote CoPa-seq the target language, i.e. CoPa restricted to sequences only. Some set and bag operations can be translated immediately into sequences, others need sorting. For this step our main result consists in showing that sorting of a sequence of length $n$ can be expressed in CoPa in $T = O(\frac{1}{\varepsilon^4} \log^2(\frac{1}{\varepsilon}))$, $W = O(n^{1+\varepsilon})$, for any $\varepsilon > 0$: for fixed $\varepsilon > 0$, this is $T = O(1), W = O(n^{1+\varepsilon})$. Of course, for some particular set and bag operations, like join, current parallel relational systems offer more practical implementations, often based on hashing. As argued in the introduction we do not make use of those techniques here: our main goal is to guarantee asymptotic complexity, and hash-based algorithms do not help us with that.

The second step is the actual compilation and has two phases. First it replaces map-recursion with while-loops. A program with complexities $T, W$ is rewritten into a recursion-free program with $O(\frac{1}{\varepsilon^k} T), O(\frac{1}{\varepsilon^k} T^\varepsilon W)$. Here $\varepsilon$ is any real number $> 0$ and $k$ depends on the program. That is, we describe in fact a rewriting *scheme*, which will produce different recursion-free programs for different $\varepsilon$. Note that by translating recursive programs into recursion free ones we eliminate the need for a stack. This is beneficial for two reasons. First in a database context, where the input collections are the database relations, the intermediate results are typically large relations stored on disc: in our stack-free implementation, only a fixed number of intermediate relations are needed to implement each program. Second, it simplifies memory management on both MIMD and SIMD architectures (SIMD architectures associate a relatively small amount of memory with each processor, making a stack impractical).

The second compilation phase translates recursion-free CoPa programs into an intermediate code, called the *sequence algebra*, SA. This new language, an abstraction of a SIMD architecture, manipulates only flat sequences, and the hard part of the compilation consists in encoding operations on nested sequences, containing nested parallelism, in terms of flat sequences, and at the same time preserving the parallel complexities $T, W$. As in the previous step, a program of complexities $T, W$ is mapped into one of complexities $O(T), O(T^{\varepsilon}W)$.

The third step generates code for a a simple parallel vector model. The model, called BVRAM (Bounded Vector Random Access Machine) and is somehow related to Blelloch's VRAM [7]. It has a finite number of vector registers: the number of registers needed depends on the source program being compiled. The BVRAM communication instructions are simpler than VRAM. The external functions (aggregates and complex user-defined functions), also set apart the BVRAM from the VRAM. All BVRAM instructions, except for external functions, have by definition the parallel time complexity $T = 1$ and a work complexity $W$ equal to the sum of sizes of the input and the output sequences. A complexity-preserving translation from SA to BVRAM is straightforward. As an interesting observation, we show also a complexity-preserving translation from BVRAM to SA, which is less obvious, this proving that BVRAM and SA are strongly equivalent.

Finally, the fourth step in Fig. 1 is architecture-dependent and implements BVRAM on the particular target parallel architectures. We have considered two such architectures: the butterfly network and the LogP model. For the butterfly network, each BVRAM instruction of work complexity $W$ can be executed in $\log W$ parallel steps on a network with $W \log W$ nodes. Note that this is possible precisely because the BVRAM has only simple communication primitives[2] For the LogP model, which is more complex than the butterfly, we prove that all BVRAM instructions except one admit near optimal implementations on LogP. Here optimal is with respect to the communication cost, which is the main complexity addressed by the LogP model. For the last BVRAM instruction we could not prove that our implementation is optimal, but we support the claim for its efficiency with experiments. We discuss this fourth step in the next section.

Considering the architecture-independent compilation only (from CoPa-seq to BVRAM) we obtain the following result:

**Theorem 4.1** *For any fixed $\varepsilon > 0$, any* CoPa-seq *program, with complexities $T, W$, can be compiled into an equivalent BVRAM program of complexities $O(T), O(T^{\varepsilon}W)$.*

While this does offer complexity guarantees, it falls short of preserving the work complexity $W$. We leave open the question whether this result can be improved to $O(T)$ and $O(W)$. Notice that we do achieve ideal complexities in the particular case when $T = O(1)$. We believe this case to be of importance in database applications, where select − from −where blocks, even nested, have $T = O(1)$.

Taking the first step into account too, we can map any CoPa function into a BVRAM by increasing their complexities to $O(T), O(W^{1+\varepsilon})$ (for that, notice that $T \leq W$ for any CoPa expression). This

[2] A result by Borodin and Hopcroft [1] shows that an arbitrary permutation cannot be computed on a butterfly network using oblivious algorithms.

step is unlikely to be improved: we prove the following lower bound, which is also of independent interest. CoPa-seq cannot express sorting in $T = O(1)$ and $W = O(n)$.

## 4.1 Implementing Sets and Bags as Sorted Sequences

We start by describing the the function *sort* : $[t] \rightarrow [t]$. Here we assume that each element of the sequence being sorted has size $O(1)$.

**Theorem 4.2** *There exists a* CoPa *function sort which sorts a sequence of length n in* $T = O(\frac{1}{\varepsilon^4} \log^2 \frac{1}{\varepsilon}), W = O(n^{1+\varepsilon})$*, for* $\varepsilon > 0$*. For fixed* $\varepsilon$*, this yields* $T = O(1)$*,* $W = O(n^{1+\varepsilon})$*.*

**Proof.** We use a multiway merge sorting function, somehow related to the $O(\log n \log \log n)$ sorting algorithm for a hypercube attributed in [46, Section 3.5.3] to Cypher and Plaxton. We need several auxiliary functions for that, which we introduce next.

**Sorting Small Integers** We start with *SmallSort* : $[t \times \mathtt{N}] \times \mathtt{N} \rightarrow [t \times \mathtt{N}]$. *SmallSort*$\langle X, m\rangle$ sorts a sequence $X$ with integer keys, under the assumption that all keys are in $\{0, 1, \ldots, m-1\}$. Assuming $m \leq n$ it constructs the sequence $I = [0, 1, \ldots, m-1]$ as $\mathsf{filter}(\lambda k.k < m)(\mathsf{enumerate}(X))$, then replicates $X$ $m$ times and selects from each copy $k = 0, m-1$ the elements with key $k$ (this is similar to *Index*, Example 3.1). Complexities are $T = O(1), W = O(mn)$.

**Direct Sorting** Using *SmallSort* we derive *DirectSort* : $[t] \rightarrow [t]$ which sorts a sequence in $T = O(1), W = O(n^2)$. Given the sequence $X = [X_0, X_1, \ldots, X_{n-1}]$, compute for each $X_i$ how many elements in the sequence are $< X_i$ (*ranking*): this takes $T = O(1), W = O(n^2)$, by replicating $X$ $n$ times. Then use *SmallSort* to sort $X$ according to these numbers (which are guaranteed to be $< n$).

**Direct Merging** The function *DirectMerge* : $[t] \times [t] \rightarrow [t]$ takes two sorted sequences $X, Y$ of lengths $m$ and $n$ and merges them. It can be expressed in CoPa in $T = O(1), W = O(mn)$. It starts by *ranking* each $X_i$ in the sequence $Y$: this takes $T = O(1), W = O(mn)$. Next it splits $Y$ according to these ranks into $m+1$ subsequences $Y = \mathsf{flatten}([Z_0, Z_1, \ldots, Z_{m-1}])$, and returns $Z_0 @ [X_0] @ Z_1 @ [X_1] @ \ldots @ [X_{m-1}] @ Z_{m-1}$. These latter steps take $T = O(1), W = O(m+n)$.

**Efficient Permutation** Given a sequence $X$ of length $n$ and a permutation, in the form of another sequence $P$ containing the numbers $0, 1, \ldots, n-1$ in some arbitrary order, we want to "permute" $X$ according to $P$. That is, the output sequence $Y = \mathit{Permute}\langle X, P\rangle$ is defined by $Y_{P_i} = X_i$, for $i = 0, n-1$. Using *SmallSort* we can compute the permutation in $T = O(1), W = O(n^2)$, since all we need to do is to sort the sequence $S \stackrel{\text{def}}{=} \mathsf{zip}\langle X, P\rangle$, whose keys are in the range $0, \ldots, n-1$. We can achieve much better as follows. Let $\varepsilon > 0$. We represent each key in $S$ in base $b \stackrel{\text{def}}{=} n^\varepsilon$: this requires at most $\frac{1}{\varepsilon}$ digits. Then we perform a standard radix-sort. At each step we sort $S$ according to a certain digit of its keys (starting with the least significant digit, ending with the most significant one). The range of the digit is $0, \ldots, n^\varepsilon - 1$ and we need $\frac{1}{\varepsilon}$ steps. Each step consists of one call to *SmallSort*. Overall complexities: $T = O(\frac{1}{\varepsilon}), W = O(bn) = O(n^{1+\varepsilon})$.

**Efficient Merging of Multiple Sequences** At the core of our sorting algorithm lies a function merging $k$ sorted sequences of total length $n$. We will always have $k = O(n^{\varepsilon})$. Under this restriction, we present an algorithm with complexities $T = O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon}), W = O(n^{1+2\varepsilon})$. Note that this also implies that we can merge in CoPa two sequences of total length $n$ in $T = O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon}), W = O(n^{1+\varepsilon})$.

We describe here the function *MultiMerge* : $[[t]] \to [t]$. Assuming the input to be $X = [X_0, X_1, \ldots, X_{k-1}]$, where each $X_i$ is a sequence, we start by choosing a set of *splitters* $S_i$ from $X_i$, for each $i = 0, k-1$. Namely $S_i$ will contain exactly $n^{\varepsilon}$ splitters, dividing $X_i$ into $n^{\varepsilon}+1$ subsequences of equal size. This can be easily achieved in $T = O(1), W = O(n)$. Next we sort the total $n^{2\varepsilon}$ splitters using *DirectSort*: $T = O(1), W = O(n^{4\varepsilon})$ (we assume $\varepsilon \leq 1/4$, hence we are still within the allowed overall complexity). Call $S$ the resulting sequence. Next we split each sequence $X_i$ into $n^{2\varepsilon}+1$ subsequences $X_{i0}, X_{i1}, X_{i2}, \ldots$, according to the splitters in $S$. For each $i$, this requires a direct merging of $S$ and $X_i$, with $T = O(1), W_i = O(n^{2\varepsilon}n_i)$, where $n_i = \mathsf{length}(X_i)$. Summing over all $i$'s, we obtain a total work complexity $W = O(n^{1+2\varepsilon})$. Now transpose the nested sequence $[[X_{00}, X_{01}, X_{02}, \ldots], [X_{10}, X_{11}, \ldots], \ldots]$, to get $P = [[X_{00}, X_{10}, X_{20}, \ldots], [X_{01}, X_{11}, X_{21}, \ldots], \ldots]$: this is a permutation problem, and can be achieved with $T = O(\frac{1}{\varepsilon}), W = O(n^{1+\varepsilon})$. Finally we apply *MultiMerge* recursively on each sequence $XX_j \stackrel{\text{def}}{=} [X_{0j}, X_{1j}, \ldots, X_{(k-1)j}]$, $j = 0, n^{2\varepsilon}$. Since $\mathsf{length}(X_{ij}) \leq \mathsf{length}(X_i/n^{\varepsilon})$, it follows that the total length of all $k$ sequences in $XX_j$ is $\leq \sum_i n_i/n^{\varepsilon} \leq n^{1-\varepsilon}$. By repeating this process, we observe that the successive recursive calls to *multi_merge* will merge nested sequences of length $n, n^{1-\varepsilon}, n^{(1-\varepsilon)^2}, \ldots$. Thus, after $T = \log\frac{1}{\varepsilon}/\log\frac{1}{1-\varepsilon} = O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ steps, we have to merge nested sequences of total length $\leq n^{\varepsilon}$. We do this using direct-sort. Overall complexities: $T = O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon})$, $W = O(n^{1+\varepsilon})$.

**Efficient Sorting** Now we can describe the efficient sorting algorithm in CoPa, *Sort* : $[t] \to [t]$, with complexities $T = O(\frac{1}{\varepsilon}\log^2\frac{1}{\varepsilon}), W = O(n^{1+\varepsilon})$. We start by splitting the input sequence $X$ into $n^{\varepsilon}$ subsequences of equal size. We sort these sequence recursively, then merge them using *MultiMerge*. Thus *Sort* will be called on sequences of lengths $n, n^{(1-\varepsilon)}, n^{(1-\varepsilon)^2}, n^{(1-\varepsilon)^3}, \ldots$. After $\log\frac{1}{\varepsilon}/\log\frac{1}{1-\varepsilon} = O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ recursive calls we have to sort sequences of length $\leq n^{\varepsilon}$. Here we use *DirectSort*. Thus, the complexities are $T = O(\frac{1}{\varepsilon^4}\log^2\frac{1}{\varepsilon}), W = O(n^{1+\varepsilon})$. □

The hard part of the sorting algorithm was to achieve $T = O(1)$, while keeping the work complexity under control. Of course, if we want work optimality first of all, we can obtain $W = O(n\log n)$:

**Proposition 4.3** CoPa *can express sort with complexities* $T = O(\log n \log\log n)$ *and* $W = O(n\log n)$.

**Proof.** This is achieved with Valiant's merge algorithm [74, 40], which can be expressed in CoPa; note that we need a user-defined external sorting function to express the last part of the algorithm (sequential merging of small sequences). □

Once we have sorting, set and bag operations can be implemented straightforwardly as sorted sequences. The only catch is with external functions, like, for example, aggregates. For example if $sum : \{\!|\mathtt{N}|\!\} \to \mathtt{N}$ is an external function on bags, then we need a corresponding function $sum : [\mathtt{N}] \to \mathtt{N}$ on sequences in order to implement bags as sequences. Call the set of external functions *closed for sequences* if for any external function over sets and/or bags there exists an *equivalent* one over sequences, with the same complexities. Here *equivalent* means that the two functions commute with the seq2bag and bag2set operators (see Sec. 2 for their definitions): we omit giving a formal, but boring definition of *equivalent*.

**Theorem 4.4** *Assume the set of external functions to be closed for sequences, and let* $\varepsilon > 0$. *Then* *any* CoPa *expression with complexities $T, W$ can be translated into an equivalent expressions with complexities $O(\frac{1}{\varepsilon}\log^2 \frac{1}{\varepsilon^4})$, $O(W^{1+\varepsilon})$ which does not use sets of bags.*

## 4.2 Elimination of map-recursion in CoPa

At this point we have CoPa-seq, which is CoPa without set and bag operations. The second compilation step consists in eliminating *map*-recursion by translating it into while-loops. The central construct is given by the following lemma, which eliminates one occurrence of map-recursion.

**Lemma 4.5** *Let $f = \mathsf{maprec}(d, c)$ where $d$ and $c$ do not contain* map*-recursion, and let its complexities be $T, W$. Then, for any $\varepsilon > 0$, $f$ can be expressed in* CoPa *without recursion with complexities $O(\frac{1}{\varepsilon}T), W(\frac{1}{\varepsilon}T^{\varepsilon}W)$*

**Proof.** We consider a particular case in detail, namely when $f$ is the function $g$ in Fig. 7, that is, a two-way divide-and-conquer recursion. Afterwards we show how to generalize to arbitrary map-recursion. Recall the types: $g : s \to t$, $e : s \to t$, $d_1, d_2 : s \to s$, and $c : t \times t \to t$. We compute $g(x)$ in two steps (here $x$ is $g$'s input, of type $s$):

**Divide Phase** Start with the singleton sequence $y = [x]$ of type $[s]$, and apply repeatedly the function $\mathsf{ext}(\lambda x.\mathsf{if}\ \ p(x)\ \ \mathsf{then}\ \ [x]\ \ \mathsf{else}\ \ [d_1(x), d_2(x)])$ having the type $[s] \to [s]$, until all its elements satisfy the predicate $p$. (We need to tag the elements resulting from $[x]$, to avoid applying $p$ repeatedly on them; we omit the details.) Call $y$ the resulting sequence.

**Combine Phase** Start by map-ing the function $e$ on $y$, and then apply repeatedly $c$ to adjacent elements of $y$: some additional bookkeeping is necessary to make sure $c$ is applied to the correct pairs (e.g., it suffices to store the depth in the divide and conquer tree for each element in $y$, and only combine adjacent elements if they have the same depth). Stop when there is only one element in the resulting list.

Obviously, the translated $g$ will have time complexity $O(T)$. The work complexity preserved only in the case in which $g$'s divide an conquer tree is perfectly balanced. Otherwise the leaves which are reached sooner have to coexists in the same sequence with the nodes which need more divide steps, and add to the total work complexity. We refine now the two steps above in order to limit the increase of the work complexity. Let $\nu$ be the number of different levels in the divide and conquer tree which contain leaves. For example in a perfectly balanced tree $\nu = 1$, while an almost perfectly balanced tree might have $\nu = 2$: a totally "unbalanced" tree may have $\nu$ equal to its depth. Obviously $\nu \leq T \leq W$. We can compute $\nu$ in time and work complexity $O(T), O(W)$, by simulating only the *divide* phase, without retaining the results. Let $\varepsilon > 0$. We improve the *divide* phase, such that the time and work complexities of the translation of $g$ into CoPa without map-recursion become $O(T)$ and $O(\nu^{\varepsilon}W)$ respectively. Namely, we start with $\frac{1}{\varepsilon} + 1$ variables $z_i$, $i = 0, \ldots, \frac{1}{\varepsilon}$, initialized to $[]$, and with $y$ initialized to the singleton $[x]$. We apply repeatedly the divide phase on $y$; whenever some leaves are reached, we move them into $z_0$. We only allow $z_0$ to be touched $\nu^{\varepsilon}$ times, after which we move its entire content into $z_1$, and empty $z_0$. We repeat this process, but only allow $z_1$ to be touched $\nu^{\varepsilon}$ times, at which point, we empty $z_1$, by moving everything into $z_2$. In general, we allow $z_i$ to accumulate only $\nu^{\varepsilon}$ times, after which we empty it, by moving everything into $z_{i+1}$. Obviously, a number of $\nu^{i\varepsilon}$ levels of leaves must be discovered, before making one move into $z_i$; thus, $z_{\frac{1}{\varepsilon}}$ will be filled exactly once, with the leaves from all $\nu$ levels. This is the refined *divide* phase. We compute its work complexity now. In addition to $W \stackrel{\text{def}}{=} W(g, x)$ we pay the price of unnecessarily touching elements in the variables $z_0, z_1, \ldots$, and of moving them from $z_i$ to $z_{i+1}$. Each leaf travels exactly once through $z_0, z_1, \ldots, z_{\frac{1}{\varepsilon}}$: hence the moving part accounts for an additional $\frac{1}{\varepsilon}W$. Next, while sitting in $z_i$, each leaf is "touched" at most $\nu^\varepsilon$ times: the touching part "costs" at most $\frac{1}{\varepsilon}\nu^\varepsilon W$. Thus, the work complexity of the *divide* phase is bounded by $O(\frac{1}{\varepsilon}\nu^\varepsilon W) \leq O(\frac{1}{\varepsilon}T^\varepsilon W)$. We omit the details of the rather complex bookkeeping necessary to keep all elements in $z_i$ sorted (but see Example 4.7 below). Now we compute the time complexity for the *divide* phase. We need $O(T)$ to work on $y$ (the "useful" divide), plus the cost of moving. Variables $z_{\frac{1}{\varepsilon}}, z_{\frac{1}{\varepsilon}-1}, \ldots, z_0$ are touched $1, \nu^\varepsilon, \nu^{2\varepsilon}, \ldots, \nu^{\frac{\varepsilon}{\varepsilon}}$ times respectively: added up, they are $< O(\frac{1}{\varepsilon}\nu) \leq O(\frac{1}{\varepsilon}T)$. Thus the time complexity of the *divide* phase is $O(\frac{1}{\varepsilon}T)$. Finally, the *combine phase* is done similarly, but in reverse. This concludes the special case of 2-way divide-and-conquer recursion.

For a general map-recursive function $f$ we proceed in a similar fashion, but now all intermediate values $x$ in the recursive call $c(x, \mathsf{map}(f)(d(x)))$ introduce an additional leaf. Thus, in the general case $\nu$ is always equal to $T$, the height of the divide and conquer tree. □

It is interesting to visualize the translated function in two particular cases. The first is the simple case: a perfectly balanced 2-way divide-and-conquer function, i.e. like the function $g$ in Fig. 7 but in which the recursion tree is perfectly balanced. Here $\nu = 1$ (all leaves are at the same depth) hence the translated function actually uses only two auxiliary variables $z_0, z_1$ and its complexity improves to $O(T), O(W)$. The second is the following 1-way recursion:

$$\mathsf{fun}\ \ h(x) = \mathsf{if}\ \ p(x)\ \ \mathsf{then}\ \ e(x)\ \ \mathsf{else}\ \ c(x, h(d(x)))$$

This simple function represents a worst case for recursion elimination, since $\nu = T$ (the value of each recursive call must be stored for the recursion unfolding). A traditional implementation would have used a stack to store all the $x$'s. Without a stack, we are forced to split a "virtual" stack into a fixed set of sequences $z_0, z_1, \ldots, z_{\frac{1}{\varepsilon}}$ and move the values between sequences to minimize the number of unnecessary touches.

Proceeding by induction on the structure of CoPa-expression and applying Lemma 4.5 we obtain:

**Theorem 4.6** *Consider some function $f$ defined in* CoPa-seq*, with time and step complexity $T, W$. Then, for any $\varepsilon > 0$, we can construct a function $f'$ in* CoPa-seq *without* map*-recursion, which is equivalent to $f$ and which has time and work complexity $T' = O(\frac{1}{\varepsilon^k}T)$ and $W' = O(\frac{1}{\varepsilon^k}T^\varepsilon W)$ respectively, where $k > 0$ is an integer depending on $f$.*

## 4.3 Compiling into Intermediate Code

The intermediate code which is the target of our compilation is the *Sequence Algebra*, SA, Fig. 9. This algebra is of independent interest, playing a similar role for flat sequences as the relational algebra does for flat sets (relations).

SA's only datatype are *flat sequences*, i.e. values of type $[s]$, where $s$ is a product of base types. Functions in SA map tuples of flat sequences to tuples of flat sequences, hence its types $t$ are given by:

$$\begin{array}{rcl} s & ::= & \mathtt{D} \times \ldots \times \mathtt{D} \\ u & ::= & [s] \\ t & ::= & u \times \ldots \times u \end{array}$$

For example $[\mathtt{N} \times \mathtt{N}]$, *unit* and $[unit] \times [\mathtt{N}]$ are types in SA where, recall, *unit* is obtained by taking $k = 0$ in the product $\mathtt{D}_1 \times \ldots \times \mathtt{D}_k$ or in $u_1 \times \ldots \times u_k$ (hence, both *unit* and $[unit]$ are valid SA types).

Unlike $\mathsf{CoPa}$, which is a language with variables, $\mathsf{SA}$ is an *algebra of functions*. There are only function expressions, no value expressions. A value of type $t$ can still be expressed, namely as a function of type $unit \to t$. $\mathsf{SA}$ allows functions to be combined via function composition, tuples, if-then-else, and while.

External functions are handled in $\mathsf{CoPa}$ as follows. First, for every scalar external function $p : d_p \to c_p$, there exists a corresponding $\mathsf{map}(p) : [d_p] \to [c_p]$ in $\mathsf{SA}$. More general, for any $\mathsf{CoPa}$ scalar function $\varphi : s \to s'$ $\mathsf{SA}$ contains $\mathsf{map}(\varphi) : [s] \to [s']$: this includes any combinations of scalar external functions, conditionals, and projections. Non-scalar external functions also have corresponding functions in $\mathsf{SA}$. To simplify the presentation, we only consider aggregate functions and defer a discussion on more general external functions to the end of this section. For every aggregate $agg : [s] \to s'$ in $\mathsf{CoPa}$, there exists a function $\widetilde{agg} : [s] \times [\mathtt{N}] \to [s']$ with the meaning $\widetilde{agg}\langle x, s\rangle = \mathsf{map}(agg)(\mathsf{split}\langle x, s\rangle)$.

$\mathsf{SA}$ has the following sequence operations: $\mathsf{select}, \mathsf{bmRoute}$ and $\mathsf{sbmRoute}$, which were not defined as primitives in $\mathsf{CoPa}$. All three can be defined in $\mathsf{CoPa}$, but require nested sequences as intermediate results, hence we include them as primitives in $\mathsf{SA}$. We describe them next.

First we need a convention for boolean types in $\mathsf{SA}$. Since we don't have union types, we can no longer take $\mathtt{B} \stackrel{\text{def}}{=} unit + unit$. Instead we encode booleans as $\mathtt{N}$ in the context of scalar operations, and as $[\mathtt{N}]$ in that of sequence operations. In the first case, 0 is $\mathsf{true}$, anything else is $\mathsf{false}$; in the second case $[0]$ is $\mathsf{true}$, anything else is $\mathsf{false}$. Then $\mathsf{select}$ returns the subsequence of elements having a $\mathsf{true}$ on their first position, see Example 2.1.

The *bounded monotone routing* $\mathsf{bmRoute} : [s'] \times [\mathtt{N}] \times [s] \to [s]$ has the following meaning: $\mathsf{bmRoute}\langle x', s', x\rangle$ replicates each element of the sequence $x$ a number of times dictated by the corresponding element in $s'$. However it is required that $x'$ should match in length the final result; $x'$ is called *the bound.* For example $\mathsf{bmRoute}\langle [u_0,\, u_1, u_2, v_0, v_1], [3, 0, 2], [a, b, c]\rangle = [a, a, a, c, c]$. The name "monotone routing" comes from the fact that the elements of $x$ are routed to different positions, but such that the relative order of the elements is preserved. The definition in $\mathsf{CoPa}$ is:

$$\mathsf{bmRoute}\langle x', s', x\rangle \stackrel{\text{def}}{=} \Pi_2(\mathsf{flatten}(\mathsf{map}(distribute_{right})(\mathsf{zip}\langle \mathsf{split}\langle x', s'\rangle, x\rangle)))$$

The *segmented monotone routing* is similar, but now $x$ is replaced by a nested sequence:

$$\mathsf{sbmRoute}\langle x', s', x, s\rangle \stackrel{\text{def}}{=} \mathsf{bmRoute}\langle x', s', \mathsf{split}\langle x, s\rangle\rangle$$

As for $\mathsf{CoPa}$, time and work complexities are defined for each evaluation $f(C) \Downarrow C'$, where $f$ is a function in $\mathsf{SA}$ and $C, C'$ are constants. All operations except the external functions have $T = O(1)$ and $W$ the total sizes of the inputs and outputs; for external functions, complexities are like in $\mathsf{CoPa}$. It follows that for any $\mathsf{SA}$ function $f$, its complexities are equal up to some constant factor to those of $f$ expressed in $\mathsf{CoPa}$.

**Example 4.7** At a technical level, several compilation steps need the function $combine : [\mathtt{B}] \times [s] \times [s] \to [s]$ (in this example we identify $\mathtt{B} = \mathtt{N}$), where $combine\langle b, x, y\rangle$ shuffles the sequences $x$ and $y$, according to the tags given by $b$. For example, when $b = [\mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{true}, \mathsf{false}, \mathsf{true}, \mathsf{true}]$ and $x = [x_0, x_1, x_2, x_3], y = [y_0, y_1, y_2]$, then:

$$combine\langle b, x, y\rangle = [x_0, y_0, y_1, x_1, y_2, x_2, x_3]$$

The arguments must satisfy $\mathsf{length}(b) = \mathsf{length}(x) + \mathsf{length}(y) = \mathsf{length}(z)$, where $z$ is the result, otherwise *combine* is undefined. This function is useful both in Lemma 4.5 and in Lemma 4.8 below,

| Expression | Type | Meaning |
|---|---|---|
| $\mathsf{map}(\varphi)$ | $[s] \to [s']$ | where $\varphi : s \to s'$ is a scalar function expressible in CoPa — see text |
| $\widetilde{agg}$ | $[s] \times [\mathtt{N}] \to [s']$ | where $agg : [s] \to s'$ is an aggregate external function in CoPa — see text |
| $\mathsf{id}_t$ | $t \to t$ | the identity |
| $g \circ f$ | $t \to t''$ | composition; where $f : t \to t'$ and $g : t' \to t''$ |
| $\pi_i^{t_1 \times \ldots \times t_n}$ | $t_1 \times \ldots \times t_n \to t_i$ | projection |
| $!$ | $t \to unit$ | |
| $\langle f_1, \ldots, f_k \rangle$ | $t \to t_1 \times \ldots \times t_k$ | where $f_i : t \to t_i$, $i = 1, n$ |
| $\mathsf{if}\ f\ \mathsf{then}\ g_1\ \mathsf{else}\ g_2$ | $t_1 \to t_2$ | where $f : t_1 \to [\mathtt{N}]$, $g_i : t_1 \to t_2$, $i = 1, 2$ |
| $[]$ | $unit \to [s]$ | empty sequence |
| singleton | $unit \to [unit]$ | returns $[\langle\rangle]$ |
| @ | $[s] \times [s] \to [s]$ | concatenation |
| length | $[s] \to [\mathtt{N}]$ | e.g. $\mathsf{length}([a, b, c]) = [3]$ |
| select | $[\mathtt{N} \times s] \to [s]$ | see text |
| zip | $[s] \times [s'] \to [s \times s']$ | |
| enumerate | $[s] \to [\mathtt{N}]$ | |
| bmRoute | $[s'] \times [\mathtt{N}] \times [s] \to [s]$ | see text |
| sbmRoute | $[s'] \times [\mathtt{N}] \times [s] \times [\mathtt{N}] \to [s]$ | see text |
| $\mathsf{while}(p, f)$ | $t \to t'$ | where $p : t \to [\mathtt{N}]$, $f : t \to t$ |

Figure 9: The Sequence Algebra, SA.

when we "move" elements from one sequence "into" another: to preserve the relative order of the elements we use *combine*. We show here how to express *combine* is SA: obviously, it can be expressed in CoPa too. Start by computing $e = \mathsf{enumerate}(b)$: in our example $e = [0, 1, 2, 3, 4, 5, 6]$. Next we compute:

$$u = \mathsf{select}(\mathsf{zip}\langle b, e\rangle)$$
$$u' = \mathsf{select}'(\mathsf{zip}\langle b, e\rangle)$$

Here $\mathsf{select}'$ is a complement of select, which selects all elements having a value false rather than true. In our example $u = [0, 3, 5, 6]$ and $u' = [1, 2, 4]$. These two lists tell us on which position each element of $x$ and $y$ must end up. It suffices to "route" each element from $x$ and $y$ to their corresponding position. For this we subtract each number in $u$ from its right neighbor (by considering $n \stackrel{\text{def}}{=} \mathsf{length}(b)$ to be the right neighbor of the last element), with the exception of the first position, where we also add the number itself; similarly for $u'$. Formally[3]:

$$v = \mathsf{map}(\lambda\langle i, a, b\rangle.\mathsf{if}\ i = 0\ \mathsf{then}\ a\ \mathsf{else}\ a - b)\mathsf{zip}\langle \mathsf{enumerate}(b), tail(u@[n]), u\rangle$$

and similarly for $v'$. In our example we get:

$$\begin{array}{rcl} v &=& [0 + (3 - 0), 5 - 3, 6 - 5, 7 - 6] &=& [3, 2, 1, 1] \\ v' &=& [1 + (2 - 1), 4 - 2, 7 - 4] &=& [2, 2, 3] \end{array}$$

Now we bmRoute $x$ and $y$, using these $v$ and $v'$ as replication sequences:

$$\begin{array}{rcl} xx &=& \mathsf{bmRoute}\langle b, v, x\rangle \\ yy &=& \mathsf{bmRoute}\langle b, v', y\rangle \end{array}$$

[3]Here $tail([u_0, \ldots, u_{m-1}]) = [u_1, \ldots, u_{m-1}]$ and is defined as $\Pi_2(\mathsf{select}'(\mathsf{zip}\langle \mathsf{enumerate}(u), u\rangle))$.

In our example $xx = [x_0, x_0, x_0, x_1, x_1, x_2, x_3]$ and $yy = [y_0, y_0, y_1, y_1, y_2, y_2, y_2]$ respectively (both have the length as $b$). Finally, we zip them together with $b$, and map the scalar function $\lambda\langle b, x, y\rangle.$if $b$ then $x$ else $y$.

□

We now describe the compilation of recursion-free CoPa-seq into the intermediate code SA. First we eliminate the free variables in CoPa expressions and subexpressions. We only sketch this step here, and refer the reader to [64] for full details. Given a CoPa expression we rewrite it into an equivalent one in which each subexpression is closed. The rewriting changes the types as follows. Recall that free variables occur in a *context*, $\Gamma = \{x_1 : t_1, \ldots, x_n : t_n\}$. The rewriting translates:

- a value expression $\Gamma \rhd e : t$ into function $t_1 \times \ldots \times t_n \to t$.
- a function expression $\Gamma \rhd f : t \to t'$ into function $t_1 \times \ldots \times t_n \times t \to t'$.

Note that after the rewriting, all value expressions become function expressions: in particular, a value of type $t$ becomes a function of type $unit \to t$. It can be shown that this translation rewrites any CoPa expression into some formalism which differs from CoPa only in that (a) all expressions and subexpressions are functions without free variables, and (b) has the following additional operators:

- Function composition $g \circ f$.
- $\mathsf{distribute}_{left}$ (see Example 2.2). This is needed to rewrite expressions like $\mathsf{map}(f)(x)$, where $f$ has some free variable $y$: we need to "distribute" $y$ over $x$ in order to eliminate the free variable from $f$.
- The "functional case statement" $f_1 + \ldots + f_k : t_1 + \ldots + t_k \to t$, when $f_i : t_i \to t$ for $i = 1, k$. Its meaning is $(f_1 + \ldots + f_k)(in_i(v)) \stackrel{\text{def}}{=} f_i(v)$: this is needed to replace the case statement, which relies on free variables.
- A function $\delta : (t_1 + \ldots + t_n) \times s \to t_1 \times s + \ldots + t_n \times s$, with meaning $\delta\langle in_i(x), y\rangle \stackrel{\text{def}}{=} \langle in_i\langle x, y\rangle\rangle$: this is needed when the case statement has free variables itself.

We refer the reader to [64] for the details of this translation. From now on we will consider all CoPa expressions to be closed functions, and move on to the main part of the compilation: flattening. For that we use *segment descriptors* [7]. A sequence of types [[D]] will be encoded as two sequences of types [D] and [N] respectively. For example $x = [[a, b], [], [c, d, e]]$ will be encoded as the pair $seq(x) = \langle [a, b, c, d, e], [2, 0, 3]\rangle$ where the first component is the *sequence body* (= $\mathsf{flatten}(x)$), while the second component is the *segment descriptor* (= $\mathsf{map}(\mathsf{length})(x)$). Before applying this idea to arbitrary types (which may contain several levels of nestings, as well as unions), we describe first in more detail the *one-level* unnesting. For each type $t$ in SA, define $SEQ(t)$ to be the encoding of $[t]$ in SA. Referring to the grammar at the beginning of this subsection, we define formally:

$$SEQU([s]) \stackrel{\text{def}}{=} [s] \times [\mathtt{N}] \tag{3}$$

$$SEQ(u_1 \times \ldots \times u_k) \stackrel{\text{def}}{=} (SEQU(u_1) \times \ldots \times SEQU(u_k)) \times [\mathtt{N}] \tag{4}$$

Equation 3 should be clear, given the above discussion. In Equation 4 the last component stores the length of the encoded sequences: it is necessary only when $k = 0$, when the sequence $[\langle\rangle, \ldots, \langle\rangle]$ is encoded as $[n]$, with $n$ its length. We keep this last component even when $k > 0$, for uniformity.

For each nested sequence $x$ of type $[t]$, define $seq_t(x)$ to be its encoding of type $SEQ(t)$. Formally (we show the binary product only; a $k$-ary product is similar):

$$sequ_{[s]}([x_0,\dots,x_{n-1}]) \stackrel{\text{def}}{=} \langle x_0@\dots@x_{n-1},[s_0,\dots,s_{n-1}]\rangle$$
$$seq_{u\times u'}[\langle x_0,x'_0\rangle,\dots,\langle x_{n-1},x'_{n-1}\rangle] \stackrel{\text{def}}{=} \langle\langle sequ_u[x_0,\dots,x_{n-1}], sequ_{u'}[x'_0,\dots,x'_{n-1}]\rangle,[n]\rangle$$

where $s_i \stackrel{\text{def}}{=} \mathsf{length}(x_i)$, $i=0,n-1$.

We now reached the crucial step of our compilation: the one-level complexity-preserving flattening.

**Lemma 4.8 (The Map Lemma)** *Let $f : t \to t'$ be some function in* SA*, and let* $\mathsf{map}(f)$ *have complexities*[4] *$T, W$. Then, for every $\varepsilon > 0$, there exists some function* $\mathsf{flatmap}(f) : SEQ(t) \to SEQ(t')$ *in* SA*, of complexities $T' = O(T)$, $W' = O(T^\varepsilon W)$, which "simulates"* $\mathsf{map}(f) : [t] \to [t']$*. More precisely, $\forall x_0,\dots,x_{n-1} \in t$, $seq_{t'}(\mathsf{map}(f)[x_0,\dots,x_{n-1}]) = \mathsf{flatmap}(f)(seq_t([x_0,\dots,x_{n-1}]))$. The structure of the expression* $\mathsf{flatmap}(f)$ *is independent of $\varepsilon$: the value $\varepsilon$ occurs in* $\mathsf{flatmap}(f)$ *only as the integer constant $\lceil\frac{1}{\varepsilon}\rceil$.*

**Proof.** This is done by induction on the structure of $f$, see Fig. 9. When $f$ is $\mathsf{map}(\varphi)$ then $\mathsf{flatmap}(f)$ is essentially the same $\mathsf{map}(\varphi)$: to be precise, when $\mathsf{map}(\varphi) : [s] \to [s']$ then[5] $\mathsf{flatmap}(\mathsf{map}(\varphi)) : [s]\times[\mathtt{N}]\times[\mathtt{N}] \to [s']\times[\mathtt{N}]\times[\mathtt{N}]$ is given by $\langle \mathsf{map}(\varphi)\circ\pi_1,\pi_2,\pi_3\rangle$, hence we say that it is "essentially" $\mathsf{map}(\varphi)$. Similarly, $\mathsf{flatmap}(\widetilde{agg})$ is essentially $\mathsf{flatmap}(\widetilde{agg})$, $\mathsf{flatmap}(\mathsf{select})$ is essentially $\mathsf{select}$, $\mathsf{flatmap}(\mathsf{bmRoute})$ is a $\mathsf{bmRoute}$ too, while $\mathsf{flatmap}(\mathsf{sbmRoute})$ is another $\mathsf{sbmRoute}$. The only difficult case is when $f$ is $\mathsf{while}(p,g)$. We describe informally how to compute $\mathsf{flatmap}(\mathsf{while}(p,g))(x)$, with $x=[x_0,\dots,x_{n-1}]$. Suppose $x$ is in variable $z_0$. We will use only two additional variables, $z_1$ and $z_2$, which are initially empty. Let $t_i$ be the number of iterations of $\mathsf{while}(p,g)(x_i)$, and assume in the beginning that $t_0 < t_1 < \dots < t_{n-1}$ (we conceptually group all $x_i$'s having the same $t_i$), which implies $t_i \ge i$. Let $\delta = n^\varepsilon$, $W_i = W(\mathsf{while}(p,g),x_i)$ and $r = \lceil\frac{1}{\varepsilon}\rceil - 1$. It follows that $\delta^r \ge \frac{n}{n^\varepsilon}$. For the moment, assume that in the sequence $x_i, g(x_i), g^{(2)}(x_i),\dots$, the last value (on position $t_i$) has the smallest size, denoted by $s_i$, so $s_i t_i \le W_i$. The simulation proceeds in $r+1$ stages, numbered $0,1,\dots,r$. For each $k=0,r$, denote $i_k = \frac{n}{\delta^{r-k}} \approx n^{(k+1)\varepsilon}$: the indexes $0,i_0,i_1,\dots,i_r = n$ divide the sequence $x_0,\dots,x_{n-1}$ into $r+1$ groups, each group $k$ being finalized during stage $k$. The cardinalities of these groups increase geometrically: the first group has $i_0 \approx n^\varepsilon$ elements, the second $i_1 - i_0 \approx n^{2\varepsilon} - n^\varepsilon$, etc. Starting at stage 0, we repeatedly apply $\mathsf{flatmap}(g)$ on $x$: after each application of $\mathsf{flatmap}(g)$, check whether there are some $x_i$'s which have reached the end of the iteration, and move them into $z_1$. Stop this stage when the first group $\{x_i \mid i = 0, i_0-1\}$ has been moved from $z_0$ into $z_1$. During this stage we have done some redundant work on $z_1$, because we have touched the values stored there several times. But this redundant work is less than $O(\frac{n}{\delta^r}\sum_{i=0,i_0} W_i) \le O(n^\varepsilon \sum_{i=0,i_0} W_i)$. After stage 0 is completed, we move the entire $z_1$ into $z_2$. Each of the remaining stages $k=1,r$ is similar to the first one. Namely we apply repeatedly $\mathsf{flatmap}(g)$ on $x$, and move, whenever some iterations terminate, the corresponding elements $x_i$ into $z_1$. More precisely, during stage $k$, all elements in group $k$, $\{x_i \mid i = i_{k-1}, i_k - 1\}$ will be moved from $z_0$ to $z_1$. At the end of stage $k$, we move the entire $z_1$ into $z_2$. Again we have some redundant work during stage $k$, due to repeatedly touching the elements already stored in $z_1$. Since the group sizes increase from one stage to another, the additional touches become larger and larger. However elements in latter groups are allowed to be touched more times, since they contributed more to the overall work complexity $W$. Putting this into formulas, each $x_i$ stored in $z_1$ is touched additionally for a total work which is $\le s_i(\frac{n}{\delta^{r-k}} - \frac{n}{\delta^{r-k+1}}) \le s_i \frac{n}{\delta^{r-k}}$. But since $i \ge \frac{n}{\delta^{r-k+1}}$, we have that

[4] Recall that $\mathsf{map}(f)$ is only in CoPa, not in SA, hence the complexities refer to CoPa.

[5] Notice that $SEQ([s]) = SEQU([s]) \times [\mathtt{N}] = [s]\times[\mathtt{N}]\times[\mathtt{N}]$.

$t_i \geq i \geq \frac{n}{\delta^{r-k}}\frac{1}{\delta}$, hence the additional work complexity for $x_i$ is $\leq s_i t_i \delta \leq W_i n^{\varepsilon}$, which, when added up over all $x_i$'s moved from $z_0$ to $z_1$ at stage $k$, accounts for $O(n^{\varepsilon} \sum_{i=i_{k-1},i_k-1} W_i)$. Now by adding up over all $r$ stages, we find that the total additional work done on $z_1$ is at most $O(n^{\varepsilon} W) \leq O(T^{\varepsilon} W)$. There is also some redundant work done on $z_2$: during all $r$ stages, $z_2$ is touched only $r$ times, for an additional $O(\frac{1}{\varepsilon} W)$ work complexity. At the end of the last stage, all $x_i$'s $(i = 1, n)$ end up in $z_2$, so $z_2$ contains the result of $\mathsf{flatmap}(\mathsf{while}(p, g))(x)$.

The assumption $t_0 < t_1 < \ldots < t_{n-1}$ is not critical. We may assume that elements $x_i$ finish the iterations in any order: for that we need to do additional bookkeeping to trace the relative order of elements in $z_0, z_1, z_2$. Here we use the function *combine* of Example 4.7. First we maintain two boolean sequences $b_0, b_1$: $b_0$ has total length $\mathsf{length}(z_0) + \mathsf{length}(z_1)$ and says how elements in $z_0$ and $z_1$ have to be interleaved, while $b_1$ has total length $\mathsf{length}(z_0) + \mathsf{length}(z_1) + \mathsf{length}(z_2) = \mathsf{length}(x)$ and says how $z_0, z_1$ together have to be interleaved with $z_2$. Whenever we "move" elements from $z_0$ to $z_1$ we use the function *combine* to put them in the right position: the work complexity is $O(\mathsf{length}(z_0) + \mathsf{length}(z_1))$, hence the above analysis remains unchanged. When moving from $z_1$ to $z_2$, we *combine* $z_0$ with $z_1$ first, then *combine* the result with $z_2$ and drop the elements which were coming from $z_0$. The total work complexity here is $O(W)$, which we can afford, since this operation is done at most $\frac{1}{\varepsilon}$ times.

Finally we have to show how to define $\mathsf{flatmap}(\mathsf{while}(p, g))(x)$ in the general case, when the sequence $x_i, g(x_i), g^{(2)}(x_i), \ldots, g^{(t_i)}$ has a minimum size on some position $m_i$ which is not necessarily the last one. In that case we first compute $m_i$, for each $i$: this can be done with complexities $O(T)$ and $O(W)$, by simply applying $\mathsf{flatmap}(g)$ repeatedly, and eliminating those elements which reach the end of their iteration. Next we split the whole iteration $\mathsf{flatmap}(\mathsf{while}(p, g))(x)$ in two parts, essentially by synchronizing the $n$ parallel iterations at the moment when they reach their minimum size, namely: (1) perform the $n$ parallel iterations, as described above, but stop the iteration over $x_i$ at step $m_i$, (2) continue the $n$ parallel iterations, from step $m_i$ to $t_i$, using the same technique, but in reverse (because now the minimum sizes are at the beginning). "In reverse" here means that all elements $x_i$ are initially stored in $z_2$, and progressively moved to $z_1$, and from here to $z_0$. The main iteration is done on $z_0$, i.e. here are the elements on which we apply $\mathsf{flatmap}(g)$ successively. Elements $x_i$ are moved from $z_2$ to $z_1$ and $z_0$ in decreasing order of their $t_i - m_i$. The moments when these elements are moved to $z_0$ are chosen such that all iterations in $z_0$ end simultaneously. Note that this requires us to know the values $t_0 - m_0, t_1 - m_1, \ldots, t_{n-1} - m_{n-1}$ in advance. But they can be computed easily in $O(T), O(W)$, by essentially simulating $\mathsf{flatmap}(\mathsf{while}(p, g))$ without keeping the results. $\square$

We show now how to flatten arbitrary types in CoPa (involving several sequence nesting and unions). For every type $t$ in CoPa define $\mathit{COMPILE}(t)$ to be the following SA type:

$$
\begin{aligned}
\mathit{COMPILE}(\mathtt{D}) &\stackrel{\text{def}}{=} [\mathtt{D}] \\
\mathit{COMPILE}(t_1 \times \ldots \times t_n) &\stackrel{\text{def}}{=} \mathit{COMPILE}(t_1) \times \ldots \times \mathit{COMPILE}(t_n) \\
\mathit{COMPILE}(t_1 + \ldots + t_n) &\stackrel{\text{def}}{=} \mathit{COMPILE}(t_1) \times \ldots \times \mathit{COMPILE}(t_n) \\
\mathit{COMPILE}([t]) &\stackrel{\text{def}}{=} \mathit{SEQ}(\mathit{COMPILE}(t))
\end{aligned}
$$

In addition we define the function $\mathit{encode}_t : t \to \mathit{COMPILE}(t)$:

$$
\begin{aligned}
\mathit{encode}_{\mathtt{D}}(x) &\stackrel{\text{def}}{=} [x] \\
\mathit{encode}_{t_1 \times \ldots \times t_n} \langle x_1, \ldots, x_n \rangle &\stackrel{\text{def}}{=} \langle \mathit{encode}_{t_1}(x_1), \ldots, \mathit{encode}_{t_n}(x_n) \rangle \\
\mathit{encode}_{t_1 + \ldots + t_n}(\mathit{in}_i(x)) &\stackrel{\text{def}}{=} \langle [], \ldots, [], \mathit{encode}_{t_i}(x), [], \ldots, [] \rangle
\end{aligned}
$$

$$encode_{[t]}([x_0, \ldots, x_{n-1}]) \stackrel{\text{def}}{=} seq_{COMPILE(t)}[encode_t(x_0), \ldots, encode_t(x_{n-1})]$$

Finally, we prove:

**Theorem 4.9** *Let $f : t \to t'$ be some* CoPa *function with complexities $T, W$. Then, for every $\varepsilon > 0$, there exists some function $COMPILE(f) : COMPILE(t) \to COMPILE(t')$ in* SA *equivalent to $f$ in the sense that $\forall x \in t$:*

$$COMPILE(f)(encode_t(x)) = encode_{t'}(f(x))$$

*The time and work complexities of $COMPILE(f)$ are $T' = O(T)$, $W' = O(T^{\varepsilon}W)$. Moreover, the only way in which the expression $COMPILE(f)$ depends on $\varepsilon$ is that it has an integer constant equal to $\lceil \frac{1}{\varepsilon} \rceil$.*

**Proof.** By induction on the structure of $f$. Recall that in our setting $f$ does not belong to CoPa, but to the slightly modified formalism described earlier: the cases are illustrated in Fig. 10. Most cases are simple with two exceptions: while (where we use the Map Lemma) and split. We briefly explain some of simple ones below, then describe split. Most become obvious once the types are explained. Starting with the scalar external function $p : c_p \to d_p$, assume $c_p = \mathtt{D}_1 \times \ldots \times \mathtt{D}_k$ and $d_p = \mathtt{D}'_1 \times \ldots \times \mathtt{D}'_l$. Then $COMPILE(c_p) = [\mathtt{D}_1] \times \ldots \times [\mathtt{D}_k]$ and $COMPILE(d_p) = [\mathtt{D}'_1] \times \ldots \times [\mathtt{D}'_l]$. A value $x \in c_p$, $x = \langle x_1, \ldots, x_k \rangle$ becomes $encode(x) = \langle [x_1], \ldots, [x_k] \rangle$: hence, the expression in Fig. 10 (we assumed zip to zip $k$ sequences instead of 2). For an aggregate $agg : [\mathtt{D}_1 \times \ldots \times \mathtt{D}_k] \to (\mathtt{D}'_1 \times \ldots \mathtt{D}'_l)$ the input type is $[\mathtt{D}_1] \times [\mathtt{N}] \times \ldots \times [\mathtt{D}_k] \times [\mathtt{N}] \times [\mathtt{N}]$ while the output type is $[\mathtt{D}'_1] \times \ldots \times [\mathtt{D}'_l]$. For equality, recall that we only have equality at base types in CoPa: hence both $COMPILE(f_1)$ and $COMPILE(f_2)$ return a value of type $[\mathtt{D}]$. Here $eq : \mathtt{D} \times \mathtt{D} \to \mathtt{N}$ is the equality function returning 0 on equal values and 1 on different ones. For $\langle f_1, \ldots, f_k \rangle$, notice $\langle COMPILE(f_1), \ldots, COMPILE(f_k) \rangle$ may return a tuple much wider than $k$, since each $COMPILE(f_i)$ returns a tuple of sequences, rather than a single sequence. For the same reason compiling a projection requires an inspection of the compiled types: $COMPILE(\pi_i)$ has to return *all* flat sequence components encoding the $i$'s component. For $in_i$, notice also that id may be in fact a tuple, hence contributes with more than one position in the result. The case statement $f \stackrel{\text{def}}{=} f_1 + \ldots + f_k$ is more interesting. Recall that on a value $x = in_i(v)$, $f(x)$ returns $f_i(v)$. The value $x$ is encoded as $\tilde{x} = \langle [], \ldots, [], encode(v), [], \ldots, [] \rangle$. Then $f_1 + \ldots + f_k$ is computed by applying each of $COMPILE(f_1), \ldots, COMPILE(f_k)$ to its respective component and "append" the $k$ results (by "append" here we mean component-wise append, since each result may be a tuple). Of course, $COMPILE(f_i)(\tilde{x})$ returns the desired result: we argue that the others return [], and moreover do this with low complexities. This is based on the following two observations: (a) for each value $u$, $encode(u)$ cannot be $\langle [], \ldots, [] \rangle$ (i.e. each encoding has at least one non-empty flat sequence), and (b) for any compiled function, $COMPILE(g)\langle [], \ldots, [] \rangle = \langle [], \ldots, [] \rangle$, and this computation takes $T = O(1), W = O(1)$. For [], assume $t$ to be its type, and $COMPILE(t) = [s_1] \times \ldots \times [s_k]$. Then

$$\begin{aligned} encode([]) &= seq_{[s_1] \times \ldots \times [s_k]}([]) \\ &= \langle sequ_{[s_1]}, \ldots, sequ_{[s_k]}, [0] \rangle \\ &= \langle [], [], \ldots, [], [], [0] \rangle \end{aligned}$$

(a $(2k+1)$-tuple), which explains the corresponding table entry. We briefly describe $\mathsf{distribute}_{left}\langle x, Y \rangle$. Recall its meaning: $x$ has to be replicated, once for each element in the sequence $Y$. The encoding of $x$ consists of sequences, hence we have to replicate sequences: this is done with sbmRoute (details omitted). The other cases, except split, are similar and are omitted. For map$(f)$ we use the Map Lemma.

We now illustrate split$\langle x, s \rangle$. The case when $x$ is a flat sequence is simple: we just use $s$ as the new segment descriptor. When $x$ is a nested sequence, then $encode(x)$ has the form $\langle bd_1, sd_1, \ldots, bd_k, sd_k, [n] \rangle$, where each $sd_i$ is a segment descriptor for the body $bd_i$, and $n = \mathsf{length}(x)$. Here all segment descriptors have the same length ($n$), but the bodies have various lengths. Then $\mathsf{split}\langle x, s\rangle$ must have $2k+1$ additional segment descriptors: $k$ are just $s$, one is $[1, 1, \ldots, 1]$ ($m \stackrel{\text{def}}{=} \mathsf{length}(s)$ times) the other $k$ require some computations:

$$encode(\mathsf{split}\langle x, s\rangle) = \langle bd_1, s_1^{\text{new}}, sd_1, s, \ldots, bd_k, s_k^{\text{new}}, sd_k, s, s, [1, \ldots, 1], [m]\rangle$$

The following example illustrates the new storage descriptors for the case $k = 1$, considering $s = [2, 0, 3]$:

$\mathsf{split}\langle [[a, b], [c, d, e], [f], [], [g, h, i]], s\rangle \quad = \quad [[[a, b], [c, d, e]], [], [[f], [], [g, h, i]]]$

| | | | | | |
|---|---|---|---|---|---|
| $bd =$ | $a\ b$ | $c\ d\ e$ | $f$ | | $g\ h\ i$ |
| $sd =$ | 2 | 3 | 1 | 0 | 3 |
| $[n] =$ | 9 | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| $bd =$ | $a\ b$ | $c\ d\ e$ | | $f$ | | $g\ h\ i$ |
| $sd^{\text{new}} =$ | 5 | | 0 | 4 | | |
| $sd =$ | 2 | 3 | | 1 | 0 | 3 |
| $s =$ | 2 | | 0 | 3 | | |
| $s =$ | 2 | | 0 | 3 | | |
| $s =$ | 1 | | 1 | 1 | | |
| $[m] =$ | 3 | | | | | |

To understand the above $encode(\mathsf{split}\langle x, s\rangle)$ notice that the three vertical stripes each encodes one of the nested sequences $[[a, b], [c, d, e]]$, $[]$, and $[[f], [], [g, h]]$ respectively, in their rows 1,3,5 ($bd, sd, s$). We need to compute the new storage descriptor $sd^{\text{new}} \leftarrow sdNew\langle bd, sd, s\rangle$. We could define $sdNew\langle bd, sd, s\rangle$ to be $\widetilde{sum}\langle sd, s\rangle$, but this assumes the presence of the aggregate function $sum$, and we do not want to make the compilation dependent on the existence of some particular aggregate functions. Instead we describe the function in Fig. 11. The figure also illustrates the function on our running example. Here $\Delta([x_0, \ldots, x_n]) \stackrel{\text{def}}{=} [x_1 - x_0, x_2 - x_1, \ldots, x_n - x_{n-1}]$, and is easily expressible in $\mathsf{SA}$.

□

Note that, in contrast to Theorem 4.6, the number of registers only depends on $f$ and not on $\varepsilon$.

A few remarks are in order.

**External functions** For presentation purposes we have considered only scalar external functions and aggregate ones. In fact all results in this subsection carry over to arbitrary external functions. It suffices to require that, for each such function $p : c_p \to d_p$ in $\mathsf{CoPa}$, $\mathsf{SA}$ contain a primitive $\tilde{p} : \mathsf{flatmap}(COMPILE([c_p])) \to \mathsf{flatmap}(COMPILE([d_p]))$, implementing $\mathsf{map}(p)$.

**Relationship to previous work** Blelloch and Sabot [12] and Blelloch [7] discuss flattening (compilation) of data-parallel languages, later deployed in the implementation of NESL [8]. Their techniques apply to functions with arbitrary recursion, and use a stack of sequences. Complexity cannot be guaranteed, except for certain recursive functions which are *contained*: this is a semantic condition and cannot be checked by a compiler. Our objective is to offer complexity guarantees for *any* program in our language: this is obtained by imposing syntactic restrictions on the recursion schemas ($\mathsf{map}$-recursion). It is true that this restriction implies that the recursive function is *contained*. However, our flattening techniques improve over theirs in the following ways: (a) we don't need a stack, (b) we are more economical in what communication primitives we use over flat sequences ($\mathsf{SA}$ does not have an arbitrary permutation primitive, prefix sums, etc), and (c) we deal with $\mathsf{while}$ in addition to recursion. The latter is a subtle but important point and deserves some comments. Any $\mathsf{while}$ expression can be translated into a tail-recursive function, hence Blelloch

| CoPa Expression $f$ | Compiled SA expression $COMPILE(f)$ |
|---|---|
| scalar function $p$ | $\langle \Pi_1, \ldots, \Pi_l \rangle \circ \mathsf{map}(p) \circ \mathsf{zip}$ |
| aggregate function $agg$ | $\langle \Pi_1, \ldots, \Pi_l \rangle \circ \widetilde{agg} \circ \langle \mathsf{zip} \circ \langle \Pi_1, \Pi_3, \ldots, \Pi_{2k+1} \rangle, \Pi_2 \rangle$ |
| $f_1 = f_2$ | $\mathsf{map}(eq) \circ \mathsf{zip} \circ \langle COMPILE(f_1), COMPILE(f_2) \rangle$ |
| $\langle f_1, \ldots, f_k \rangle$ | $\langle COMPILE(f_1), \ldots, COMPILE(f_k) \rangle$ |
| $\pi_i$ | $\langle \pi_j, \pi_{j+1}, \ldots, \pi_{j'} \rangle$ |
| $in_i^{t_1+\ldots+t_n}$ | $\langle [] \circ !, \ldots, \mathsf{id}, \ldots, [] \circ ! \rangle$ |
| $f_1 + \ldots + f_k$ | $(COMPILE(f_1) \circ \pi_1) @ \ldots @ (COMPILE(f_k) \circ \pi_k)$ |
| $g \circ f$ | $COMPILE(g) \circ COMPILE(f)$ |
| $[]$ | $\langle [], [], \ldots, [], [], \mathsf{map}(0) \circ \mathsf{singleton} \rangle$ |
| $[f]$ | $\langle \pi_1, \mathsf{length} \circ \pi_1, \ldots, \pi_k, \mathsf{length} \circ \pi_k, \mathsf{map}(1) \circ \mathsf{singleton} \circ ! \rangle \circ COMPILE(f)$ where $f : s \to t$ and $COMPILE(t) = t_1 \times \ldots \times t_k$ |
| $f_1 @ f_2$ | $\langle @ \circ \langle \pi_1, \pi_{k+1} \rangle, \ldots, @ \circ \langle \pi_{k-1}, \pi_{2k-1} \rangle, \mathsf{map}(+) \circ \mathsf{zip} \circ \langle \pi_k, \pi_{2k} \rangle \rangle$ where $f_i : s \to t$ and $COMPILE(t) = t_1 \times \ldots \times t_k$ |
| $\mathsf{map}(f)$ | $FLATMAP(COMPILE(f))$ |
| flatten | $\langle \pi_1, \pi_3, \ldots \rangle$ |
| length | $\pi_k$ |
| get | if $eq1 \circ \pi_k$ then $\langle \pi_1, \pi_3, \ldots \rangle$ else $error$ |
| zip | if $eq \circ \langle \pi_{k+1}, \pi_{2k+2} \rangle$ then $\langle \pi_1, \ldots, \pi_k, \pi_{k+2}, \ldots, \pi_{2k+2} \rangle$ else $error$ |
| enumerate | $\langle \mathsf{enumerate} \circ \pi_2, \mathsf{map}(\lambda x.1) \circ \pi_2, \pi_k \rangle$ |
| split | see text |
| $\mathsf{distribute}_{left}$ | $\ldots$ sbmRoute $\ldots$ (see text) |
| $\mathsf{while}(p, f)$ | $\mathsf{while}(COMPILE(p), COMPILE(f))$ |

Figure 10: Compiling recursion-free CoPa into SA.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $bd$ | | $a\ b$ | $c\ d\ e$ | | $f$ | | $g\ h\ i$ | |
| $sd$ | | 2 | 3 | | 1 | 0 | 3 | |
| $s$ | | 2 | | 0 | 3 | | | |
| $es \leftarrow \mathsf{enumerate}(s)$ | | 0 | | 1 | 2 | | | |
| $u \leftarrow \mathsf{bmRoute}\langle sd, s, es \rangle$ | | 0 | 0 | | 2 | 2 | 2 | |
| $v \leftarrow \mathsf{bmRoute}\langle bd, sd, u \rangle$ | | 0 0 | 0 0 0 | | 2 | | 2 2 2 | 3 |
| $d \leftarrow \Delta(v@[m])$ | | 0 0 | 0 0 2 | | 0 | | 0 0 1 | |
| $i \leftarrow \mathsf{map}(\lambda x.x+1)(\mathsf{enumerate}(bd))$ | | 1 2 | 3 4 5 | | 6 | | 7 8 9 | |
| $w \leftarrow \mathsf{bmRoute}\langle s, d, i \rangle$ | 0 | 5 | | 5 | 9 | | | |
| $sd^{\text{new}} \leftarrow \Delta([0]@v)$ | | 5 | | 0 | 4 | | | |

Figure 11: The function $sdNew\langle bd, sd, s \rangle$ needed to compile split. Here $m = 3$.

| Instruction | Meaning |
|---|---|
| $V_i \leftarrow V_j$ | move instruction |
| $V_i \leftarrow \mathsf{map}(\varphi)(V_j, V_k, \ldots)$ | scalar external functions |
| $V_i \leftarrow \widetilde{agg}(V_j, V_k)$ | aggregates |
| $V_i \leftarrow []$ | empty sequence |
| $V_i \leftarrow [c]$ | singleton sequence |
| $V_i \leftarrow V_j @ V_k$ | sequence concatenation |
| $V_i \leftarrow [\mathsf{length}(V_j)]$ | singleton sequence (length) |
| $V_i \leftarrow \mathsf{bmRoute}(V_j, V_k, V_l)$ | bounded, monotone routing |
| $V_i \leftarrow \mathsf{sbmRoute}\ (V_j, V_k, V_l, V_m)$ | segmented bounded monotone routing |
| $V_i \leftarrow \mathsf{select}(V_j, V_k)$ | selection |
| goto $l$ | unconditional jump |
| if empty($V_i$) then goto$l$ | conditional jump |
| halt | halts the program |

Figure 12: Instruction set for the BVRAM.

and Sabot's techniques can compile while-recursion. However this translation may incur a significant increase in the work complexity because of the additional recursion-unfolding. For example consider a while statement which in the first $n$ iterations have each a work complexity 1, then in the last $\log n$ iterations constructs a sequence $x$ of length $n$ (by starting with a sequence of length 1, then doubling its length at each step), and returns $x$ as a result. The total work complexity is $W = 1 \cdot n + (1 + 2 + 4 + \ldots + n) = O(n)$: our compilation technique would compile this into one with $W = O(n)$ (because there is no map applied to a while). The equivalent tail-recursive function however would have $W = O(n^2)$, because of the additional cost of copying the result when the recursion unfolds: Blelloch and Sabot's techniques would compile the tail-recursive function into one with $W = O(n^2)$.

**Practical Matters** The encoding has a few of nested sequences into flat sequences redundancies. For example a simple flat sequence type [D] is compiled into three flat sequences $\langle$[D], [N], [N]$\rangle$: only the first holds useful information, the others store $[1, 1, \ldots, 1]$ and $[n]$ respectively. We implemented a small CoPa compiler following the techniques described here. We found that a simple static-flow analysis on the generated code can identify and eliminated many unnecessary flat sequences: such an analysis would identify the two sequences above as redundant and eliminate them. We also discovered that several minor changes in the code generation (either $\mathit{COMPILE}(f)$ or flatmap($f$)) can also result in significant savings. For example consider the compilation of length($x$), with $x$ of type [D] as above: we could either compute length of the body (first component), or simply return the third component. The first choice is better, since the second one makes the $[n]$ component non-redundant: this can add up significantly if, for instance, the code is subject to one or more flatmap's. Finally we also optimized $\mathsf{distribute}_{left}\langle x, Y\rangle$ to use bmRoute whenever $x$ is of scalar type.

## 4.4 Generating Code for the BVRAM

The next compilation step generates code for the Bounded Vector Random Access Machine, BVRAM. A BVRAM consists of a fixed number of **sequence registers** $V_1, \ldots, V_r$. Each $V_i$ can hold a flat sequence (a vector) of arbitrary length. To keep the model simple, we do not include *scalar* registers: a number will be represented instead by a sequence of length 1 consisting of that number. BVRAM's instruction set is shown in Fig. 12. Most of the operations are simply inherited from SA.

The conditional jump $\mathsf{if\ empty}(V_i)\ \mathsf{then\ goto}\ l$, takes the jump to the instruction labled $l$ when $V_i$ holds the empty sequence, otherwise continues with the next instruction. The BVRAM has a special set of instructions related to external functions. These follow exactly the external fucnctions in SA: $\mathsf{map}(\varphi)$ for scalar functions $\varphi$, and $\widehat{agg}$. For example $\mathsf{map}(+)(V_j, V_k)$ and $\widehat{sum}(V_i, V_j)$ (the latter meaning $\mathsf{map}(sum)(\mathsf{split}(V_i, V_j))$). As was the case in SA, we can consider more complex external functions too. A BVRAM *program* is a sequence of labeled instructions.

For each BVRAM instruction $V_i \leftarrow instr(V_j, V_k, \ldots)$, except for external functions, we define the parallel complexities $T, W$ to be:

$$
\begin{aligned}
T &\stackrel{\text{def}}{=} 1 \\
W &\stackrel{\text{def}}{=} \mathsf{length}(V_i) + \mathsf{length}(V_j) + \mathsf{length}(V_k) + \ldots
\end{aligned}
$$

that is, the work complexity is the sum of the lengths of all input sequences *and* of the output sequence. With one exception the output is no larger than the sum of the inputs: the exception is sbmRoute where inputs of length $O(n)$ may produce an output of length $O(n^2)$. For external functions (scalar and aggregates) their complexities are those from SA (and CoPa), and they may have $T > O(1)$.

A BVRAM program $P$ takes input sequences in some designated input registers and returns its results in designated output registers. For some input, the result of $P$ might be undefined, if $P$ enters an infinite loop, or if an error occurs. For a terminating execution of $P$, we define the complexities $T, W$ to be the sum of the $T$'s and $W$'s over all instructions executed by $P$.

It is straightforward to generate complexity-preserving BVRAM code for SA. Rather surprisingly, the converse (expressing a BVRAM program as an SA function) is not immediate: we prove it next.

**Proposition 4.10** SA *and BVRAM are strongly equivalent, i.e. any function $f$ in* SA *with time and work complexity $T, W$ can be executed on a BVRAM with complexities $O(T), O(W)$, and conversely.*

**Proof.** The traditional way of simulating a register machine as a single while-loop does not preserve the work complexity (the while statement touches *every* register at each iteration, instead of just the relevant ones). Let $r$ be the number of registers of a BVRAM $M$, and $h$ some function in SA of type $[\mathtt{N}] \times ([\mathtt{N}])^r \rightarrow [\mathtt{N}] \times ([\mathtt{N}])^r$ performing *one step* of the program of $M$ (where the program counter is encoded by a singleton sequence, on the first position). As said, iterating $h$ results in a increased work complexity. To avoid this, we define a sequence of $r$ functions $f_i$, $i = 1, r$. The inputs and outputs for $f_i$ are: the values of the $i$ "smallest" registers, at some particular moment, the indexes of these $i$ registers, the size $S$ of the next largest register, and the program counter. $f_i$ iterates the one-step function as long as it only affects the $i$ registers it sees, and as long as all the $i$ sizes stay less than $S$. If any of these conditions is violated, $f_i$ stops. To do its job, $f_i$ calls $f_{i-1}$, which iterates steps on $M$ by only looking at the smallest $i-1$ registers: when $f_{i-1}$ finishes, $f_i$ tries to do one more step by taking into account the $i$'s smallest register as well, which $f_{i-1}$ ignores. If it cannot, then it returns (to $f_{i+1}$). Else, it performs the operation, and calls $f_{i-1}$ again, possibly with a different set of $i-1$ registers, from the set of $i$ registers it sees. $\square$

Although only one direction of Proposition 4.10 is actually needed for the Compilation Theorem 4.1, the converse is significant from the point of view of optimizations: it implies that any optimizations done for the BVRAM can also be performed at the level of the SA language.

## 4.5 Application: Lower Bounds for CoPa's Complexity

Using the Compilation Theorem 4.1 we prove here that CoPa is a communications-aware language. Given an arbitrary sequence $[x_0, \ldots, x_{n-1}]$, CoPa does not allow us to construct an arbitrary permutation $[x_{i_0}, \ldots, x_{i_{n-1}}]$, unless we pay a certain price in the complexities $T, W$ corresponding to the communication cost needed to achieve such a permutation on a shared-nothing multiprocessor.

The permutations subject to the lower bound are:

$$\begin{aligned} \mathit{reverse} &: [\mathtt{D}] \to [\mathtt{D}] \\ \mathit{tr} &: [\mathtt{D}] \to [\mathtt{D}] \end{aligned}$$

with the following meanings:

$$\begin{aligned} \mathit{reverse}([a_0, a_1, \ldots, a_{n-1}]) &= [a_{n-1}, \ldots, a_1, a_0] \\ \mathit{tr}([a_{00}, a_{10}, \ldots, a_{(\sqrt{n}-1,0)}, \ldots a_{(\sqrt{n}-1,\sqrt{n}-1)}]) &= [a_{00}, a_{01}, \ldots, a_{(0,\sqrt{n}-1)}, \ldots, a_{(\sqrt{n}-1,\sqrt{n}-1)}] \end{aligned}$$

We show that they cannot be computed *uniformly* in $T = O(1), W = O(n)$. By "uniformly" we mean with no particular knowledge about the elements of type D. Without the uniformity condition, for particular inputs we may compute *reverse* and/or *tr*: e.g. we can obviously reverse the sequence $[0, 1, \ldots, n-1]$ in $T = O(1)$ and $W = O(n)$, by subtracting each element in the sequence from $n-1$.

We make the uniformity condition precise as follows. A BVRAM is **uniform** in the type D if no external function has type $p : t_1 \times \ldots \times t_n \to \mathtt{D}$. When $\mathtt{D} = \mathtt{N}$, then no BVRAM would be uniform in D, since we always assume the existence of external functions like $+ : \mathtt{N} \times \mathtt{N} \to \mathtt{N}$. Hence we shall assume $\mathtt{D} \neq \mathtt{N}$ in the sequel.

**Theorem 4.11** *Let $M$ be a BVRAM which is uniform in* D*. Then any algorithm for computing reverse or tr in linear work complexity ($W = O(n)$) on $M$, requires at least $T = \Omega(\log \log n)$ parallel steps.*

The proof is given in Appendix B

We can apply now the complexity-preserving compilation and obtain:

**Corollary 4.12** *Neither reverse nor tr can be expressed uniformly in* CoPa *with $T = O(1), W = O(n)$.*

**Proof.** Indeed, suppose we could. Then using the Compilation Theorem we obtain a BVRAM program computing *reverse*, or *tr*, with $T = O(1), W = O(n)$, contradiction. □

# 5 Architecture-Dependent Implementations

## 5.1 Implementing BVRAM on a Butterfly Network

The butterfly network is a bounded-degree, $\log n$-diameter network. Communication algorithms have been studied extensively for the butterfly network: we refer the reader to [46], whose terminology we follow here. Our goal is to prove that every BVRAM instruction admits an *optimal* implementation on a butterfly network. We believe this, together with the LogP implementation described later, to be evidence for our claim that BVRAM instructions are simple to implement on a large variety of architectures.

The nodes in butterfly network are arranged in a matrix with $n$ rows and $\log n$ columns. Traditionally input data items reside in the left-most column, and a computation proceed in left-right sweeps and back, each sweep taking $\log n$ steps. Our BVRAM implementation follows the same principles: all input sequences are stored in the left-most column. We assume the input sequences to be aligned: that is, if some instruction expects three input sequences $X, Y, Z$, then $X_i, Y_i, Z_i$ will all be in the same row. This requirement is not essential: alignment can be achieved in $O(\log n)$ steps. Throughout this subsection, *optimal* implementation means a constant number of sweeps, i.e. $O(\log n)$ steps. To simplify the presentation we assume that $n$, the number of rows, is larger than the length of any input or output sequence. When that is not the case sequences can be wrapped around, and one can show that the algorithms will be slowed down only by a factor equal to the number of wrappings: we omit the details. Also, we assume that the BVRAM has only scalar external functions, and that these have complexity $T = O(1)$.

**Theorem 5.1** *Any BVRAM instruction of work complexity $W$ can be implemented optimally (i.e. in $O(\log W)$ steps) on a butterfly network. Moreover, all communications are oblivious.*

**Proof.** (Sketch) The scalar external functions involve no communication at all, since input sequences are aligned, thus are implemented in $O(1)$ steps. The append operation $V_i \leftarrow V_j @ V_k$ only requires a *monotone routing*, (in the sense of [46, pp. 534]), which can be done in $O(\log n)$ steps, using the greedy routing algorithm. The $\mathsf{bmRoute}$ instruction requires a prefix sum followed by a routing: the routing is monotone again, hence $O(\log n)$ steps suffice. Only $\mathsf{sbmRoute}(x', s', x, s)$ involves a non-monotone routing, and we explain it next. Recall the meaning of $\mathsf{sbmRoute}$: both $x', s'$ and $x, s$ are viewed as nested sequences $X', X$, with segment descriptors $s', s$, and they must have equal length (hence $\mathsf{length}(s) = \mathsf{length}(s')$). Then each $X_i$ is replicated a number of times equal to $s'_i = \mathsf{length}(X'_i)$. For example, when $X' = [u_0, \ldots, u_4]$, then $\mathsf{sbmRoute}(X', [2,3], [a_0, a_1, a_2, b_0, b_1], [3,2])$ is $[a_0, a_1, a_2, a_0, a_1, a_2, b_0, b_1, b_0, b_1, b_0, b_1]$. Assume first that $\mathsf{length}(s) = \mathsf{length}(s') = 1$. Then a single sequence, $X_0 (= x)$, has to be replicated $s'_0$ times. Start with the simpler case when both $\mathsf{length}(x)$ and $s'_0$ are powers of 2: say $2^p$ and $2^q$. Then the output has length $2^{p+q}$, so we may assume to have $n = 2^{p+q}$ many rows. The communication problem is the following. $2^p$ data items ($x$) reside in the first $2^p$ rows. Each data item with index $i$ has to be sent to addresses $i, i + 2^p, i + 22^p, i + 32^p, \ldots, i + 2^{q-1}2^p$. This can be achieved in a single sweep through the network by a simple greedy, oblivious algorithm. When $\mathsf{length}(x)$ is not a power of 2, then we first pad it with dummy elements up to the closest power of 2, and run the same greedy algorithm. The result has now several dummy data items: we eliminate these using a "packing" routing [46] which runs in $O(\log n)$ steps. (The case when $s'_0$ is not a power of 2 is even simpler). Now consider the general case, when $X', X$ have arbitrary lengths. Let $n_i \stackrel{\text{def}}{=} \mathsf{length}(X'_i)\mathsf{length}(X_i)$: the meaning is that after replication, the subsequence $X_i$ will occupy $n_i$ rows. We round up each $n_i$ to the nearest power of 2. The communication algorithm proceeds as follows. It starts by "spreading" the subsequences $X_0, X_1, \ldots$ of $X$ (i.e. it introduces gaps between $X_i$ and $X_{i+1}$) such that each $X_i$ starts at row $n_0 + n_1 + \ldots + n_{i-1}$: the idea is that each $X_i$ now "owns" $n_i$ rows. "Spreading" takes $\log n$ steps [46]. Next it applies the previous algorithm locally to each $X_i$, in order to replicating it $\mathsf{length}(X'_i)$ times. This is done in parallel, exploiting the recursive nature of the butterfly network: due to the spreading these parallel computations do not interfere. Finally it packs the results, by eliminating the gaps due to rounding the values $n_i$ to a power of 2. □

We illustrate the power of Theorem 5.1 with an application. It is not known whether $n$ items can be sorted on an $n$-node hypercube in $O(\log n)$ parallel steps [46]. However, $n$ items can be sorted in $O(\log n)$ parallel steps on an $N$-node hypercube, when $N = n^{1+\varepsilon}$ (see [46] Section 3.5.2; and Preparata [58]). We can derive an algorithm with the same complexity for a butterfly network as follows:

1. First consider the sorting algorithm *sort* of Subsection 4.1 with complexities $T = O(1), W = O(n^{1+\varepsilon})$.

2. Compile it using Theorem 4.1 to get a BVRAM program with the same time and work complexities.

3. Apply Theorem 5.1: the result is a $T' = O(T \log W) = O(\log n)$ time sorting algorithm on a butterfly network with $W \log W = n^{1+\varepsilon} \log n = n^{1+\varepsilon'}$ nodes, with $\varepsilon' > \varepsilon$.

## 5.2 The LogP Model

The LogP model of parallel computation was introduced by Culler et al. [22] with the goal of capturing accurately the communication costs in parallel computers. The main purpose of the model is to serve as basis for the design and analysis of parallel algorithms, without encouraging loopholes existing in other parallel models. The LogP consists of a fixed number $P$ of processors, each with local memory, connected by a network allowing point-to-point communications. The parameters $L$, $o$, and $g$ define the communication network. The *latency* $L$ is an upper bound on the delay of delivering a word of a message: a word sent at time $t$ will arrive at destination no later than $t + L$. The *gap* $g$ is the amount of time a node needs to wait between sending or receiving two consecutive words of a message: if a processor has sent a word at time $t$ it may not send the next word until $t + g$ (it takes $(s-1)g$ time to send a message of size $s$). The *overhead* $o$ is the time a processor spends in preparing the network for sending or receiving one message: a processor cannot do computations, or prepare another message when spending this overhead. We show here that a BVRAM admits a near-optimal implementation on the LogP model. By *near-optimal* here we mean with a running time no larger than some lower bound plus a constant: this is much stronger that the asymptotic $O(\ldots)$ notation, and in spirit with the optimal algorithms described in [42].

Some details of the LogP model are not discussed in the original presentation, and do not affect the algorithms in [22, 42]. However they become important in our implementation of the BVRAM, so we discuss them here. First we allow variable-length messages instead of fixed-length ones (this option was considered in [22] too). Second we will allow a node to send *and* receive a message simultaneously. This was not an issue in the original work because the messages where short, and, in addition, in all algorithms discussed the *send*s at each node always had to wait for data from the preceding *receive*. In our setting however neither restriction applies, and allowing or disallowing a *send* to overlap with a *receive* makes a huge impact on the algorithm: we assume here that each node has a separate input and output port, thus allowing send/receive overlap. Finally, long messages raise the question of when the receiving processor spends the overhead $o$: when the message starts arriving or when it is completed ? We make the latter choice, which corresponds to the case when each node has a network DMA. We assume each incoming message to contain in its header the destination memory address, so the DMA can deliver the message to the right address and interrupt the processor only at completion (when the processor spends the overhead $o$ to process the message). This assumption is realistic in our setting because a BVRAM has only a fixed number of registers and, hence, there are only a fixed number of possible destinations for each packet.

Fig. 13 illustrates in detail the steps of sending a message of $s$ words from one processor to another. Both the sender and the receiver spent $o$ units of time each for this communication: they are free

| Time | Activity at sender |
|---|---|
| $t$ | processor busy |
| $t+o$ | send word 1 |
| $t+o+g$ | send word 2 |
| $t+o+2g$ | send word 3 |
| $\ldots$ | $\ldots$ |
| $t+o+(s-1)g$ | send word $s$ |

| Time | Activity at receiver |
|---|---|
| $t+o+L$ | receive word 1 |
| $t+o+L+g$ | receive word 2 |
| $\ldots$ | $\ldots$ |
| $t+o+L+(s-1)g$ | receive word $s$<br>processor busy |
| $t+o+L+(s-1)g+o$ | ready |

Figure 13: Sending a message of size $s$ in the LogP model

to perform other computations otherwise. The sender will be able to send the next message at the time $t+o+sg$, while the receiver will be ready to receive the next message at the time $t+o+L+sg$.

In the particular case when $g=1, o=0$ the LogP is called the *postal model* [6]. Moreover, every LogP model with $o=0$ is equivalent to a postal model, because one can normalize $g$ to 1.

**Data Partitioning** Each BVRAM register will be, in general, much longer than $P$. The process of splitting a large collection on several nodes is called *data partitioning* [27, 26, 73], or *declustering*. Current parallel database systems use a variety of partitioning techniques [27]. Here we have to partition sequences (as opposed to sets), and use a new method: *block partitioning*[6]. A sequence $X$ is split into $P$ continuous subsequences, $X_1, \ldots, X_P$, and stored in consecutive nodes. We call $X$ the *global sequence* and each subsequence a *local sequence*. In a block partitioning the global sequence is always the concatenation of the local ones. We also let each node know the length of the global sequence. To achieve good load balance we always strive to do *uniform partitioning*, i.e. to have the local sequences of the same lengths: when $\mathsf{length}(X)$ is not divisible by $P$, then we let the first $\mathsf{length}(X)$ *mod* $P$ local sequences have one more element than the others. In the implementation of a BVRAM instruction some intermediate results may be non-uniformly partitioned. However, at the beginning and end of each BVRAM instruction all BVRAM registers are uniformly partitioned.

**Control Communications** Karp et al. [42] present provably optimal LogP algorithms for broadcast and prefix sum (the latter is implicit in their summation algorithm). In broadcast, one node holds one word and has to send it to all other $P-1$ nodes. In prefix-sum each node $i$ holds one integer word $x_i$, and the purpose is to compute at each node the sum $x_1+x_2+\ldots+x_i$. For both problems the given algorithms are absolutely optimal (i.e. not within a constant factor), and their running time is given by an exact (but complex) formula which, asymptotically, is $O(\log P)$. In the BVRAM implementation we make use of both algorithm: they will form the *control communications*.

**Data Communications** In addition, each BVRAM instruction needs *data communications*, and one expects these to account for bulk of the cost of the BVRAM implementation. These communications are of a different nature than broadcast and summation treated in [42]. In the case of BVRAM, all data communications are *monotone communications* (except $\mathsf{sbmRoute}$). A data communication is called *monotone* if the element order in the global sequence before and after the communication is the same. Fig. 14 illustrates both a monotone and a non-monotone communication. We describe in the next subsection a greedy algorithm for the monotone communication problem whose running time is either optimal (for one-to-one communications), or larger than the

[6]While not discussed in [27], block partitioning is equivalent to round-robin of [27], in the case of sets, where order does not matter.

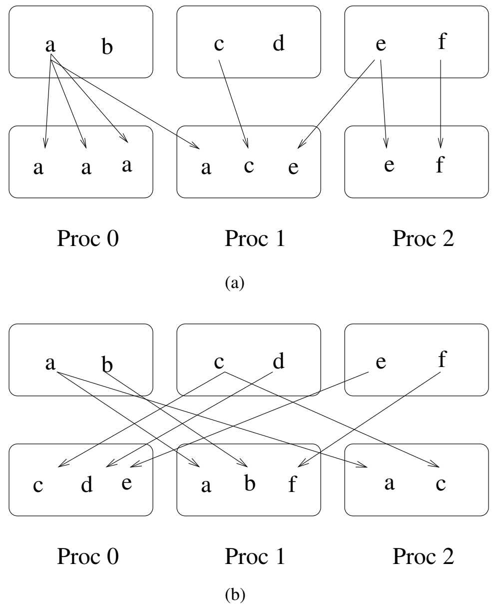


Figure 14: A monotone and a non-monotone communication

optimal by at most the cost of one broadcast operation: we call such an algorithm *near-optimal*. These monotone-communication algorithms form the bases of our BVRAM implementation on LogP.

**Implementing BVRAM** Every BVRAM instruction, except sbmRoute, can be implemented as a sequence of (a) local computations, (b) control communications and (c) monotone data communications: sbmRoute is the only one requiring non-monotone data communications. We illustrate here a few:

- $z \leftarrow x@y$. This is implemented by exactly two monotone communications: one to shift the $x$'s left, the second to shift $y$'s right. Fig. 15 (a) illustrates the first of the two.
- bmRoute: it starts with a prefix-sum over the segment descriptors, followed by a monotone one-to-many communication (Fig. 14 (a) could be the data communication for a bmRoute)
- External functions. In our setting, all external functions are sequential. Our concern in BVRAM is for expressions of the form $\mathsf{map}(f)(X)$, which have to be parallelized. When $f$ is a scalar function and takes $T = O(1)$, then we can run $f$ locally on each processor, and obtain a good load balance (since $X$ was uniformly distributed). The interesting case is when $f$ is not scalar, say when $X$ is a nested sequence, given as $X', S$ (with $S$ the segment descriptors and $X' = \mathsf{flatten}(X)$ the body). All aggregate BVRAM instructions $\widetilde{agg}(V_j, V_k)$ fall into this category. Here we want to run $f$ sequentially over each subsequence of $X$, but the problem is that some subsequences may cross node boundaries, since $X'$ was distributed uniformly. Hence, we redistribute $X'$ fist, such that no subsequence crosses node boundaries: we call this process *clustering*. Then run $f$ sequentially at each node, on all subsequences at that node and finally decluster the result. Thus, during intermediate steps, the sequence $X'$ may be distributed unevenly on the nodes. We need to do clustering in such a way that the subsequent executions of $f$ are well balanced. We describe this in more detail. Let $X = [X_0, X_1, \ldots, X_{n-1}]$ and $c_i = T(f, X_i)$ (the running time of $f$ on $X_i$). Our assumption for external functions is that we

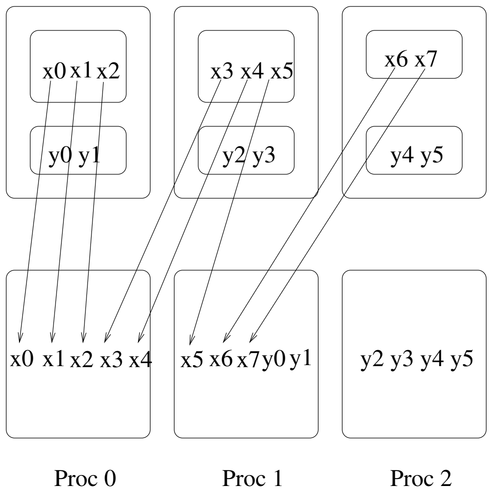


Figure 15: First step in the implementation of $x@y$.

can always compute this cost from $\mathsf{length}(X_i)$ (and $\mathsf{length}(X_i)$ is in the segment descriptor $S$). Define $s_i \stackrel{\text{def}}{=} \sum_{j=0,i-1} c_j$ and $p_i \stackrel{\text{def}}{=} \lfloor \frac{s_i P}{s} \rfloor$: clustering will send the entire sequence $X_i$ to node $p_i$. This ensures that the load at each node is no more than $\max_{i=0,n-1} c_i$ higher than that at any other node.

In summary:

**Proposition 5.2** *All BVRAM instructions except* sbmRoute *can be implemented on the LogP model as combinations of provably near-optimal communication steps.*

## 5.3 Near-Optimal Monotone Communications on the LogP Model

Recall that in a monotone communication each node holds an input local sequence, whose elements have to be sent to zero, one, or more nodes. Once elements arrive at a node they are assembled in an output local sequence. Referring to Fig. 14 (a), the input local sequences are $[a,b],[c,d],[e,f]$, while the output local sequences are $[a,a,a],[a,c,e],[e,f]$.

Throughout this subsection we denote with $X$ the input global sequence, and with $Y$ the output global sequence. The communication is called *monotone* if the order of elements in $Y$ is the same as that in $X$. Note that there is no requirement for either $X$ or $Y$ to be uniformly distributed on the nodes. We denote with $x_i$ and $y_j$ the input and the output elements respectively, i.e. $X = [x_0, x_1, \ldots, x_{m-1}], Y = [y_0, y_1, \ldots, y_{m-1}]$. We can view the monotone communication as requiring each $x_i$ to be "sent" to some continuous subset of elements in $Y$: $y_j, y_{j+1}, \ldots, y_k$ (in fact $x_i$ will not be sent to $y_j$ but to the node holding $y_j$, etc.). We assume throughout this subsection that the node holding $x_i$ knows (1) the indexes $j, j+1, \ldots, k$ of the global output sequence $Y$ where $x_i$ has to be sent to, (2) the node numbers holding each of these output elements, and (3) the offsets where the output elements occur in their local sequence. For all monotone communications required in the BVRAM implementation these informations are either directly available, or computable using a prefix sum.

$$\overbrace{\underbrace{\boxed{x_{k_0}, x_{k_0+1}, \ldots, x_{k_1-1}}}_{\text{To node } j_0} \underbrace{\boxed{x_{k_1}, \ldots, x_{k_2-1}}}_{\text{To node } j_1} \cdots \underbrace{\boxed{x_{k_{\lambda-1}}, \ldots, x_{k_\lambda-1}}}_{\text{To node } j_{\lambda-1}} \underbrace{\boxed{x_{k_\lambda}, x_{k_\lambda+1}, \ldots}}_{\text{To node } j_\lambda}}^{\text{node } i}$$

Figure 16: The *send*-blocks in the optimal monotone, one-to-one algorithm

In a monotone communication some data elements $x_i$ have to be "send" to the same node (possible in addition to other nodes): for example, element $c$ in Fig. 14 (a). We assume that each node detects such elements in a preprocessing step and will not consider them as part of the communication problem: the node may still need to copy those elements locally to their final destination in the output sequence $Y$. Hence in a monotone communication each data element has to be sent to a different node.

In the sequel *optimal* means absolute minimal running time, while *near-optimal* means at most the cost of one broadcast communication more than the absolute minimal running time. An algorithm is *local* if all its decisions at a node are based only on the data available at that node.

**Theorem 5.3** *Suppose $3o \leq sg$, where $s$ is the size of the packets. (1) There exists an optimal, local algorithm for the monotone one-to-one communication problem. (2) There exists a near-optimal, local algorithm for the general monotone communication problem.*

**Proof.** We start with (1) and describe the algorithm at node $i$. Let $x_{k_0}$ be the first outgoing packet at node $i$. Divide all outgoing packets at $i$ into blocks according to their destination nodes, see Figure 16.

Because the communication is monotone, the nodes $j_1, j_2, \ldots, j_{\lambda-1}$ will not receive any other messages except from node $i$: hence there will be no message contention at these destinations, no matter in which order node $i$ sends these messages. Only the first and last block should be send out with care, to avoid contention at nodes $j_0$ and $j_\lambda$ with messages coming from nodes $i-1, i-2, \ldots$ and $i+1, i+2, \ldots$ respectively. Note that when $\lambda = 0$, then both nodes $i-1$ and $i+1$ may send messages to $j_0$. Our strategy to avoid contention is to arrange for the messages arriving at any destination $j$ to arrive from left to right. For this, it suffices for every node $i$ to send its blocks in the right-to-left order, i.e. to nodes $j_\lambda, j_{\lambda-1}, \ldots, j_1, j_0$. Moreover, $i$ may proceed at full speed, except for the last block $j_0$: here it needs to wait first for node $i-1$ to finish sending its messages to $j_0$. We explain the details below.

The algorithm at each node consists of two threads: a send-thread and a receive-thread. The send-thread is described in Figure 17.

The idea is to delay sending the first block to node $j_0$ until all node to the left have sent their packets to node $j_0$: this takes time $t' + o = dg + o$, where $d$ is equal to the number of packets which have to arrive at $j_0$ *before* those from node $i$.

We prove optimality by identifying a *critical node* in the algorithm: a node which is busy all the time during the execution, doing necessary work on its local data, thus proving that in any other algorithm that node has to run at least as long as in ours. First we consider the postal model ($o = 0$), where the messages don't need to be received explicitly, hence there is no need for a receive-thread. Let $i$ be the process sending out the last word of the last message, at time $T - L - o$ (which is $T - L$, because $o = 0$), where $T$ is the running time of the algorithm. There are two cases: (1) node $i$ didn't have to wait at line W (that is $t' \leq t$). Then node $i$ has sent out its messages at full speed:

procedure *send-thread*
  for $b = \lambda$ to 1 do
    *send* block $b$ to node $j_b$

  let $t = g\cdot$ (size of all blocks sent above)
  let $d =$ the offset of $x_{k_0}$ at the destination node
  let $t' = dg$
  W: if $t' > t$ then wait $t' - t$ units of time else continue
  *send* block 0 to node $j_0$
end

Figure 17: The *send*-thread in the optimal monotone, one-to-one algorithm

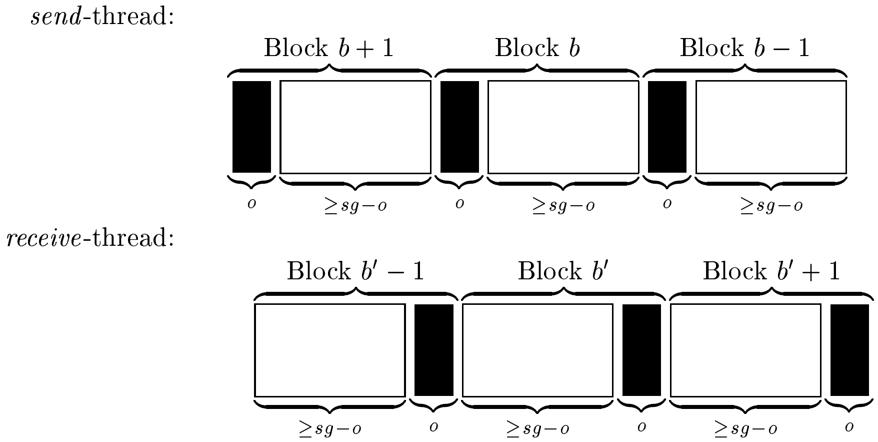


Figure 18: Possible interference between the *send*-thread and the *receive*-thread

it is a critical node, and therefore $T$ is the optimal running time. (2) node $i$ had to wait. Then let $j$ be the destination processor of the last message from $i$. Here we argue that $j$ receives his incoming messages at full speed, hence it is a critical node, and our algorithm is again optimal.

For a LogP model with $o > 0$, the processors have to spend an additional overhead $o$ to receive each incoming message, and the problem is that this overhead may compete with the send-thread. We design the receive-thread to accept incoming messages eagerly, but without disturbing the send-thread. That is the receive-thread will not initiate receiving a new message if the time left before the send-thread sends its next message is $< o$; in that case it will wait until the send-thread completes the send, and then it will proceed receiving the next message. REfering to Fig. 18, this means that the receive-thread shifts the black boxes to the right as far as necessary in order to overlap with a black box of the send-thread. By inspecting the figure one can see that, when $3o \leq sg$, then this does not disturb the next black box in the receive-thread: indeed each black box needs to be shifted to the right at most $2o$ units, while the distance to the next black box is $\geq sg - o \geq 2o$. Hence this "shifting" will not affect the last black box in the critical node: we discuss this next.

We analyze the running time using the same critical-node argument as in the postal model. Namely let $i$ be the node sending out the last word of the last message at, say, time $t$. As before, we distinguish two cases. (1) The send-thread of node $i$ didn't wait at all. Then node $i$ has sent out its messages at full speed. Moreover, the destination node $j$ of the last message will receive its last word at time $t + L$: the send-thread on node $j$ has finished before that, so the receive-thread will handle this message without delay, hence node $i$ is a critical node, and the algorithm is optimal. (2) The send-thread of node $i$ had to wait before sending out its last message. Let $j$ be the destination node of the last message. As in the postal model case, we argue that $j$ has received messages at full speed (it may have processed them with some delay). We only have to argue that the last message was processed without delay. But this is obvious, because by the time the last word of the last message arrives, all outgoing messages from node $j$ have been sent already. So node $j$ is critical.

Now we prove (2) (one-to-many monotone communications). We note that each general monotone communication can be decomposed into a one-to-one communication followed by a series of parallel broadcasts (the elements with more than one destination need to be broadcast). In addition, each processor may be required to do some local replication of some element(s): we assume this cost to be negligible compared to the communication cost, and do not account for it. Hence we only have to show that the final parallel broadcasts can be implemented at cost no higher than one single global broadcast. This is obvious since the local broadcasts have almost no interference. Recall from [42] that the broadcast algorithm logically arranges the nodes in a tree, with the root holding the item to be broadcast. The only possible overlap between our multiple broadcast trees is that some node may be the root in one tree and simultaneously the last leaf in another tree. But the root performs only *send* operations, while the last leave performs a single *receive* operation, and hence can be overlapped.

□

The condition $3o \leq sg$ is realistic. For existing multiprocessor architectures, $o$ typically no larger than $g$, at worst $o \cong 2g$, so it suffice to assume that the packets have size $\geq 6$ words. If this condition is not satisfied then our algorithm has problems performing the *sends* and *receives* simultaneously.

## 5.4 Experiments

In order to test the feasibility of our LogP implementation techniques, we run two preliminary experiments for two simple benchmarks using a bare-bones CoPa compiler and a LogP simulator. The compiler takes CoPa front-end syntax and generates BVRAM code using the techniques described in Subsec. 4.3. It performs only two optimization: dead code elimination, which resulted in significant savings (as explained at the end of Subsec. 4.3), and common subexpression elimination. It does not implement union types. We implemented the BVRAM instructions on LogP, as outlined in subsections 5.2, 5.3: for sbmRoute we used a straightforward algorithm. Finally we ran the benchmarks on a LogP simulator. All three components (compiler, BVRAM implementation, LogP simulator) were written in ML using the SML-NJ compiler.

We fed the simulator with the $L$, $o$, and $g$ parameters for the Connection Machine 5 ($L = 60, o = 105, g = 100$, measured in clock units) and the SP1 multiprocessors ($L = 70, o = 120, g = 55$) [9]: we observed no changes in overall behavior, so we report here only the experiments with the parameters for the CM5. Our primary goal was to compare the cost of communications to that of the "useful" computation, and the cost of control communications and to that of data communications.

We ran two benchmarks: a simple *merge* algorithm, and an object-oriented flavored benchmark, inspired from the OO7 benchmark of [19].

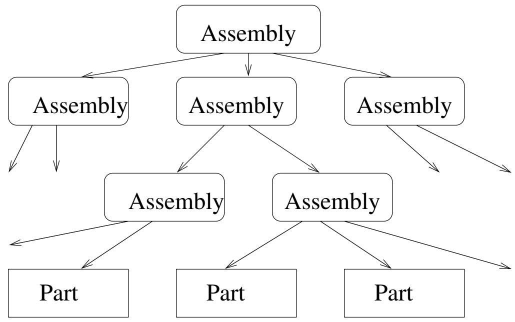


Figure 19: An example of an assembly hierarchy of depth 3 and its parts

The *merge* function is a pragmatic adaptation of Valiant's parallel merge algorithm [74, 40]. Two sorted sequences $x = [x_0, x_1, \ldots, x_{m-1}]$ and $y = [y_0, y_1, \ldots, y_{n-1}]$ are given as inputs. In order to merge them, our algorithm starts by dividing $x$ into $\sqrt{m}$ subsequences of length $\sqrt{m}$: $x = x^0 @ x^1 @ \ldots @ x^{\sqrt{m}-1}$. In parallel for each subsequence $x^i$, it finds the corresponding subsequence $y^i$ of $y$ with which $x^i$ has to be merged. Then each $x^i$ is merged with $y^i$ by an external, sequential *sequential-merge* function. (Valiant's algorithm continues recursively, for a total of $O(\log \log m)$ parallel steps.) This single divide-step generates enough subproblems ($\sqrt{m}$) to allow good processor balance.

Our second benchmark has an object-oriented flavor and is inspired from the OO7 benchmark of [19]. It consists of an *assembly hierarchy* and a collection of *parts*, Figure 19. Each assembly could be a *composite assembly* or a *base assembly*. A composite assembly contains a number of *assembly components*, while a base assembly contains a number of *parts*. Each assembly and each part has a binary information associated with it: this could be a large text file, or an image, etc. The benchmark has two parts. The first traverses the assembly hierarchy and applies an external predicate *inspect* to the binary data stored in each assembly: the predicate decides whether the assembly is *good* or *bad*. The second phase applies another external function *adjust* to all parts which are direct or indirect members of a "bad" assembly.

The data sets for the *merge* benchmark were generated such that $x$ is uniformly distributed in the interval $[x_0, x_{m-1}]$, while $y$ is uniformly distributed in the middle third of $x$. This implies that about 2/3 of the $y^i$'s will be empty, thus testing the ability of the BVRAM implementation to load-balance the calls to *sequential-merge*. A worse distribution for our simple merge algorithm would be when less than $P$ $y^i$ are non-empty: it is possible however to extend the algorithm to a second divide step applied in parallel to each pair $\langle y^i, x^i \rangle$, which would guarantee that there are no bad cases [64].

The data sets for the object-oriented benchmark were chosen as follows. The depth of the assembly hierarchy was kept constant ($h = 4$), in order to facilitate comparisons between different experiments: a deeper hierarchy would require more iteration steps. All base assemblies (tree leaves) are at the same depth in the hierarchy, i.e. the assembly tree is perfectly balanced. The flawed assemblies represent 25% of all assemblies and are chosen uniformly from the prefix order of the assembly tree. The root assembly is never flawed: otherwise we would have a trivial data set in which all parts need processing in the second half of the benchmark. Finally the the size of the binary data associated to each assembly and each part is constant: 10 words.

Each point in the graphs represent a single run: this is because the simulator is deterministic and we used a single data set, hence subsequent runs would have produced the same results. The simulator ignores other factors besides communication and computation time, such as I/O cost, start-up time, system load, etc.

We ran each experiment in two modes. In the *communication-only* mode, the simulator counts only the communication time, and ignores the computation time, except for the communication overhead $o$: this is the same as assuming infinite processor speed, hence we obtain 0 running time for $P = 1$ node. This mode is useful for measuring the communication costs. In the *communication and computation* mode, the computation time (which is estimated) is taken into account too. There are two components of the computation time which are counted in this mode. First there is the computation time of the external functions. This time strictly decreases as $P$ increases. Second, there is the computation time spent in preparing the communications for the BVRAM operators: this is in some sense "unuseful" work, since for $P = 1$ this time is 0.

We tested two performance metrics *speedup* and *scaleup* [27]. In the speedup experiments we kept the data size constant, and varied $P$ from 1 to 64: for *merge*, both $x$ and $y$ have length 100,000, while for the *assembly-hierarchy* benchmark we had height $h = 4$, fan-out $d = 16$ (for a total of $na = 4,369$ assemblies); there are $np = 40,000$ parts, and 25% of all assemblies are "bad". In the scaleup experiments for *merge* we varied $P$ again from 1 to 64, and chose the length of $x$ and $y$ to be $10,000 \cdot P$. For *assembly*, we kept the depth of the assembly hierarchy constant, and varied the fan-out $d$. This gave us a total number of assemblies $na = \frac{d^h-1}{d-1}$, which varied with $P$ as shown below:

| $P$ | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| $d$ | 10 | 13 | 16 | 20 | 25 | 32 |
| $na$ | 1,111 | 2,380 | 4,369 | 8,421 | 16,276 | 33,825 |

Thus $na \approx 1,100P$; the number of parts is $np = 2,000P$.

We describe the experiment results next.

The speedup of *merge* in Figure 20 reveals a nice surprise: the communication time decreases with $P$. For $P = 1$ there are no communications at all. When $P$ increases, the time for the control communications is $O(\log P)$, forcing the total communication time to increase. But at the same time each processor has to send less data messages, because the input data size is constant, hence total communication cost decreased with $P$. Had we increased $P$ even further we would have observed an increase of the communication time, because it starts being dominated by the control communications. Of course, the larger the data sets, the more we can increase $P$ while still observing a decrease in communication costs.

Once we account for the computation time too, the graph in Figure 20 shows a good overall speedup.

The second graph in Figure 20 shows the total number of words sent. It reinforces the observation about the communication time being dominated by the data communications. Indeed the total number of words in the data messages is constant (because the input size is constant), while that in the control messages increases ($P - 1$ for *broadcast*, $O(P \log P)$ for *scan*): the graph shows that the total number of words sent grows slowly with $P$, hence the constant component is significant. The other two plots in this graph reveal a negative phenomenon of our implementation on the LogP model. As $P$ grows, more and smaller data messages are exchanged: the total number of messages grows linearly with $P$.

In the scaleup experiment, Figure 21, the communication time always increases. This is normal, because the cost of data communications per node remains roughly constant, while that of control communications increases. Still the growth is not significant as long as the communication phase is dominated by the data messages, as in the range $P = 4 \ldots 32$. For $P = 64$ however, the communi-

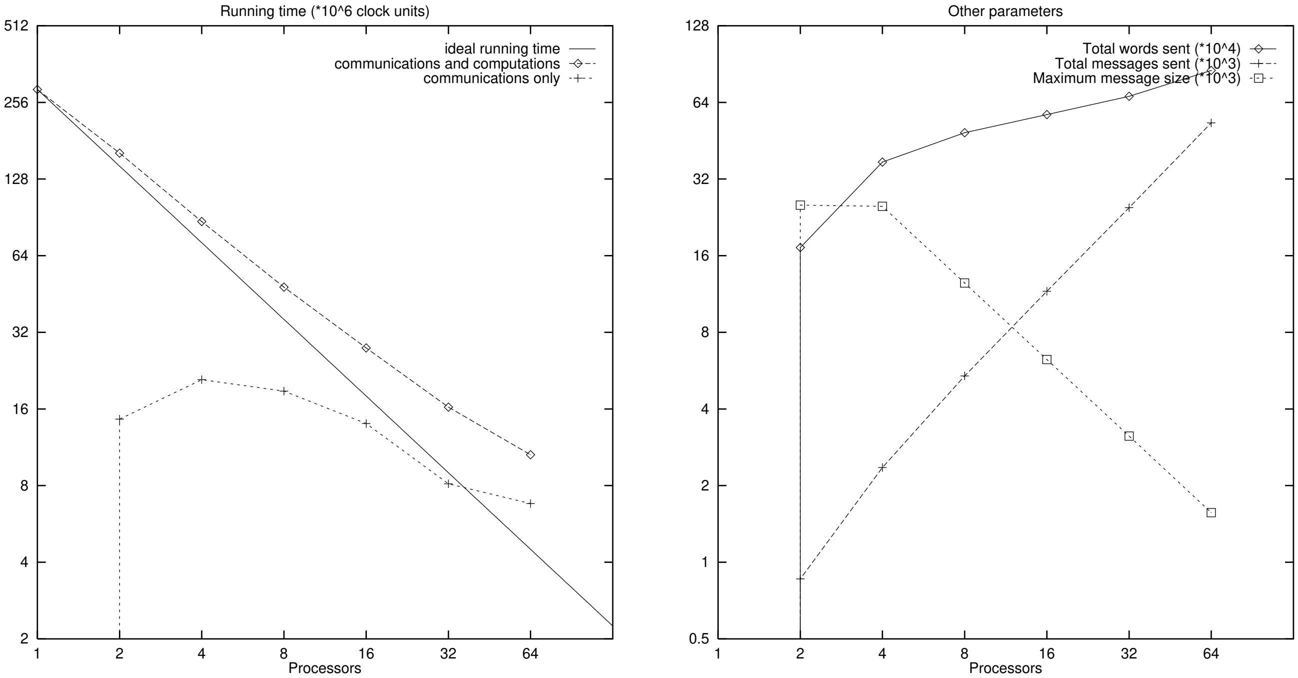


Figure 20: Speedup of $merge(x, y)$.

cation time starts getting dominated by the control messages, and it grows sharply. The maximum message size shown in Figure 21 stabilizes at $10,000$, which means that, at some point, a processor sends all its $x$ or $y$ subsequence to another processor. Unlike in the speedup experiment, the total number of words sent increases linearly both for the data messages and for the control messages.

The speedup experiment for the OO benchmark, Figure 22 shows the same behavior as in the speedup experiment for *merge*. Here the communication-only time decreases too, but less sharply.

The larger gap between the communication time and the ideal computation time shown in the scaleup experiment of Figure 23 is due to the relatively high cost of the external methods *inspect* and *adjust*. In general, applications with costly external methods are easier to speed up or to scale up than pure relational database applications.

# 6 Future Work

We believe the following issues to be important in future work.

**Sequence Optimizations** Database query optimizations are performed at the level of query execution plans. The optimizer chooses among several equivalent execution plans the one with smallest cost. There are two major components to an optimizer: the algebra (describing the operators of the physical plans) and a cost model. Several algebras for sets and bags have been considered in the context of optimizers; the BVRAM seems a promising choice for a sequence optimizer. Many equations for sets and bags operators carry over naturally to those BVRAM, and BVRAM's novel operations ($\mathsf{bmRoute}, \mathsf{sbmRoute}$) seem to be amenable to similar kinds of equations. Designing a

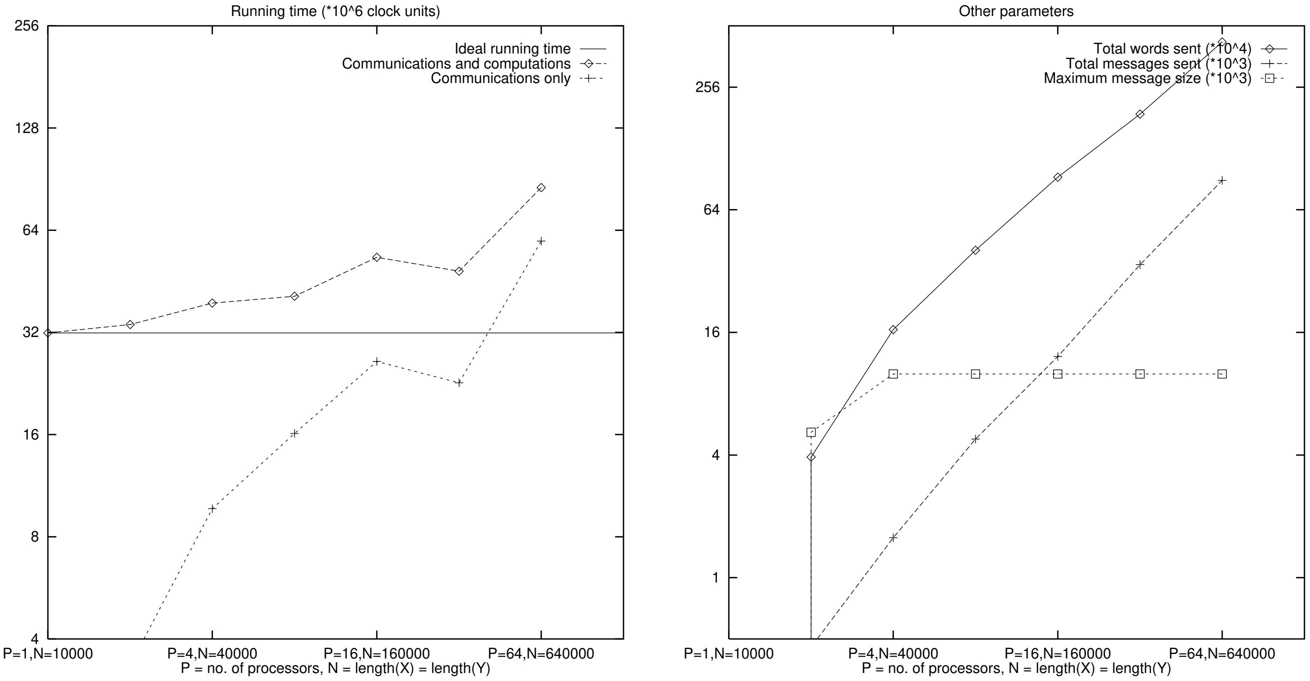


Figure 21: Scaleup of *merge*$(x, y)$.

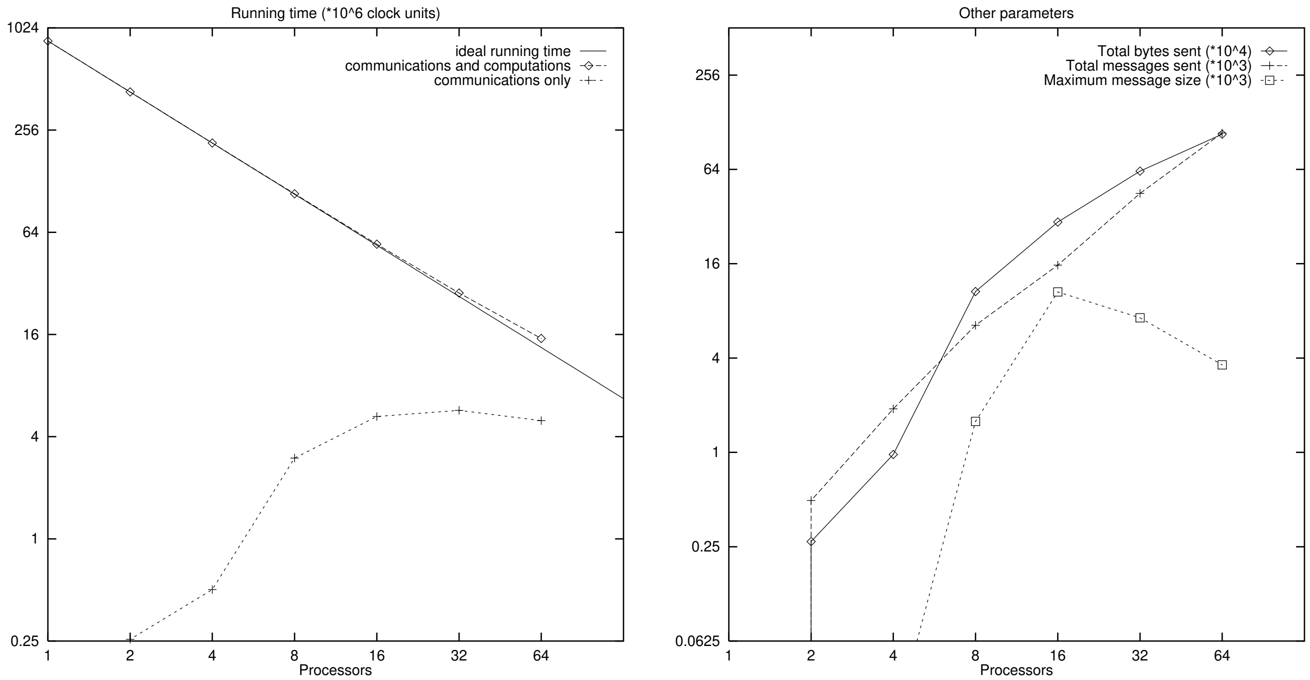


Figure 22: Speedup of the OO benchmark.

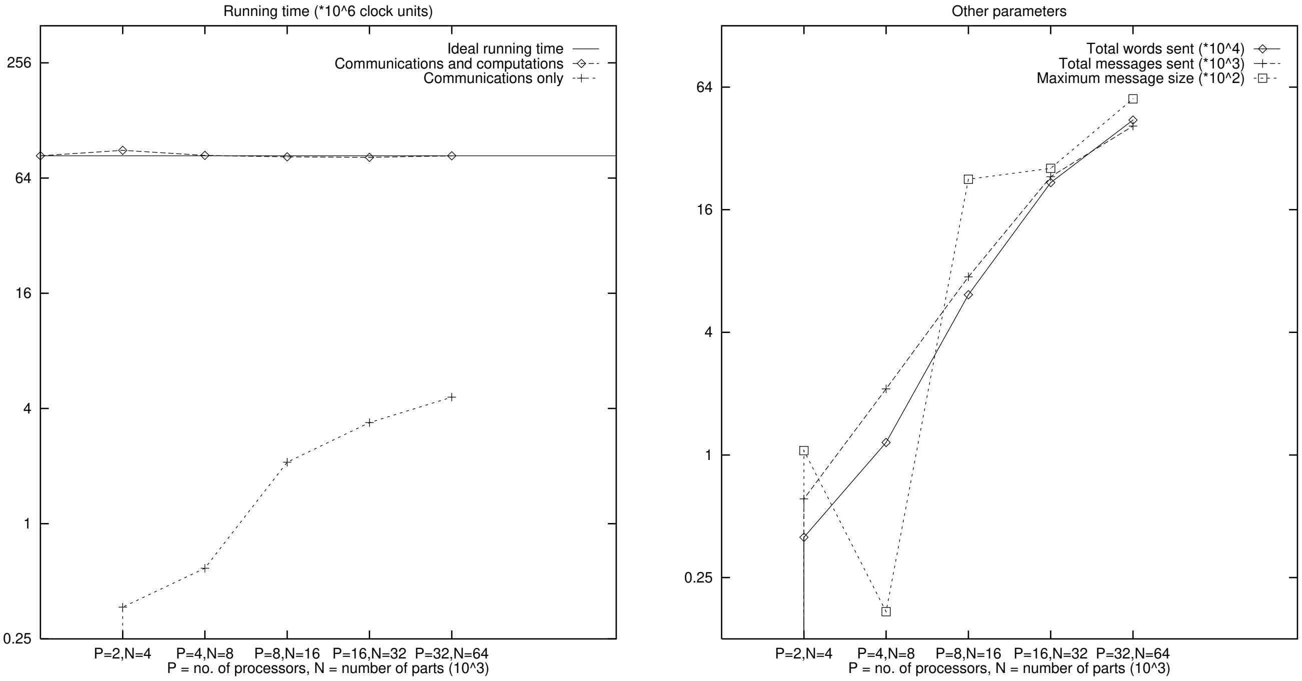


Figure 23: Scaleup of the OO benchmark.

good cost model for the BVRAM operations is less obvious: here we hope that the "bounded" nature of most of BVRAM's operators will be of help.

**Complexities of Set Operations** CoPa has no special join operation, but can express them as nested loops. Written like this, the complexity of a join is $T = O(1), W = O(n^2)$ and this makes it an easy problem for a complexity-preserving compilation. The same phenomenon occurs for other operations frequent in set-oriented query languages (like nesting). A better way would be to have additional set primitives in the language, with lower complexities, and require a compiler to preserve *those* complexities.

**Object IDs and Object Encapsulation** There are two components to objects, in an object-oriented or object-relational database: the encapsulation of methods, and the ability to refer to an object (by using the Object ID like a pointer). Regarding methods, CoPa addresses them to some extent in the form of external functions. Methods can add more complexity in that the set of methods in a database may vary dynamically. Object IDs allow *object sharing* and *object updates*. Neither the BVRAM model nor our LogP implementation can currently deal with these features.

**Implementation of Database Primitives on the LogP Model** Our results on optimal implementation of monotone communications seems to be the first discussing database operations in the context of the LogP model. But monotone communications are specific to sequences (the definition of *monotone* is based on order): traditional database operations like join, nest, etc, deal specifically with unordered collections, and hence would require a new approach. A study of database operations in the context of the LogP model could help a lot understanding the communication costs in parallel database systems.

# References


[1] A.Borodin and J.Hopcroft. Routing, merging, and sorting on parallel models of computation. *JCSS*, 30(1):130–145, 1985.

[2] Albert Alexandrov, Mihai Ionescu, Klaus Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, 1995.

[3] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

[4] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In Francois Bancilhon, Claude Delobel, and Paris Kanellakis, editors, *Building and object-oriented database system. The story of $O_2$*. Morgan Kaufmann Publishers, 1992.

[5] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proceedings of 13th International Conference on Very Large Data Bases*, pages 97–105, 1987.

[6] Amotz Bar-Noy and Shlomo Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, 1992.

[7] Guy E Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, Massachusetts, 1990.

[8] Guy E Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, Pittsburgh, PA 15213, 1993.

[9] Guy E Blelloch. The high cost of communication: hardware vs. software development. Invited talk at the workshop on Suggesting Computer Science Agendas for High-Performance Computing, March 1994.

[10] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proc. of the 3rd Symposium on the Forntiers of Massively Parallel Computation*, pages 471–480, University of Maryland, October 1990.

[11] Guy E. Blelloch and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–112, San Diego, May 1993.

[12] Guy E Blelloch and Gary Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[13] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *JCSS*, 7(4):448–461, 1973.

[14] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 9–19. Morgan Kaufmann, August 1991. Also available as UPenn Technical Report MS-CIS-92-17.

[15] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

[16] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[17] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of proramming with collection types. *Theoretical Computer Science*, 149:3–48, 1995.

[18] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, Minneapolis, MN, May 1994.

[19] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the ACM SIGMOD conference*, Washington, D.C., May 1993.

[20] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, California, 1996.

[21] Marina Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 131–139, 1986.

[22] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 1–12, San Diego, California, May 1993.

[23] P. L. Curien. The $\lambda\rho$-calculus: An abstract framework for environment machines. Technical Report URA 725, Laboratoire d'Informatique, Departement de Mathematiques et d'Informatique, Ecole Normale Superieure, 45 Rue d'Ulm, 75230 Paris Cedex 05, France, 1988.

[24] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Experiences using control dependence in PTRAN. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 186–212. The MIT Press, Cambridge, Massachusetts, 1990.

[25] O. Deux. The story of $O_2$. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[26] D. DeWitt, J. Naughton, and J. Burger. GAMMA - a high performance dataflow database machine. In *Proceedings of the Twelfth International Conference on Very large Data Bases*, pages 228–237, 1986.

[27] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[28] Digital Equipment Corporation. *ISO-ANSI Working Draft — Database Language SQL (SQL3)*, August 1993.

[29] Leonidas Fegaras and David Maier. Towards an effective calculus of object query languages. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 47–58, June 1995.

[30] John T. Feo. Arrays in Sisal. In Lenore Mullin, Michael Jenkins, Gaetan Hains, Robert Bernecky, and Guang Goa, editors, *Arrays, Functional Languages, and Parallel Systems*, pages 93–106. Kluwer, Boston, 1991.

[31] John T. Feo and David Cann. A report on the Sisal langauge project. *Journal of Parallel and Distributed Computing*, pages 349–365, 1990.

[32] Christian Foisy, Julie Vachon, and Gaetan Hains. DBML: de la semantique a l'implantation. *Journees Francophones des Langauges Applicatifs*, February 1994.

[33] Shahram Ghandeharizadeh, David Wilhite, Kaiming Lin, and Xiaoming Zhao. Object placement in parallel object-oriented database systems. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 198–209, February 1993.

[34] A. Goldberg, P. Mills, L. Nyland, J. Prins, J. Reif, and J. Riely. Specification and development of parallel algorithms with the proteus system,. manuscript available by XMosaic from http://www.cs.unc.edu/proteus-publications.html, 1994.

[35] Stephane Grumbach and Tova Milo. Towards tractable algebras for bags. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 49–58, Washington, D. C., May 1993.

[36] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[37] Gaetan Hains. Programmation fonctionelle et parallelisme: une approche pragmatique. manuscript, August 1993.

[38] Gaetan Hains and Christian Foisy. The data-parallel categorical abstract machine. In *PARLE'93: Conference on Parallel Architectures and Languages Europe*. Springer Verlag, 1993.

[39] Daniel Hillis and Guy Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[40] Joseph Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[41] Gilles Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987.

[42] Richard Karp, Abhijit Sahay, Eunice Santos, and Klaus Erik Schauser. Optimal broadcast and summation in the logp model. In *Proceedings of 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 142–153, 1993.

[43] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance FORTRAN Handbook*. The MIT Press, Cambridge, Massachusetts, 1991.

[44] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), October 1991.

[45] J. Larus. $c^{**}$: a large-grain, object-oriented, data-parallel programming language. In *Proc 5th Int'l Wkshp on Languages and Compilers for Parallel Computing*, pages 326–341, 1993.

[46] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures : Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1992.

[47] Leonid Libkin and Limsoon Wong. Conservativity of nested relational calculi with internal generic functions. *Information Processing Letters*, 49(6):273–280, March 1994. See also UPenn Technical Report MS-CIS-93-60.

[48] Leonid Libkin and Limsoon Wong. Some properties of query languages for bags. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 97–114. Springer-Verlag, January 1994. See also UPenn Technical Report MS-CIS-93-36.

[49] Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *JCSS*, 55(2):241–272, 1997.

[50] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, Berlin, 1971.

[51] S. Merrall. Tpl–an imp[[lemnetation of paralation lisp in eulisp. Technical report, School of Math Sciences, Univ of Bath, 1991.

[52] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[53] C. Mohan, H. Pirahesh, W. Tang, and Y. Wang. Parallelism in relational database managemant systems. *IBM Systems Journal*, 33(2):349–371, 1994.

[54] Peter H. Mulls, Lars S. Nyland, Jan F. Prins, John H. Reif, and Robert A. Wagner. Prototyping parallel and distributed programs in *Proteus*. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 26–34, December 1991.

[55] Jan Paredaens and Dirk Van Gucht. Converting nested relational algebra expressions into flat algebra expressions. *ACM Transaction on Database Systems*, 17(1):65–93, March 1992.

[56] D. Stott Parker, Eric Simon, and Patrick Valduriez. SVP: A model capturing sets, streams, and parallelism. In Li-Yan Yuan, editor, *Proceedings of 18th International Conference on Very Large Databases, Vancouver, August 1992*, pages 115–126, San Mateo, California, August 1992. Morgan-Kaufmann.

[57] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[58] Franco Preparata. New parallel-sorting schemas. *IEEE Transactions on Computers*, C-27(7):669–673, July 1978.

[59] Jan Prins and Daniel Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA., May 19-22, 1993. ACM Press.

[60] G. Sabot. *The paralation model: architecture-independent SIMD programming*. MIT Press, 1988.

[61] Donovan Schneider and David DeWitt. A performance evaluation of four parallel join algorithms in a shared nothing multiprocessor environment. In *Proceedings of the 1989 SIGMOD Conference*, Portland, 1989.

[62] Stephen Skedzielewski. Sisal. In Loleslaw Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–157. Addison-Wesley Publishing Co, 1991.

[63] Michael Stonebaker and Dorothy Moore. *Object-Relational DBMSs — The Next Wave*. Morgan Kaufmann, San Francisco, CA, 1996.

[64] Dan Suciu. *Parallel programming languages for collections.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1995. Available as University of Pennsylvania IRCS Report 95-18.

[65] Dan Suciu and Val Breazu-Tannen. A query language for NC. In *Proceedings of 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 167–178, Minneapolis, Minnesota, May 1994.

[66] Dan Suciu and Val Tannen. Efficient compilation of high-level data parallel algorithms. In *Proceedings of 6th ACM SIGACT-SIGMOD-SIGART Symposium on Parallel Algorithms and Architectures*, pages 57–66, June 1994.

[67] Dan Suciu and Val Tannen. A query language for NC. *Journal of Computer and System Sciences*, 55(2):299–321, October 1997.

[68] Dan Suciu and Limsoon Wong. On two forms of structural recursion. In Georg Gottlob and Moshe Y. Vardi, editors, *Proceedings of the Fifth International Conference on Database Theory*, number 893 in Lecture Notes in Computer Science, pages 111–124. Springer Verlag, January 1995.

[69] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Programming Guide*, May 1993.

[70] P. W. Trinder. Comprehensions, a query notation for DBPLs. In *Proceedings of 3rd International Workshop on Database Programming Languages, Nahplion, Greece*, pages 49–62. Morgan Kaufmann, August 1991.

[71] P. W. Trinder and P. L. Wadler. List comprehensions and the relational calculus. In *Proceedings of 1988 Glasgow Workshop on Functional Programming*, pages 115–123, Rothesay, Scotland, August 1988.

[72] L.W. Tucker and A. Mainwaring. CMMD: Active messages on the CM-5. *Parallel Computing*, 20(4), April 1994.

[73] Patrick Valduriez and Setrag Khoshafian. Parallel evaluation of the transitive closure of a database relation. *International Journal of Parallel Programming*, 17(1):19–42, 1988.

[74] Leslie G. Valiant. Parallelism in comparison problems. *SIAM Journal of Computing*, 4(3):348–355, 1975.

[75] Philip Wadler. Comprehending monads. In *Proceedings of ACM Conference on Lisp and Functional Programming*, Nice, June 1990.

[76] David A. Watt and Phil Trinder. Towards a theory of bulk types. Fide Technical Report 91/26, Glasgow University, Glasgow G12 8QQ, Scotland, July 1991.

[77] Limsoon Wong. *Querying Nested Collections.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.

[78] Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *JCSS*, 52(3):495–505, 1996.

| Syntax | Operational Semantics | Complexities |
|---|---|---|
| $\dfrac{}{x : t, \Gamma \rhd x : t}$ | $\dfrac{}{x = C, \rho \bullet x \Downarrow C}$ | $T((x = C, \rho), x) \stackrel{\mathrm{def}}{=} 1$ <br> $W((x = C, \rho), x) \stackrel{\mathrm{def}}{=} \mathit{size}(C) + \mathit{size}(\rho)$ |
| $\dfrac{}{\Gamma \rhd c : t_c} \ (c \in \Sigma)$ | $\dfrac{}{\rho \bullet c \Downarrow c}$ | $T(\rho, c) \stackrel{\mathrm{def}}{=} 1$ <br> $W(\rho, c) \stackrel{\mathrm{def}}{=} 1 + \mathit{size}(\rho)$ |
| $\dfrac{}{\Gamma \rhd p : d_p \to c_p} \ (p \in \Sigma)$ | $\dfrac{f(C) = C'}{\rho \bullet f(C) \Downarrow C'} \ (p \in \Sigma)$ | $T(\rho, p, C) \stackrel{\mathrm{def}}{=} 1 + T(p, C)$ <br> $W(\rho, p, C) \stackrel{\mathrm{def}}{=} T(p, C) + \mathit{size}(\rho)$ |

Figure 24: Variables, constants, and external functions

# A CoPa's Formal Definition with Complexities

Figures 24, to 30 contain the definitions of CoPa's typing rules, operational semantics, and complexities. For each external function $p : d_p \to c_p$ we assume to have a function $T(p, C)$ given which returns the time needed to compute $f(C)$ (sequential time). For each $p$ we assume $T(p, C) \geq \mathit{size}(C)$, and to be computable in constant amount of time.

# B Proof of Theorem 4.11

Assume w.l.o.g. that the BVRAM $M$ has only two kinds of vector registers: those of type [D] and those of type [N]. For each step $t$, $t = 0, T$, let $v^t$ be the sequence of type [D] obtained by concatenating all values of the [D] vector registers at time $t$. Thus we get a sequence of $T$ sequences $v^0, v^1, v^2, \ldots, v^T$: we call this the **D-trace** of the BVRAM $M$. If $p : [\mathtt{D}] \to [\mathtt{D}]$ is the permutation under consideration, we may assume w.l.o.g. that $v^0 = x$ and $v^T = p(x)$: the first condition holds since all non-input registers are initialized with [], while the second condition can be enforced by extending $M$ with assignments $V_i \leftarrow []$ for all registers $V_i$ except the output register. We introduce now a definition:

**Definition B.1** *A* **potential trace** *of length $T$, with input $a : [\mathtt{D}]$, output $b : [\mathtt{D}]$, and constant $c \geq 1$ is a sequence $x^0, x^1, x^2, \ldots, x^T$, where $x^t \in [\mathtt{D}]$, $t = 1, T$, with the following properties:*

1. *$a = x^0, b = x^T$.*
2. *$x^{t+1}$ is obtained from $x^t$ in one of the following two ways:*
   - *(a) by deleting some of its elements.*
   - *(b) by the following* sbmRoute*-like operation. There exists a number $k$, $k$ sequences $x_1, \ldots, x_k$, and $k$ numbers $m_1, \ldots, m_k$ such that:*

$$\begin{aligned} x^t &= x_1 @ x_2 @ \ldots @ x_k \\ x^{t+1} &= \underbrace{x_1 @ \ldots @ x_1}_{m_1 \ \mathit{times}} @ \ldots @ \underbrace{x_k @ \ldots @ x_k}_{m_k \ \mathit{times}} \end{aligned}$$

| Syntax | Operational Semantics | Complexities |
|---|---|---|
| $\dfrac{\Gamma \rhd e_1 : t_1, \ldots, \Gamma \rhd e_n : t_n}{\Gamma \rhd \langle e_1, \ldots, e_n \rangle : t_1 \times \ldots \times t_n}$ | $\dfrac{\rho \bullet e_1 \Downarrow C_1, \ldots, \rho \bullet e_n \Downarrow C_n}{\rho \bullet \langle e_1, \ldots, e_n \rangle \Downarrow \langle C_1, \ldots, C_n \rangle}$ | $T(\rho, \langle e_1, \ldots, e_n \rangle) \overset{\text{def}}{=} 1 + \sum_{i=1,n} T(\rho, e_i)$<br>$W(\rho, \langle e_1, \ldots, e_n \rangle) \overset{\text{def}}{=} \sum_{i=1,n} W(\rho, e_i) + size\langle C_1, \ldots, C_n \rangle + size(\rho)$ |
| $\dfrac{\Gamma \rhd e : t_1 \times \ldots \times t_n, i = 1, n}{\Gamma \rhd \pi_i^{t_1 \times \ldots \times t_n}(e) : t_i}$ | $\dfrac{\rho \bullet e \Downarrow \langle C_1, \ldots, C_n \rangle}{\pi_i(e) \Downarrow C_i}$ | $T(\rho, \pi_i(e)) \overset{\text{def}}{=} 1 + T(\rho, e)$<br>$W(\rho, \pi_i(e)) \overset{\text{def}}{=} W(\rho, e) + size(C_i) + size(\rho)$ |
| $\dfrac{\Gamma \rhd e : t_i, i = 1, n}{\Gamma \rhd in_i^{t_1 + \ldots + t_n}(e) : t_1 + \ldots + t_n}$ | $\dfrac{\rho \bullet e \Downarrow C}{\rho \bullet in_i(e) \Downarrow in_i(C)}$ | $T(\rho, in_i(e)) \overset{\text{def}}{=} 1 + T(\rho, e)$<br>$W(\rho, in_i(e)) \overset{\text{def}}{=} W(\rho, e) + size(in_i(C)) + size(\rho)$ |
| $\dfrac{\begin{array}{c}\Gamma \rhd e : t_1 + \ldots + t_n \\ x_1 : t_1, \Gamma \rhd e_1 : t \\ \ldots \\ x_n : t_n, \Gamma \rhd e_n : t\end{array}}{\Gamma \rhd \left(\begin{array}{ccc}\mathsf{case} & e & \mathsf{of} \\ in_1(x) & \Rightarrow & e_1 \mid \\ & \ldots & \\ in_n(x_n) & \Rightarrow & e_n\end{array}\right) : t}$ | $\dfrac{\rho \bullet e \Downarrow in_i(C) \qquad x_i = C, \rho \bullet e_i \Downarrow C'}{\rho \bullet \left(\begin{array}{ccc}\mathsf{case} & e & \mathsf{of} \\ in_1(x) & \Rightarrow & e_1 \mid \\ & \ldots & \\ in_n(x_n) & \Rightarrow & e_n\end{array}\right) \Downarrow C'}$ | $T(\rho, \mathsf{case} \ldots) \overset{\text{def}}{=} 1 + T(\rho, e) + T((x_i = C, \rho), e_i)$<br>$W(\rho, \mathsf{case} \ldots) \overset{\text{def}}{=} W(\rho, e) + W((x_i = C, \rho), e_i) + size(C_i) + size(\rho)$ |

Figure 25: Product types and sum types

| Syntax | Operational Semantics | Complexities |
|---|---|---|
| $\dfrac{\Gamma \rhd f : t_1 \to t_2 \qquad \Gamma \rhd e : t_1}{\Gamma \rhd f(e) : t_2}$ | $\dfrac{\rho \bullet e \Downarrow C_1 \qquad \rho \bullet f(C_1) \Downarrow C_2}{\rho \bullet f(e) \Downarrow C_2}$ | $T(\rho, f(e)) \overset{\text{def}}{=} 1 + T(\rho, e) + T(\rho, f, C_1)$<br>$W(\rho, f(e)) \overset{\text{def}}{=} W(\rho, e) + W(\rho, f, C_1) + size(C_2) + size(\rho)$ |
| $\dfrac{x : t_1, \Gamma \rhd e : t_2}{\Gamma \rhd \lambda x{:}t_1.e : t_1 \to t_2}$ | $\dfrac{x = C_1, \rho \bullet e \Downarrow C_2}{\rho \bullet (\lambda x.e)(C_1) \Downarrow C_2}$ | $T(\rho, \lambda x.e, C_1) \overset{\text{def}}{=} 1 + T((x = C_1, \rho), e)$<br>$W(\rho, \lambda x.e, C_1) \overset{\text{def}}{=} W((x = C_1, \rho), e) + size(C_2) + size(\rho)$ |

Figure 26: $\lambda$-abstraction and function application

| Syntax | Operational Semantics | Complexities |
|---|---|---|
| $\dfrac{\Gamma \rhd p : t \to \mathsf{B} \qquad \Gamma \rhd f : t \to t}{\Gamma \rhd \mathsf{while}(p, f) : t \to t}$ | $\dfrac{\rho \bullet p(C) \Downarrow \mathsf{false}}{\rho \bullet \mathsf{while}(p, f)(C) \Downarrow C}$ | $T(\rho, \mathsf{while} \ldots, C) \stackrel{\text{def}}{=} 1 + T(\rho, p, C)$<br>$W(\rho, \mathsf{while} \ldots, C) \stackrel{\text{def}}{=} W(\rho, p, C) + size(C) + size(\rho)$ |
| | $\dfrac{\begin{array}{rcl} \rho \bullet p(C) & \Downarrow & \mathsf{true} \\ \rho \bullet f(C) & \Downarrow & C_1 \\ \rho \bullet \mathsf{while}(p, f)(C_1) & \Downarrow & C_2 \end{array}}{\rho \bullet \mathsf{while}(p, f)(C) \Downarrow C_2}$ | $T(\rho, \mathsf{while} \ldots, C) \stackrel{\text{def}}{=} 1 + T(\rho, p, C) + T(\rho, f, C) + T(\rho, \mathsf{while} \ldots, C_1)$<br>$W(\rho, \mathsf{while} \ldots, C) \stackrel{\text{def}}{=} W(\rho, p, C) + W(\rho, f, C) + W(\rho, \mathsf{while} \ldots, C_1) + size(\rho)$<br>Note: $size(C_2)$ does not occur here. |
| $\dfrac{d : t \to [t] \qquad c : t \times [t'] \to t'}{\mathsf{maprec}(d, c) : t \to t'}$ | $\dfrac{\begin{array}{rcl} \rho \bullet d(C) & \Downarrow & [C_0, \\ & & \vdots \\ & & C_{n-1}] \\ \rho \bullet \mathsf{maprec}(d, c)(C_0) & \Downarrow & C'_0 \\ \ldots & & \\ \rho \bullet \mathsf{maprec}(d, c)(C_{n-1}) & \Downarrow & C'_{n-1} \\ \rho \bullet c\langle C, [C'_0, \ldots, C'_{n-1}]\rangle & \Downarrow & C' \end{array}}{\mathsf{maprec}(d, c)(C) \Downarrow C'}$ | $T(\rho, \mathsf{maprec}(d, c), C) \stackrel{\text{def}}{=} 1 + T(\rho, d, C) + \max_i (T(\rho, \mathsf{maprec}(d, c), C_i)) + T(\rho, c, [C'_0, \ldots, C'_{n-1}])$<br>$W(\rho, \mathsf{maprec}(d, c), C) \stackrel{\text{def}}{=} W(\rho, d, C) + \sum_i W(\rho, \mathsf{maprec}(d, c), C_i) + W(\rho, c, [C'_0, \ldots, C'_{n-1}]) + size([C'_0, \ldots, C'_{n-1}]) + size(\rho)$ |

Figure 27: Iteration and map-recursion

| Syntax | Operational Semantics | Complexities |
|---|---|---|
| $\dfrac{}{\Gamma \rhd [\,] : [t]}$ | $\dfrac{}{\rho \bullet [\,] \Downarrow [\,]}$ | $T(\rho, [\,]) \stackrel{\text{def}}{=} 1$ <br> $W(\rho, [\,]) \stackrel{\text{def}}{=} 1 + size(\rho)$ |
| $\dfrac{\Gamma \rhd e : t}{\Gamma \rhd [e] : [t]}$ | $\dfrac{\rho \bullet e \Downarrow C}{\rho \bullet [e] \Downarrow [C]}$ | $T(\rho, [e]) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ <br> $W(\rho, [e]) \stackrel{\text{def}}{=} W(\rho, e) + size(C) + size(\rho)$ |
| $\dfrac{\Gamma \rhd e_1 : [t] \qquad \Gamma \rhd e_2 : [t]}{\Gamma \rhd e_1 @ e_2 : [t]}$ | $\dfrac{\begin{array}{l} \rho \bullet e \Downarrow [C_0, \ldots C_{m-1}] \\ \rho \bullet e' \Downarrow [C'_0, \ldots, C'_{n-1}] \end{array}}{\rho \bullet e @ e' \Downarrow [C_0, \ldots, C_{m-1}, C'_0, \ldots, C'_{n-1}]}$ | $T(\rho, e @ e') \stackrel{\text{def}}{=} 1 + T(\rho, e) + T(\rho, e')$ <br> $W(\rho, e @ e') \stackrel{\text{def}}{=} W(\rho, e) + W(\rho, e') + size([C_0, \ldots, C'_{n-1}]) + size(\rho)$ |
| $\dfrac{\Gamma \rhd e : [[t]]}{\Gamma \rhd \mathsf{flatten}(e) : [t]}$ | $\dfrac{\rho \bullet e \Downarrow [[C_{00}, C_{01}, \ldots], [C_{10}, C_{11}, \ldots], \ldots]}{\rho \bullet \mathsf{flatten}(e) \Downarrow [C_{00}, C_{01}, \ldots, C_{10}, C_{11}, \ldots]}$ | $T(\rho, \mathsf{flatten}(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ <br> $W(\rho, \mathsf{flatten}(e)) \stackrel{\text{def}}{=} W(\rho, e) + size([C_{00}, \ldots]) + size(\rho)$ |
| $\dfrac{\Gamma \rhd e : [t]}{\Gamma \rhd \mathsf{length}(e) : \mathbb{N}}$ | $\dfrac{\rho \bullet e \Downarrow [C_0, \ldots, C_{n-1}]}{\rho \bullet \mathsf{length}(e) \Downarrow n}$ | $T(\rho, \mathsf{length}(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ <br> $W(\rho, \mathsf{length}(e)) \stackrel{\text{def}}{=} W(\rho, e) + 1 + size(\rho)$ |
| $\dfrac{\Gamma \rhd e : [t]}{\Gamma \rhd \mathsf{get}(e) : t}$ | $\dfrac{\rho \bullet e \Downarrow [C]}{\rho \bullet \mathsf{get}(e) \Downarrow C}$ | $T(\rho, \mathsf{get}(e)) \stackrel{\text{def}}{=} 1 + T(\rho, e)$ <br> $W(\rho, \mathsf{get}(e)) \stackrel{\text{def}}{=} W(\rho, e) + size(C) + size(\rho)$ |
| $\dfrac{\Gamma \rhd f : t_1 \rightarrow t_2}{\Gamma \rhd \mathsf{map}(f) : [t_1] \rightarrow [t_2]}$ | $\dfrac{\rho \bullet f(C_0) \Downarrow C'_0 \quad \ldots \quad \rho \bullet f(C_{n-1}) \Downarrow C'_{n-1}}{\rho \bullet \mathsf{map}(f)([C_0, \ldots, C_{n-1}]) \Downarrow [C'_0, \ldots, C'_{n-1}]}$ | $T(\rho, \mathsf{map}(f), \ldots) \stackrel{\text{def}}{=} 1 + \max_{i=0,n-1} (T(\rho, f, C_i))$ <br> $W(\rho, \mathsf{map}(f), \ldots) \stackrel{\text{def}}{=} \sum_{i=0,n-1} W(\rho, f, C_i) + size([C'_0, \ldots, C'_{n-1}]) + size(\rho)$ |

Figure 28: Operational semantics for collection operations. Only sequences are shown: sets, bags, and conversions are similar.

| Syntax | Operational Semantics | Complexities |
|---|---|---|
| $\overline{\mathsf{zip} : [s] \times [t] \to [s \times t]}$ | $\dfrac{\rho \bullet e \Downarrow [C_0, \ldots, C_{n-1}] \quad \rho \bullet e' \Downarrow [C'_0, \ldots, C'_{n-1}]}{\rho \bullet \mathsf{zip}\langle e, e' \rangle \Downarrow [\langle C_0, C'_0 \rangle, \ldots, \langle C_{n-1}, C'_{n-1} \rangle]}$ | $T(\rho, \mathsf{zip}\langle e, e' \rangle) \stackrel{\mathrm{def}}{=} 1 + T(\rho, e) + T(\rho, e')$ <br> $W(\rho, \mathsf{zip}\langle e, e' \rangle) \stackrel{\mathrm{def}}{=} W(\rho, e) + W(\rho, e') + \mathit{size}([\langle C_0, C'_0 \rangle, \ldots]) + \mathit{size}(\rho)$ |
| $\overline{\mathsf{enumerate} : [t] \to \mathbb{N}}$ | $\dfrac{\rho \bullet e \Downarrow [C_0, \ldots, C_{n-1}]}{\rho \bullet \mathsf{enumerate}(e) \Downarrow [0, \ldots, n-1]}$ | $T(\rho, \mathsf{enumerate}(e)) \stackrel{\mathrm{def}}{=} 1 + T(\rho, e)$ <br> $W(\rho, \mathsf{enumerate}(e)) \stackrel{\mathrm{def}}{=} W(\rho, e) + n + 1 + \mathit{size}(\rho)$ |
| $\overline{\mathsf{split} : [t] \times [\mathbb{N}] \to [[t]]}$ | $\dfrac{\rho \bullet e_1 \Downarrow [C_0, \ldots C_{n_0+\ldots+n_{m-1}-1}] \quad \rho \bullet e_2 \Downarrow [n_0, \ldots, n_{m-1}]}{\rho \bullet \mathsf{split}\langle e_1, e_2 \rangle \Downarrow [[C_0, \ldots, C_{n_0-1}], [C_{n_0}, \ldots, C_{n_0+n_1-1}], \ldots [C_{n_0+\ldots+n_{m-2}}, \ldots, C_{n_0+\ldots+n_{m-1}-1}]]}$ | $T(\rho, \mathsf{split}(\ldots)) \stackrel{\mathrm{def}}{=} 1 + T(\rho, e_1) + T(\rho, e_2)$ <br> $W(\rho, \mathsf{split}(\ldots)) \stackrel{\mathrm{def}}{=} W(\rho, e_1) + W(\rho, e_2) + \mathit{size}([[C_0, \ldots], \ldots]) + \mathit{size}(\rho)$ |

Figure 29: Operations on sequences

| Syntax | Operational Semantics | Complexities |
|---|---|---|
| $\dfrac{\Gamma \rhd e : t}{x : t', \Gamma \rhd e : t_2}$ | $\dfrac{\rho \bullet e \Downarrow C}{X = C', \rho \bullet e \Downarrow C}$ | $T((x = C', \rho), e) \stackrel{\mathrm{def}}{=} 1 + T(\rho, e)$ <br> $W((x = C', \rho), e) \stackrel{\mathrm{def}}{=} W(\rho, e) + \mathit{size}(C) + \mathit{size}(x = C', \rho)$ |
| $\dfrac{\Gamma \rhd f : t_1 \to t_2}{x : t', \Gamma \rhd f : t_1 \to t_2}$ | $\dfrac{\rho \bullet f(C_1) \Downarrow C_2}{X = C', \rho \bullet f(C_1) \Downarrow C_2}$ | $T, W$ same as above |

Figure 30: The weakening rules

*In particular $x^t$ can be equal to $x^{t+1}$.*

3. *For each $t = 0, T$, $\mathsf{length}(x^t) \le c\,\mathsf{length}(a)$.*

Next we prove that the D-trace can be transformed into a potential trace by at most doubling its length.

**Lemma B.2** *Let $v^0, v^1, \ldots, v^T$ be the* D*-trace of the BVRAM $M$ with linear work complexity $W = O(n)$. Then there exists a potential trace $x^0, \ldots, x^{2T}$ with input $v^0$, output $v^T$, and some constant $c$, such that $v^t = x^{2t}$, for $t = 0, T$.*

**Proof.** The Lemma is easily proven by induction on $t$. Here it is essential to note that the set of BVRAM instructions which can affect the vector registers of type [D] is restricted, due to the uniformity condition. It can be only one of: $[], \mathsf{append}, \mathsf{select}, \mathsf{bmRoute}, \mathsf{sbmRoute}$. We illustrate some of the cases:

- $V_i \leftarrow []$. This translates in "deleting" the elements from $v^t$ to get $v^{t+1}$.
- $V_i \leftarrow V_j @ V_k$. This is the more involved case. For illustration assume that in $v^t$, and therefore in $x^{2t}$, the registers $V_i, V_j, V_k$ appear as follows:

  $$v^t = x^{2t} = \ldots @ V_k @ \ldots @ V_i @ \ldots @ V_j @ \ldots$$

  Here we first do a "sbmRoute-like" operation and define:

  $$x^{2t+1} = \ldots @ \underbrace{(V_k @ \ldots @ V_i @ \ldots @ V_j) @ \ldots @ (V_k @ \ldots @ V_i @ \ldots @ V_j)}_{4 \text{ times}} @ \ldots$$

  The left and right ends in $x^{2t+1}$ are the same as those in $x^{2t}$. Now it suffices to delete elements from the 4 groups and to leave only $(V_k \ldots)$, $V_j$, $V_k$, and $(\ldots V_j)$ respectively, which gives us:

  $$x^{2t+2} = v^{t+1} = \ldots @ (V_k @ \ldots) @ (V_j @ V_k) @ (\ldots @ V_j) @ \ldots$$

  Note that the size of $x^{2t+1}$ is at most 4 times larger than $size(x^{2t})$.
- $V_i \leftarrow \mathsf{select}(V_j, V_k)$. Here $V_i$ and $V_k$ are of type [D] while $V_j$ is of type [N]. This is essentially a delete operation, but preceded by a "sbmRoute-like", as in the case of append.
- bmRoute and sbmRoute are treated similarly.

Since $W = O(n)$ it follows that $size(v^t) \le c'n$, for some constant $c' \ge 0$. It suffices than to choose $c = 4c'$. $\square$

Let $[a_0, \ldots, a_{n-1}]$ be some sequence of type [D], and suppose we have a total order $<$ defined on the set $\{a_0, \ldots, a_{n-1}\}$. We define an **ascending subchain of length** $L$ to be a sequence $a_{i_0}, \ldots, a_{i_{L-1}}$ s.t. $0 \le i_0 < i_1 < \ldots < i_{L-1} < n$ and $a_{i_0} < a_{i_1} < \ldots < a_{i_{L-1}}$. E.g. for the order relation $\alpha < \beta < \gamma$, there are three ascending subchains of length 3 of the sequence $x = [a_0, a_1, a_2, a_3, a_4, a_5, a_6] = [\beta, \gamma, \alpha, \beta, \alpha, \beta, \gamma]$; namely: $(a_2, a_3, a_6)$, $(a_2, a_5, a_6)$ and $(a_4, a_5, a_6)$. All three are $\alpha < \beta < \gamma$.

The key lemma is given below.

**Lemma B.3** *Let $[a_0, \ldots, a_{n-1}]$ be the output of a potential trace of length $T$ and constant $c \geq 1$. Assume the elements $a_0, \ldots, a_{n-1}$ are distinct and that the set $\{a_0, \ldots, a_{n-1}\}$ ordered by $a_0 < \ldots < a_{n-1}$. Then there exists an ascending sub-chain of length $\geq \frac{1}{c^{2^T-1}} n$ in the input $x^0$ of the potential trace.*

**Proof.** For $T = 0$ we have $\frac{1}{c^{2^T-1}} n = n$ and the input equals the output: the whole sequence $a_0, \ldots, a_{n-1}$ is an ascending subchain of length $\geq n$. For $T > 0$, by induction hypothesis, there is a sub-chain of length $\geq \frac{1}{c^{2^{T-1}-1}} n = dn$ in $x^1$. We consider how $x^1$ has been derived from $x^0$:

**Case 1** $x^1$ is obtained by deleting elements from $x^0$. Then the same ascending subchain of $x^1$ is also a subchain of $x^0$, and it is certainly $\geq \frac{1}{c^{2^T-1}} n$.

**Case 2** There exists: a number $k$, $k$ sequences $x_1, \ldots, x_k$, and $k$ numbers $m_1, \ldots, m_k$ such that:

$$\begin{array}{rcl} x^0 & = & x_1 @ x_2 @ \ldots @ x_k \\ x^1 & = & \underbrace{x_1 @ \ldots @ x_1}_{m_1 \text{ times}} @ \ldots @ \underbrace{x_k @ \ldots @ x_k}_{m_k \text{ times}} \end{array}$$

Let $y$ be the ascending subchain of $x^1$ of length $\geq dn$. Denote with $y_i$ the subsequence which lies inside $x_i @ \ldots @ x_i$, $i = 1, k$. Let $n_i = \mathsf{length}(y_i)$, hence $\mathsf{length}(y) = \sum_{i=1,k} n_i \geq dn$.

Furthermore, let $n_{ij}$ be the length in the of the subsequence of $y_i$ falling into the $j$'s copy of $x_i$, $j = 1, m_i$. That is $n_i = \sum_{j=1,m_i} n_{ij}$. Let $p_i = \max_{j=1,m_i}(n_{ij})$; for each $i$ we will pick the corresponding maximal subsequence $y_{ij}$ of $y_i$ having length $p_i$; by concatenating all these $k$ sequences we get a sub-chain of $x^0$ of length $\sum p_i$. Now we prove that this length is "large enough". We have the following:

$$\sum_{i=1,k} n_i \geq dn \text{ (see above)} \tag{5}$$

$$\sum_{i=1,k} m_i n_i \leq cn \text{ (because } n_i \leq \mathsf{length}(x_i) \text{ and } \mathsf{length}(x^1) \leq cn) \tag{6}$$

$$m_i p_i \geq n_i \text{ (because } \textstyle\sum_j n_{ij} = n_i \text{ and } p_i \text{ is max)} \tag{7}$$

Then we have the following:

$$\begin{array}{rcl} (\sum_i p_i)(cn) & \geq & (\sum_i p_i)(\sum_i m_i n_i) \text{ (by 6)} \\ & \geq & (\sum_i \sqrt{p_i m_i n_i})^2 \text{ (by Schwartz's inequality)} \\ & \geq & (\sum_i \sqrt{n_i n_i})^2 \text{ (by 7)} \\ & \geq & d^2 n^2 \text{ (by 5)} \end{array}$$

Hence $\sum_i p_i \geq \frac{d^2}{c} n = \frac{1}{c^{2^T-1}} n$.

$\square$

Finally we can prove Theorem 4.11 as follows. Suppose that, on input $x = [a_{n-1}, \ldots, a_0]$, the BVRAM will produce the output $y = [a_0, a_1, \ldots, a_{n-1}]$, in time $T$, with $W = O(n)$. Then by Lemma B.2 there exists a potential trace of length $2T-1$ with input $x$, output $y$ and some constant $c$. But the input $x = [a_{n-1}, \ldots, a_0]$ does not have increasing subchains longer than 1. Hence we get $1 \geq \frac{1}{c^{2^{2T-1}-1}} n$, i.e.

$$T \geq \frac{1 + \log(1 + \frac{\log n}{\log c})}{2}$$

For transposition, the input has increasing subchains of length at most $\sqrt{n}$. Hence we get $\sqrt{n} \geq \frac{1}{c^{2^{2T-1}-1}} n$, i.e.

$$T \geq \frac{1 + \log(1 + \frac{\log n}{2 \log c})}{2}$$